پروژه سیگنال و سیستم ها

پردیس زهرایی

99109777

(1

*** در این سوال من چندین حالت مختلف (از مقادیر کم تا زیاد) برای شیفت استفاده کرده ام و در نوتبوک تمامی این حالت ها موجود است و فایلی که فرستادم برای شیفت زیاد بود و در نوتبوک هم نوشتم که برای هر کدام از مقادیر شیفت به چه شکل تغییر پیدا کرده است.

متدولوژی:

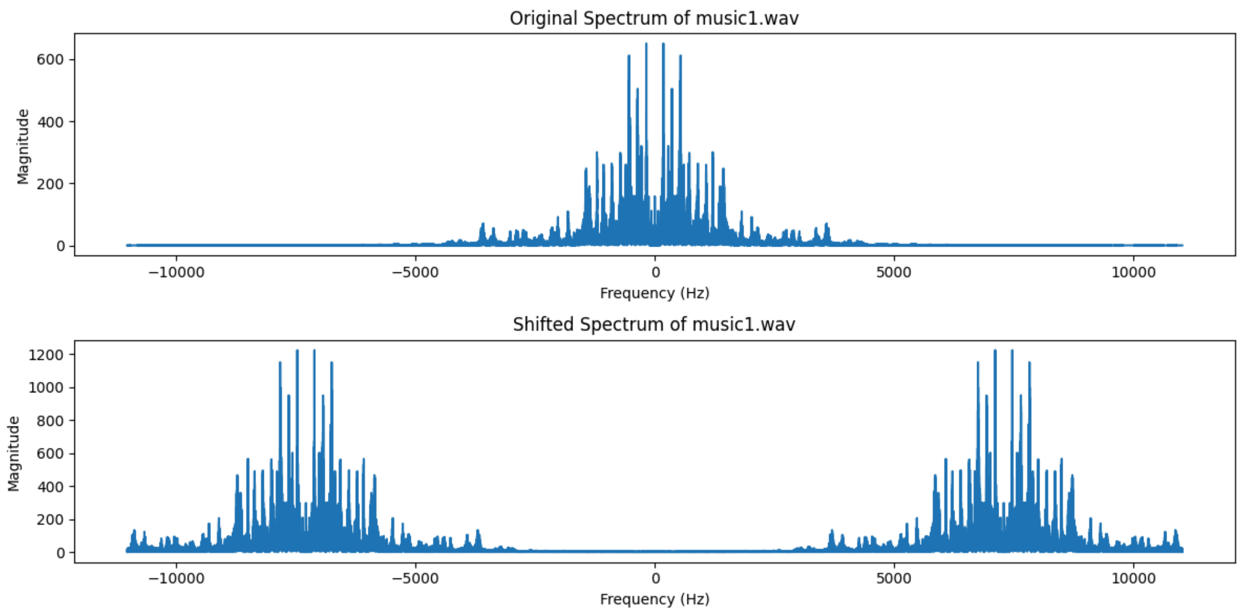
قسمت 1: تغییر فرکانس music1

* یک تغییر فرکانس را برای music1.wav در حوزه فرکانس اعمال کردم. (من مقادیر شیفت نمونه‌گیری متفاوتی را امتحان کردم و دیدم که چگونه آن را شدیدتر می‌کنم، تفاوت آن با نسخه اصلی بیشتر می‌شود و به هدف این سوال بیشتر میخورد و همه مراحل در نوت‌بوک موجود است و مقادیر داده شده دلخواه هستند و شدت آنها متفاوت است) فرکانس به صورت زیر بود:

```python
# Perform a more dramatic frequency shift
shift_amount = (3 * sample_rate1)   # Shift by 3 times the sample rate
n = len(freq_domain1)
shifted_freq_domain1 = np.zeros_like(freq_domain1)
shifted_freq_domain1[shift_amount:] = freq_domain1[:n-shift_amount]
shifted_freq_domain1[:shift_amount] = freq_domain1[n-shift_amount:]

# Convert back to time domain
shifted_time_domain1 = np.real(ifft(ifftshift(shifted_freq_domain1)))

# Normalize the shifted signal
shifted_time_domain1 = shifted_time_domain1 / np.max(np.abs(shifted_time_domain1))

# Play the shifted signal
print("Playing frequency-shifted music1.wav:")
ipd.display(ipd.Audio(shifted_time_domain1, rate=sample_rate1))

# Plot original and shifted spectra for comparison
def plot_spectrum(signal, sample_rate, title):
    spectrum = np.abs(fftshift(fft(signal)))
    freqs = np.linspace(-sample_rate/2, sample_rate/2, len(spectrum))
    plt.plot(freqs, spectrum)
    plt.title(title)
    plt.xlabel('Frequency (Hz)')
    plt.ylabel('Magnitude')

plt.figure(figsize=(12, 6))
plt.subplot(2, 1, 1)
plot_spectrum(music1, sample_rate1, 'Original Spectrum of music1.wav')
plt.subplot(2, 1, 2)
plot_spectrum(shifted_time_domain1, sample_rate1, 'Shifted Spectrum of music1.wav')
plt.tight_layout()
plt.show()
```

```
Playing frequency-shifted music1.wav:
0:03 / 0:09
```

که بعد اسپکتروم هم کشیدم:

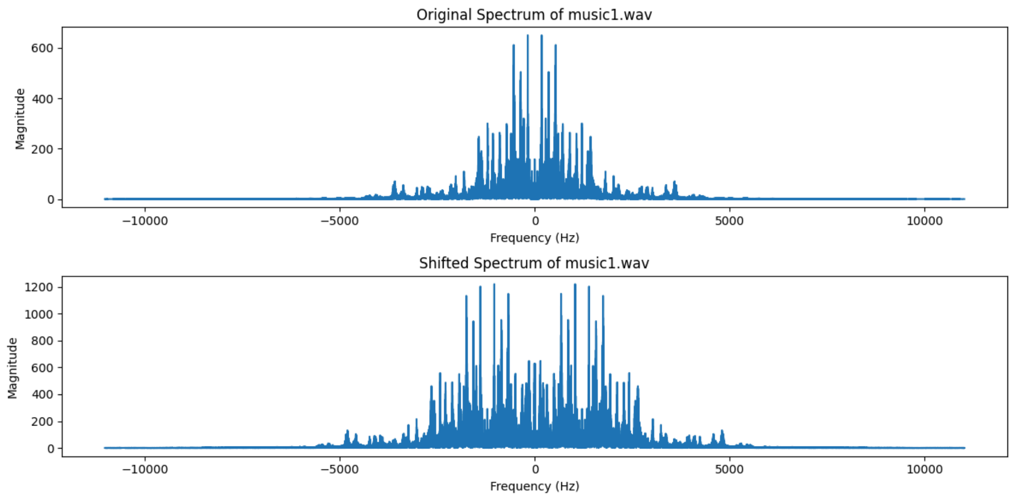


ولی حالات کمتری که صدا واضح تر بود بعد شیفت هم داشتم، مثل این:

```python
# Perform a more dramatic frequency shift
shift_amount = sample_rate1 // 2  # Shift by half the sample rate
n = len(freq_domain1)
shifted_freq_domain1 = np.zeros_like(freq_domain1)
shifted_freq_domain1[shift_amount:] = freq_domain1[:n-shift_amount]
shifted_freq_domain1[:shift_amount] = freq_domain1[n-shift_amount:]

# Convert back to time domain
shifted_time_domain1 = np.real(ifft(ifftshift(shifted_freq_domain1)))

# Normalize the shifted signal
shifted_time_domain1 = shifted_time_domain1 / np.max(np.abs(shifted_time_domain1))

# Play the shifted signal
print("Playing frequency-shifted music1.wav:")
ipd.display(ipd.Audio(shifted_time_domain1, rate=sample_rate1))

# Plot original and shifted spectra for comparison
def plot_spectrum(signal, sample_rate, title):
    spectrum = np.abs(fftshift(fft(signal)))
    freqs = np.linspace(-sample_rate/2, sample_rate/2, len(spectrum))
    plt.plot(freqs, spectrum)
    plt.title(title)
    plt.xlabel('Frequency (Hz)')
    plt.ylabel('Magnitude')

plt.figure(figsize=(12, 6))
plt.subplot(2, 1, 1)
plot_spectrum(music1, sample_rate1, 'Original Spectrum of music1.wav')
plt.subplot(2, 1, 2)
plot_spectrum(shifted_time_domain1, sample_rate1, 'Shifted Spectrum of music1.wav')
plt.tight_layout()
plt.show()
```

```
Playing frequency-shifted music1.wav:
```

0:03 / 0:09



در هنگام پخش، music تغییر یافته 1 به نظر می رسد distorted و کمی نامشخص است. این نشان داد که چگونه تغییر فرکانس می تواند به طور چشمگیری ویژگی های قابل درک یک سیگنال صوتی را با برهم زدن ساختار هارمونیک آن و حرکت اجزای فرکانس خارج از محدوده اصلی آنها تغییر دهد.

قسمت 2: ترکیب Shifted music1 با music2

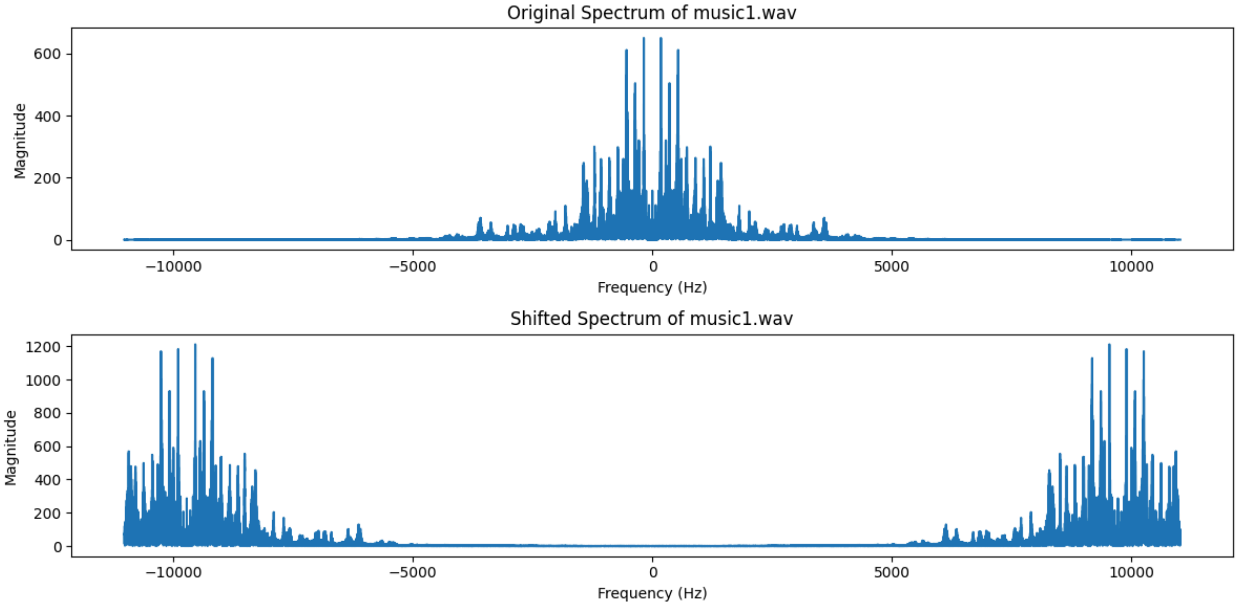
* من frequency domain music 1 تغییر یافته را به music2 اضافه کردم. سیگنال ترکیبی، هنگام پخش، عمدتاً مانند music2 به نظر می رسید.

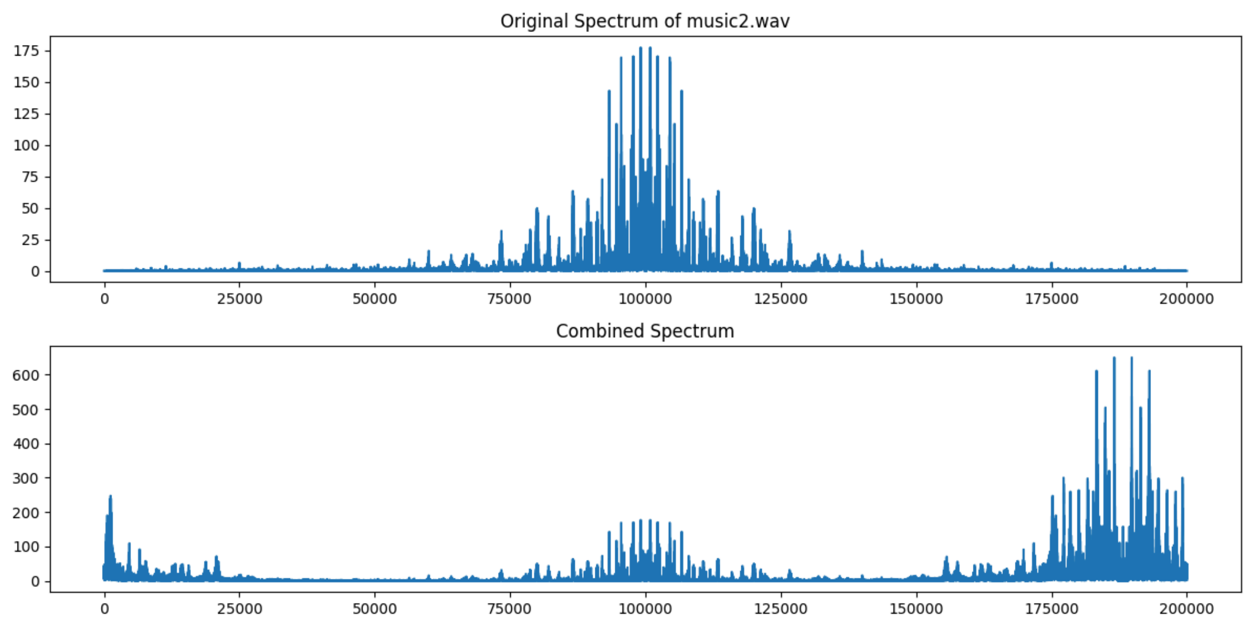
dominance music2 در خروجی را می توان به خاطر دلایل زیر دانست:

الف) ساختار منسجم و تغییر نکرده اجزای فرکانس music2.

ب) اجزای جابجا شده music1 بیشتر به نویز پس زمینه یا تغییرات کوچک کمک می کند تا صدای قابل تشخیص.

```python
# Perform a more dramatic frequency shift
shift_amount = (4 * sample_rate1)   # Shift by 4* the sample rate
n = len(freq_domain1)
shifted_freq_domain1 = np.zeros_like(freq_domain1)
shifted_freq_domain1[shift_amount:] = freq_domain1[:n-shift_amount]
shifted_freq_domain1[:shift_amount] = freq_domain1[n-shift_amount:]

# Convert back to time domain
shifted_time_domain1 = np.real(ifft(ifftshift(shifted_freq_domain1)))

# Normalize the shifted signal
shifted_time_domain1 = shifted_time_domain1 / np.max(np.abs(shifted_time_domain1))

# Play the shifted signal
print("Playing frequency-shifted music1.wav:")
ipd.display(ipd.Audio(shifted_time_domain1, rate=sample_rate1))

# Plot original and shifted spectra for comparison
def plot_spectrum(signal, sample_rate, title):
    spectrum = np.abs(fftshift(fft(signal)))
    freqs = np.linspace(-sample_rate/2, sample_rate/2, len(spectrum))
    plt.plot(freqs, spectrum)
    plt.title(title)
    plt.xlabel('Frequency (Hz)')
    plt.ylabel('Magnitude')

plt.figure(figsize=(12, 6))
plt.subplot(2, 1, 1)
plot_spectrum(music1, sample_rate1, 'Original Spectrum of music1.wav')
plt.subplot(2, 1, 2)
plot_spectrum(shifted_time_domain1, sample_rate1, 'Shifted Spectrum of music1.wav')
plt.tight_layout()
plt.show()

# Add the shifted frequency information of music1 with frequency information of music2
combined_freq_domain = shifted_freq_domain1 + freq_domain2

# Convert back to time domain
combined_time_domain = np.real(ifft(ifftshift(combined_freq_domain)))
```





بخش 3: تغییر فرکانس سیگنال ترکیبی

* من یک تغییر فرکانس دیگر را به سیگنال ترکیبی از قسمت 2 اعمال کردم (دفعه قبل مقدار شیفت را بیشتر گذاشته بودم این دفعه با مقدار کمتری شیفت داده ام).

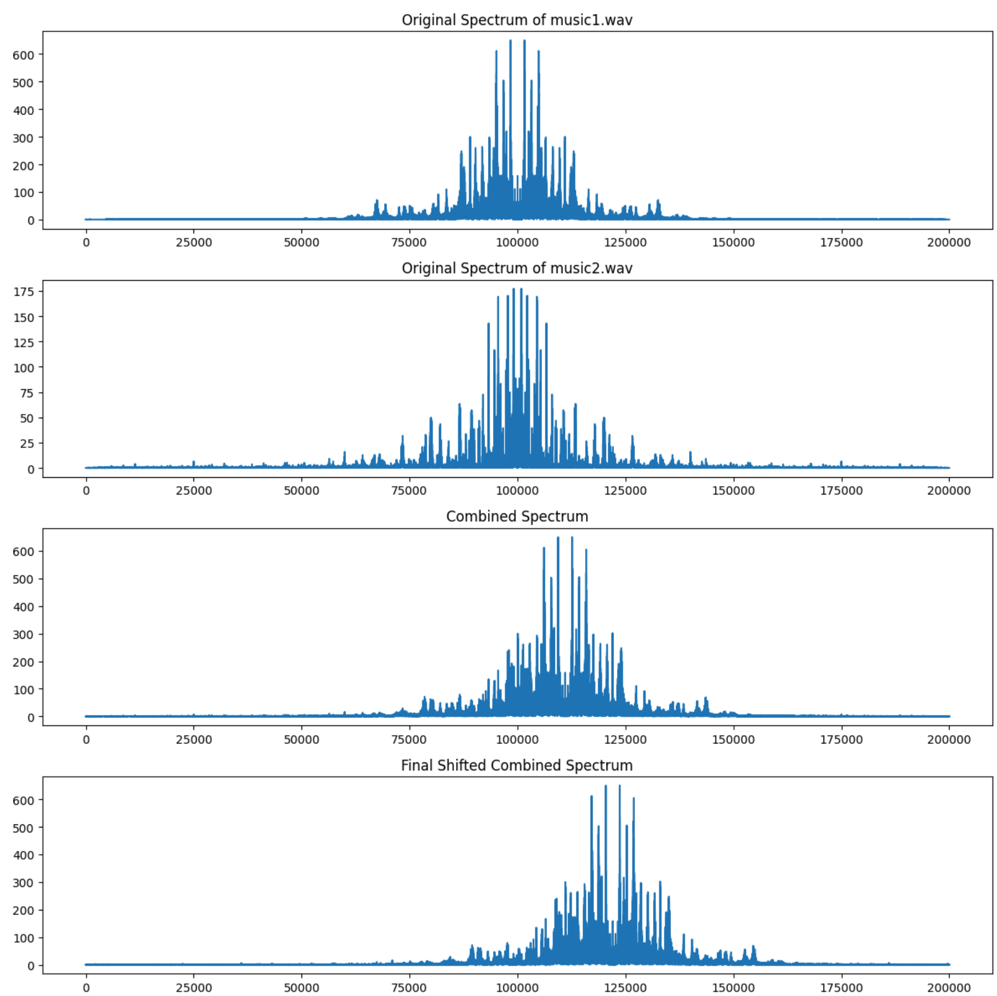
* صدای حاصل عمدتاً مانند music 1 به نظر می رسید، البته با مقداری distortion.

* این پدیده را می‌توان با موارد زیر توضیح داد:

الف) تغییر دوم که تغییر اولیه اعمال شده در music را تا حدی لغو می‌کند و اجزای آن را به موقعیت اصلی خود نزدیک‌تر می‌کند.

ب) به طور همزمان، مؤلفه‌های موزیک2 اولیه به less audible frequency ranges منتقل شدند.

```
# Perform initial frequency shift on music1
shift_amount = sample_rate1 //2  # Shift by half the sample rate
n = len(freq_domain1)
shifted_freq_domain1 = np.zeros_like(freq_domain1)
shifted_freq_domain1[shift_amount:] = freq_domain1[:n-shift_amount]
shifted_freq_domain1[:shift_amount] = freq_domain1[n-shift_amount:]

# Add the shifted frequency information of music1 with frequency information of music2
combined_freq_domain = shifted_freq_domain1 + freq_domain2

# Perform second frequency shift on the combined signal
shifted_combined_freq_domain = np.zeros_like(combined_freq_domain)
shifted_combined_freq_domain[shift_amount:] = combined_freq_domain[:n-shift_amount]
shifted_combined_freq_domain[:shift_amount] = combined_freq_domain[n-shift_amount:]

# Convert back to time domain
final_time_domain = np.real(ifft(ifftshift(shifted_combined_freq_domain)))

# Normalize the final signal
final_time_domain = final_time_domain / np.max(np.abs(final_time_domain))

# Play the final signal
print("Playing final shifted combined signal:")
ipd.display(ipd.Audio(final_time_domain, rate=sample_rate1))
```



نتیجه گیری:

1. تغییر فرکانس می تواند به طور قابل توجهی کیفیت صدا را تغییر دهد و صداهای قابل تشخیص را نامشخص یا نامفهوم کند.
2. هنگام ترکیب صدای جابجا شده و جابجا نشده، اجزای تغییرنکرده اغلب به دلیل ساختار هارمونیک دست نخورده غالب می شوند.
3. جابجایی های فرکانس متعدد می تواند منجر به بازیابی اطلاعات صوتی پنهان شده قبلی می شود.
(فایل های ارسالی با شیفت بالایی هستند برای همین موسیقی مقداری تغییر کرده، اگر منظور شیفت کمتر است، کافی است مقدار را کمتر کنیم ولی تحلیل به همان شکل است.)

(2
(2.1
گام به گام تبدیل فوریه گسسته معکوس (IDFT) از فرمول تبدیل فوریه گسسته (DFT) را پیدا میکنیم.
1: ما با معادله DFT شروع می کنیم:

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

2: هر دو طرف معادله را در e^(j(2π/N)km ضرب کرده که m یک عدد صحیح بین 0 و N-1 است:

$$X[k]e^{(j(2\pi/N)km)} = \sum_{n=0}^{N-1} x[n]e^{-j(2\pi/N)kn}e^{(j(2\pi/N)km)}$$

3: هر دو طرف را از k از 0 تا N-1 جمع کرده:

$$\sum_{k=0}^{N-1} X[k]e^{(j(2\pi/N)km)} = \sum_{k=0}^{N-1}\sum_{n=0}^{N-1} x[n]e^{-j(2\pi/N)kn}e^{(j(2\pi/N)km)}$$

4: سمت راست را می توان به این شکل نوشت:

$$\sum_{n=0}^{N-1} x[n] \sum_{k=0}^{N-1} e^{j(2\pi/N)k(m-n)}$$

5: از ویژگی complex exponentials استفاده کرده:

$$\sum_{k=0}^{N-1} e^{j(2\pi/N)k(m-n)}$$

- If m = n, it equals N
- If m ≠ n, it equals 0

این به این دلیل است که مجموع N نقطه با فاصله مساوی در اطراف دایره واحد مختلط صفر است مگر اینکه نقاط همه در 1+0j باشند.

6: با استفاده از این ویژگی، معادله ما به صورت زیر ساده می شود:

$$\sum_{k=0}^{N-1} X[k]e^{(j(2\pi/N)km)} = Nx[m]$$

7: تقسیم هر دو طرف بر N به ما می دهد:

$$x[m] = \frac{1}{N}\sum_{k=0}^{N-1} X[k]e^{(j(2\pi/N)km)}$$

مرحله 8: تعمیم برای همه n از آنجایی که m یک عدد صحیح دلخواه بین 0 و N-1 بود، می توانیم m را با n جایگزین کنیم تا شکل کلی را بدست آوریم:

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

این فرمول تبدیل فوریه گسسته معکوس (IDFT) است.

2.2) برای پیاده سازی این بخش من به 2 صورت عمل کردم:

الف) radix-2 Cooley-Tukey FFT algorithm

با استفاده از Divide and Conquer الگوریتم ورودی اندازه N را به دو نیمه تقسیم می کند. این کار به صورت بازگشتی انجام می شود تا زمانی که به آرایه های تک عنصری برسیم. و ساختار بازگشتی به این شکل است که ورودی را تقسیم کرده FFT هر نیمه را به صورت بازگشتی محاسبه کرده و سپس نتایج را ترکیب می کنیم. این تقسیم بازگشتی پیچیدگی را از O(N^2) DFT ساده به O(N log N) کاهش می دهد. این الگوریتم به ویژه برای ورودی‌های با اندازه توان ۲ کارآمد است زیرا تقسیم همیشه یکنواخت است، اما می‌توان آن را با برخی تغییرات برای اندازه‌های دیگر تطبیق داد.

```python
import cmath

def fft(x):
    N = len(x)

    # Base case: if the input has only one element, return it
    if N <= 1:
        return x

    # Recursive case
    # Split the input into even and odd indices
    even = fft(x[0::2])
    odd = fft(x[1::2])

    # Combine the results
    T = [cmath.exp(-2j * cmath.pi * k / N) * odd[k] for k in range(N // 2)]
    return [even[k] + T[k] for k in range(N // 2)] + [even[k] - T[k] for k in range(N // 2)]

def generate_signal(N):
    return [cmath.exp(2j * cmath.pi * k / N) for k in range(N)]

N = 16
x = generate_signal(N)
X = fft(x)

print("Input signal:")
print(x)
print("\nFFT result:")
print(X)
```

```
Input signal:
[(1+0j), (0.9238795325112867+0.3826834323650898j), (0.7071067811865476+0.7071067811865475j), (0.38268343236508984+0.9238795325112867j), (6.123233995736766e-17+1j), (-0.3826834

FFT result:
[(-8.99620797152345e-16-4.406877377903816e-17j), (16-1.5771969893796585e-15j), (6.652624085152891e-16+8.321236844764102e-17j), (-2.051424657947913e-16-1.699455831017226e-16j),
```

ب) تابع DFT که بر اساس هر اندازه ورودی کار میکند و طبق فرمول DFT، هر جزء فرکانس X[k] را با جمع کردن حاصلضرب هر نمونه ورودی محاسبه می‌کند. تابع idft عملیات معکوس را انجام می دهد و سیگنال اصلی را از اجزای فرکانس آن بازسازی می کند.

```python
def dft(x):
    N = len(x)
    X = np.zeros(N, dtype=complex)

    for k in range(N):
        for n in range(N):
            X[k] += x[n] * cmath.exp(-2j * cmath.pi * k * n / N)

    return X

def idft(X):
    N = len(X)
    x = np.zeros(N, dtype=complex)

    for n in range(N):
        for k in range(N):
            x[n] += X[k] * cmath.exp(2j * cmath.pi * k * n / N)
        x[n] /= N

    return x
```

که به صورت زیر الگوریتم را چک هم کردیم که مناسب بود

```python
for N in [10, 15, 20]:
    print(f"\nTesting with N = {N}")
    x = generate_signal(N)

    print("Original signal:")
    print(x)

    X = dft(x)
    print("\nDFT result:")
    print(X)

    x_recovered = idft(X)
    print("\nRecovered signal after IDFT:")
    print(x_recovered)

    # Check if the recovered signal is close to the original
    if np.allclose(x, x_recovered):
        print("\nSuccessfully recovered the original signal!")
    else:
        print("\nWarning: Recovered signal differs from the original.")
```

```
Testing with N = 10
Original signal:
[(1+0j), (0.8090169943749475+0.5877852522924731j), (0.30901699437494745+0.9510565162951535j),

DFT result:
[-5.55111512e-16+0.00000000e+00j  1.00000000e+01+0.00000000e+00j
 -6.66133815e-16+8.88178420e-16j -3.88578059e-16-2.22044605e-16j
  3.33066907e-16-3.33066907e-16j -3.33066907e-16-5.55111512e-16j
 -2.22044605e-16-7.23345702e-16j  1.55431223e-15+3.44169138e-15j
  6.66133815e-16-1.33226763e-15j  6.66133815e-15+3.21964677e-15j]

Recovered signal after IDFT:
[ 1.        +4.38368021e-16j  0.80901699+5.87785252e-01j
  0.30901699+9.51056516e-01j -0.30901699+9.51056516e-01j
 -0.80901699+5.87785252e-01j -1.        -6.16003705e-16j
 -0.80901699-5.87785252e-01j -0.30901699-9.51056516e-01j
  0.30901699-9.51056516e-01j  0.80901699-5.87785252e-01j]

Successfully recovered the original signal!
```

2.3) برای این بخش هم دوباره یکبار با الگوریتم:

radix-2 Cooley-Tukey IFFT algorithm

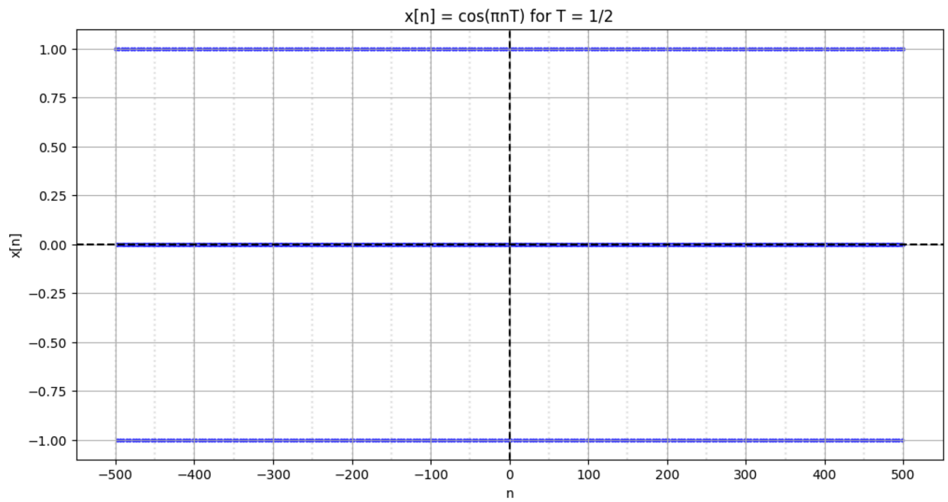
و یکبار هم همان حالت پیاده شده تابع ریاضی آن که در ب هم آورده بودیم.

```python
def ifft(X):
    N = len(X)
    if N <= 1:
        return X
    even = ifft(X[0::2])
    odd = ifft(X[1::2])
    T = [cmath.exp(2j * cmath.pi * k / N) * odd[k] for k in range(N // 2)]
    return [even[k] + T[k] for k in range(N // 2)] + [even[k] - T[k] for k in range(N // 2)]

def generate_signal(N):
    return [cmath.exp(2j * cmath.pi * k / N) for k in range(N)]


N = 4
x = generate_signal(N)

print("Original signal:")
print(x)

X = fft(x)
print("\nFFT result:")
print(X)

x_recovered = ifft(X)
x_recovered = [val / N for val in x_recovered]
print("\nRecovered signal after IFFT:")
print(x_recovered)

# Check if the recovered signal is close to the original
if all(abs(a - b) < 1e-10 for a, b in zip(x, x_recovered)):
    print("\nSuccessfully recovered the original signal!")
else:
    print("\nWarning: Recovered signal differs from the original.")
```

```
Original signal:
[(1+0j), (6.123233995736766e-17+1j), (-1+1.2246467991473532e-16j), (-1.8369701987210297e-16-1j)]

FFT result:
[(-1.2246467991473532e-16+1.2246467991473532e-16j), (4-2.4492935982947064e-16j), (1.2246467991473532e-16+1.2246467991473532e-16j), 0j]

Recovered signal after IFFT:
[(1+0j), (6.123233995736767e-17+1j), (-1+1.2246467991473532e-16j), (-1.8369701987210297e-16-1j)]

Successfully recovered the original signal!
```

```python
def idft(X):
    N = len(X)
    x = np.zeros(N, dtype=complex)

    for n in range(N):
        for k in range(N):
            x[n] += X[k] * cmath.exp(2j * cmath.pi * k * n / N)
        x[n] /= N

    return x
```

2.4) به این صورت است:




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

# Parameters
T = 1/2
N = 1000

# Create the time vector
n = np.arange(-500, 501)
t = n * T

# Create the signal vector
x = np.cos(np.pi * t)

# Plot the signal
plt.figure(figsize=(12, 6))
plt.scatter(n, x, s=5, c='b', alpha=0.7)  # Use scatter plot with small points
plt.title('x[n] = cos(πnT) for T = 1/2')
plt.xlabel('n')
plt.ylabel('x[n]')
plt.grid(True)
plt.axhline(y=0, color='k', linestyle='--')
plt.axvline(x=0, color='k', linestyle='--')

plt.xticks(np.arange(-500, 501, 100))

# Add vertical lines to emphasize discreteness
for i in range(-500, 501, 50):
    plt.axvline(x=i, color='gray', alpha=0.2, linestyle=':')

plt.show()
```

(2.5

تبدیل فوریه یک تابع به صورت زیر تعریف می شود:

$$X(\omega) = \int_{-\infty}^{\infty} x(t)\, e^{(-j\omega t)} dt$$

برای x(t) = cos(πt) ، می توانیم از فرمول اویلر استفاده کنیم:

$$cos(\pi t) = (e^{(j\pi t)} + e^{(-j\pi t)}) / 2$$

$$X(\omega) = \int_{-\infty}^{\infty} (e^{(j\pi t)} + e^{(-j\pi t)})\, e^{(-j\omega t)} / 2dt$$

$$X(\omega) = \frac{1}{2} \int_{-\infty}^{\infty} (e^{(j\pi t - j\omega t)} + e^{(-j\omega t - j\pi t)})\, dt = \frac{1}{2} \int_{-\infty}^{\infty} (e^{(j(\pi-\omega)t)} + e^{(-j(\omega+\pi)t)})\, dt$$

تابع دلتا را از انتگرال e^(jat) از منفی بی نهایت تا مثبت بی نهایت 2πδ(a) است، جایگزین می کنیم

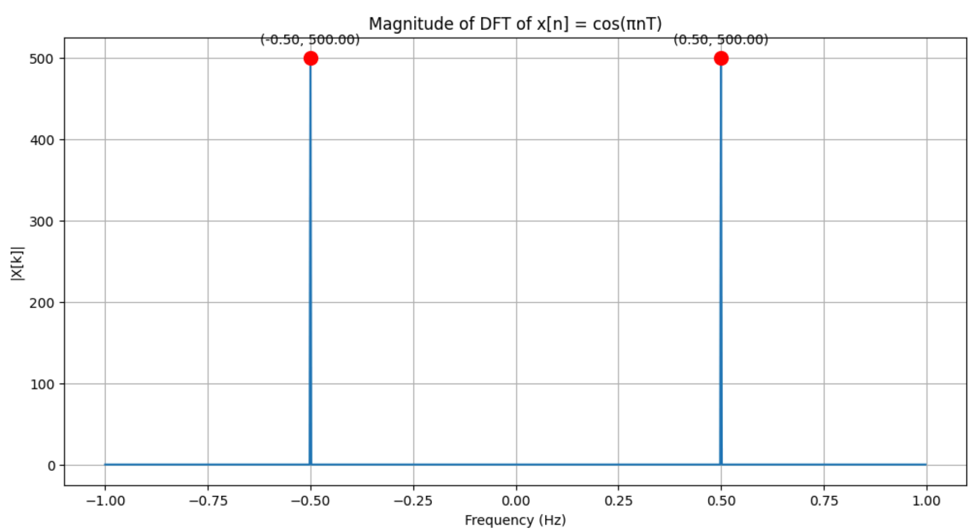
X(ω) = (1/2) * 2π * [δ(π-ω) + δ(-π-ω)] = π [δ(ω-π) + δ(ω+π)]

X(ω) = π [δ(ω-π) + δ(ω+π)]

این نتیجه نشان می‌دهد که یک تابع کسینوس با فرکانس π انرژی دقیقاً در دو نقطه در حوزه فرکانس متمرکز است: π+ و π-.

(2.6

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

T = 1/2
N = 1000

# Create the time vector
n = np.arange(-500, 500)
t = n * T

# Create the signal vector
x = np.cos(np.pi * t)

# Compute the FFT
X = np.fft.fft(x)

# Shift the zero frequency component to the center
X_shifted = np.fft.fftshift(X)

# Create the frequency vector
freq = np.fft.fftshift(np.fft.fftfreq(N, T))

# Plot the magnitude of the DFT
plt.figure(figsize=(12, 6))
plt.plot(freq, np.abs(X_shifted))
plt.title('Magnitude of DFT of x[n] = cos(πnT)')
plt.xlabel('Frequency (Hz)')
plt.ylabel('|X[k]|')
plt.grid(True)

# Highlight the peaks
peak_indices = np.where(np.abs(X_shifted) > np.max(np.abs(X_shifted))/2)[0]
plt.plot(freq[peak_indices], np.abs(X_shifted[peak_indices]), 'ro', markersize=10)

for idx in peak_indices:
    plt.annotate(f'({freq[idx]:.2f}, {np.abs(X_shifted[idx]):.2f})',
                 (freq[idx], np.abs(X_shifted[idx])),
                 textcoords="offset points",
                 xytext=(0,10),
                 ha='center')

plt.show()
```



که متوجه می شویم به درستی کار می کند

اگر این نتیجه را با تبدیل فوریه پیوسته [$X(\omega) = \pi [\delta(\omega-\pi) + \delta(\omega+\pi)]$ مقایسه کنیم:

شباهت ها:

تبدیل پیوسته دارای impulse در $\omega = \pm \pi$ rad/s است که با $f = \pm 0.5$ هرتز مطابقت دارد.

در نمودار DFT ما ± 0.5 هرتز را می بینیم.

تبدیل پیوسته دارای(perfect impulses (delta function است.

هم تبدیل پیوسته و هم DFT ما باید در حدود 0 هرتز متقارن باشند.

تفاوت ها:

DFT ما یک نسخه نمونه از تبدیل پیوسته را ارائه می دهد.

دامنه های DFT در مقایسه با تبدیل پیوسته با N/2 مقیاس بندی می شوند.

به دلیل طول سیگنال محدود، به جای impulses های کامل،peaks را می بینیم.

DFT تقریب خوبی از تبدیل فوریه پیوسته ارائه می‌کند و اجزای فرکانس ضروری سیگنال کسینوس ما را می‌گیرد.

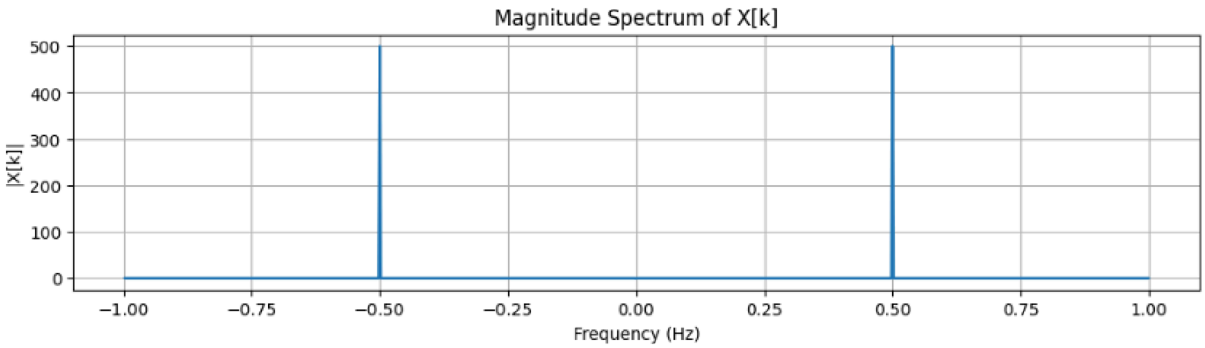
* در این تحلیل، ما طیف بزرگی magnitude را برای تابع کسینوس ترسیم می کنیم. طیف بزرگی در اینجا کافی است زیرا مولفه های فرکانس موجود در سیگنال را نشان می دهد که تمرکز اصلی ما است. طیف فاز در مولفه فرکانس مثبت (مرتبط با فرکانس کسینوس) و π (یا -π) در مولفه فرکانس منفی صفر خواهد بود. تمام اجزای فرکانس دیگر اساساً magnitude صفر خواهند داشت و مقادیر فاز آنها را از نظر عددی ناپایدار یا بی معنی می کند. طیف magnitude به تنهایی مکان و قدرت مولفه های فرکانس اصلی را به وضوح نشان می دهد.

بدون استفاده از نامپای و با پیاده سازی نیز به این شکل می شود:

```
def dft(x):
    N = len(x)
    n = np.arange(N)
    k = n.reshape((N, 1))
    e = np.exp(-2j * np.pi * k * n / N)
    return np.dot(e, x)

def fftfreq(n, d=1.0):
    if n % 2 == 0:
        N = n // 2 + 1
    else:
        N = (n + 1) // 2

    val = 1.0 / (n * d)
    results = np.arange(N, dtype=float)  # Changed to float
    results[1:] *= val
    if n % 2 == 0:
        results = np.concatenate([results, -results[1:-1][::-1]])
    else:
        results = np.concatenate([results, -results[1:][::-1]])

    return results
```

```
# Compute the DFT
X = dft(x)

# Create frequency vector
freq = fftfreq(N, T)

# Compute magnitude and phase
magnitude = np.abs(X)
phase = np.angle(X)

# Plot the magnitude spectrum
plt.figure(figsize=(12, 6))
plt.subplot(2, 1, 1)
plt.plot(freq, magnitude)
plt.title('Magnitude Spectrum of X[k]')
plt.xlabel('Frequency (Hz)')
plt.ylabel('|X[k]|')
plt.grid(True)

# Print the maximum value of |X[k]|
print(f"Maximum value of |X[k]|: {np.max(np.abs(X)):.2f}")
```

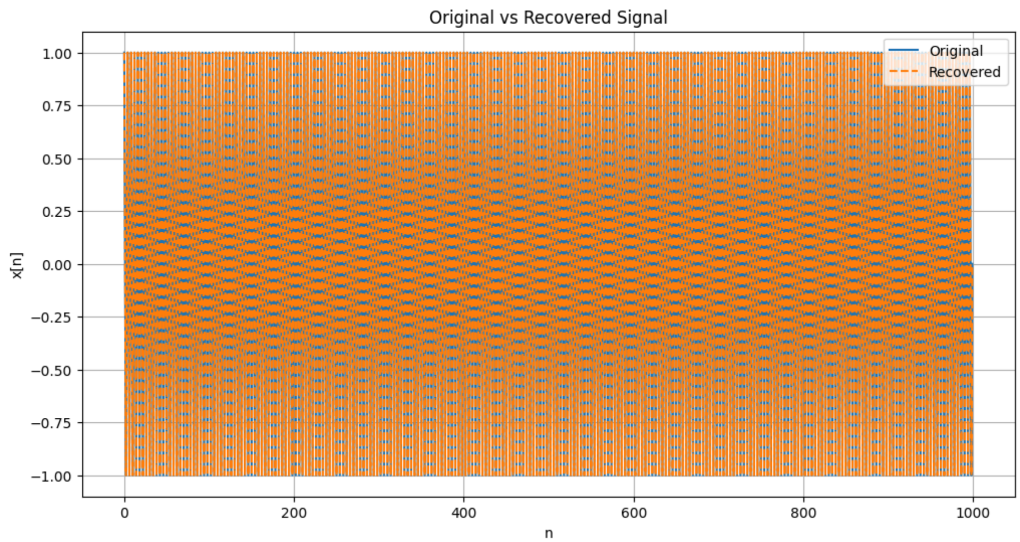
```
Maximum value of |X[k]|: 500.00
```



(2.7

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

# Parameters
T = 1/2
N = 1000  # Can be any number

# Create the signal vector
n = np.arange(N)
x = np.cos(np.pi * n * T)

# Compute FFT
X = np.fft.fft(x)

# Compute IFFT
x_recovered = np.fft.ifft(X)

# Check if recovered signal is equal to original
is_equal = np.allclose(x, x_recovered.real, atol=1e-10)

print(f"Is the recovered signal equal to the original? {is_equal}")
print(f"Maximum difference: {np.max(np.abs(x - x_recovered.real))}")

# Plot original and recovered signals
plt.figure(figsize=(12, 6))
plt.plot(n, x, label='Original')
plt.plot(n, x_recovered.real, '--', label='Recovered')
plt.title('Original vs Recovered Signal')
plt.xlabel('n')
plt.ylabel('x[n]')
plt.legend()
plt.grid(True)
plt.show()
```

```
Is the recovered signal equal to the original? True
Maximum difference: 4.440892098500626e-16
```

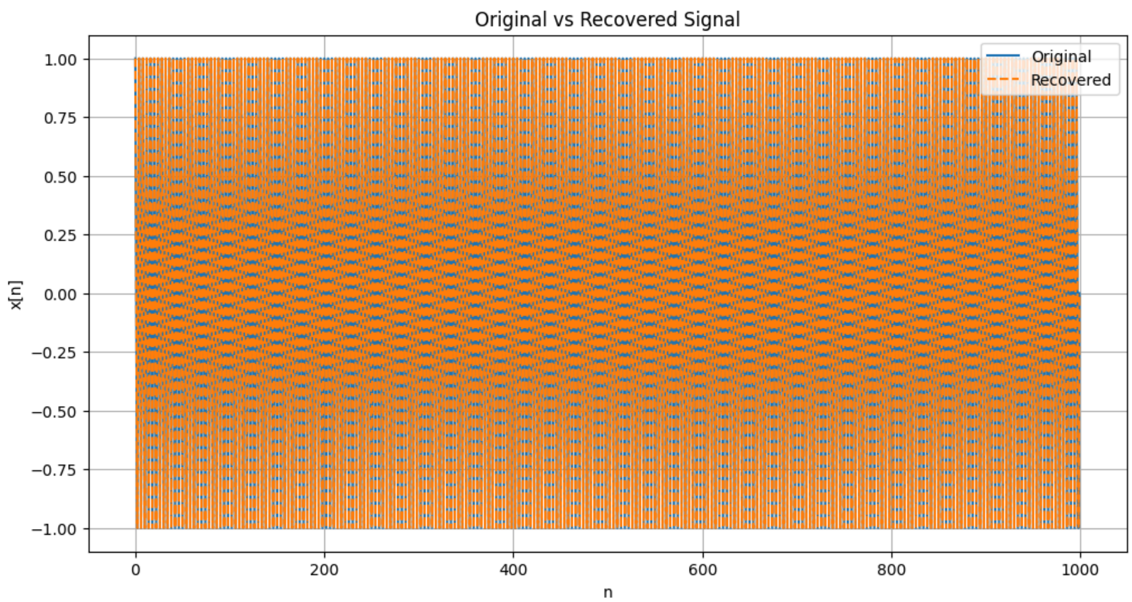


بله با دقت خوبی برابر است و recover می توان کرد

اگر با توابعی که خودمان پیاده کردیم برویم هم همین اتفاق میفتد و فرقی نمی کند

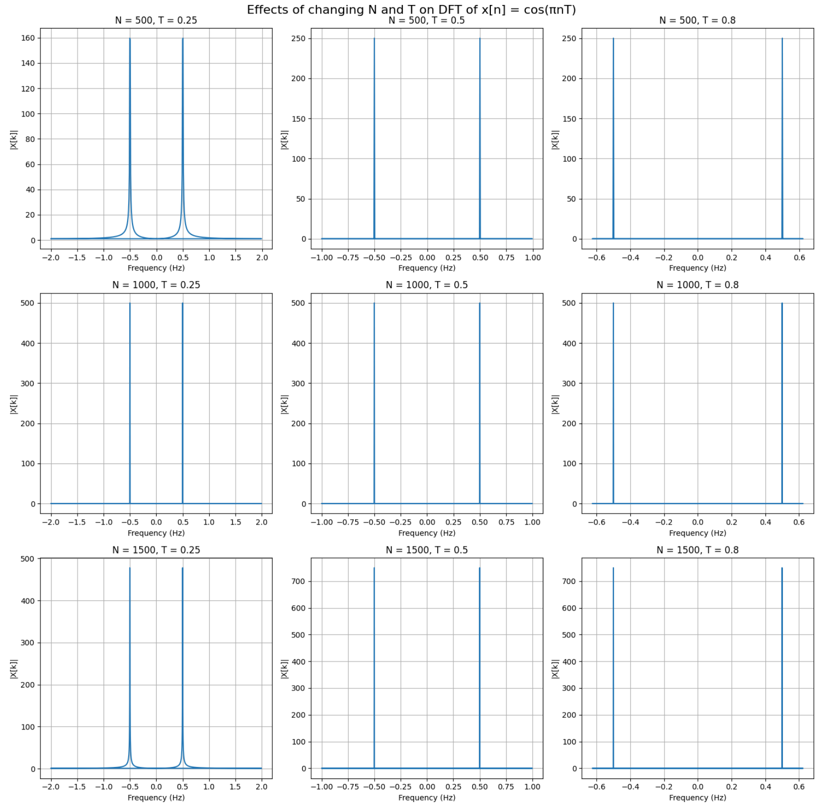
```python
def dft(x):
    N = len(x)
    n = np.arange(N)
    k = n.reshape((N, 1))
    e = np.exp(-2j * np.pi * k * n / N)
    return np.dot(e, x)

def idft(X):
    N = len(X)
    n = np.arange(N)
    k = n.reshape((N, 1))
    e = np.exp(2j * np.pi * k * n / N)
    return np.dot(e, X) / N

# Parameters
T = 1/2
N = 1000  # Can be any number

# Create the signal vector
n = np.arange(N)
x = np.cos(np.pi * n * T)

# Compute DFT
X = dft(x)

# Compute IDFT
x_recovered = idft(X)

# Check if recovered signal is equal to original
is_equal = np.allclose(x, x_recovered.real, atol=1e-10)
print(f"Is the recovered signal equal to the original? {is_equal}")
print(f"Maximum difference: {np.max(np.abs(x - x_recovered.real))}")

# Plot original and recovered signals
plt.figure(figsize=(12, 6))
plt.plot(n, x, label='Original')
plt.plot(n, x_recovered.real, '--', label='Recovered')
plt.title('Original vs Recovered Signal')
plt.xlabel('n')
plt.ylabel('x[n]')
plt.legend()
plt.grid(True)
plt.show()
```

```
Is the recovered signal equal to the original? True
Maximum difference: 5.252917743558e-13
```

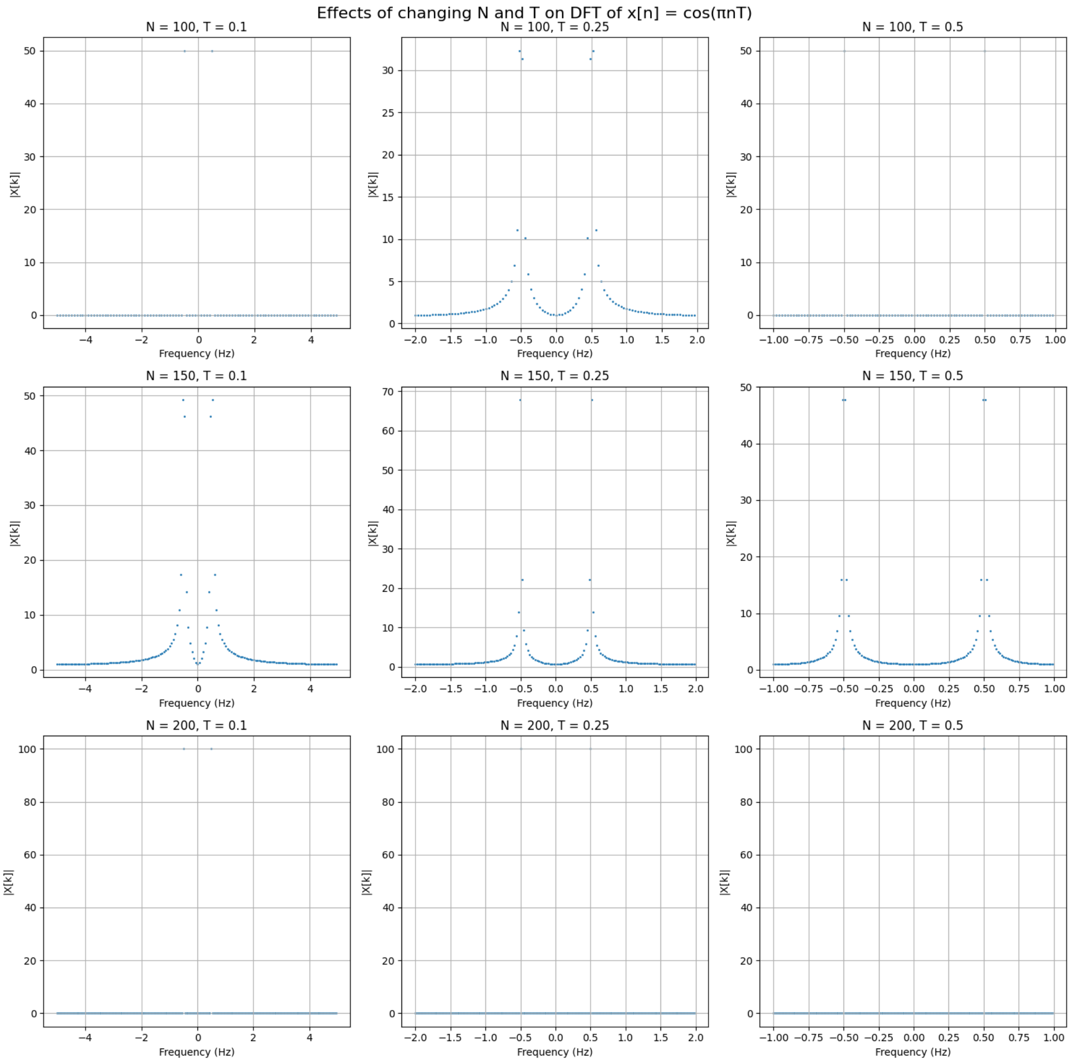


(2.8

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

def compute_and_plot_dft(N, T, ax):
    n = np.arange(N)
    x = np.cos(np.pi * n * T)
    X = np.fft.fft(x)
    freq = np.fft.fftfreq(N, T)

    ax.plot(freq, np.abs(X))
    ax.set_title(f'N = {N}, T = {T}')
    ax.set_xlabel('Frequency (Hz)')
    ax.set_ylabel('|X[k]|')
    ax.grid(True)

# Create a 3x3 grid of subplots
fig, axs = plt.subplots(3, 3, figsize=(15, 15))
fig.suptitle('Effects of changing N and T on DFT of x[n] = cos(πnT)', fontsize=16)

# Different values of N and T to test
N_values = [500, 1000, 1500]
T_values = [1/4, 1/2, 0.8]

for i, N in enumerate(N_values):
    for j, T in enumerate(T_values):
        compute_and_plot_dft(N, T, axs[i, j])

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

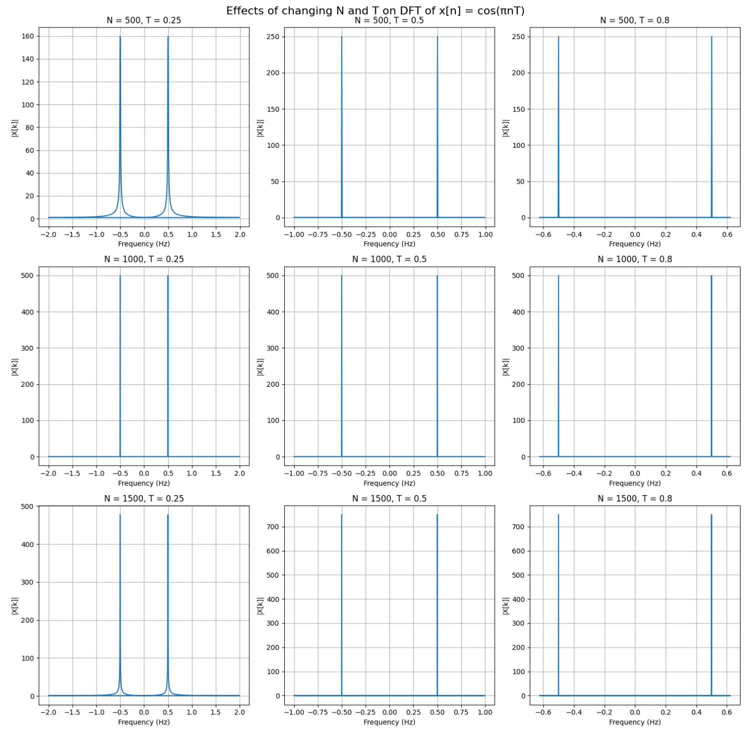


یا اگر به صورت scatter بخواهیم ببینیم هم:



اگر همه را خودمان پیاده کنیم نیز به همان شکل است:

```
def dft(x):
    N = len(x)
    n = np.arange(N)
    k = n.reshape((N, 1))
    e = np.exp(-2j * np.pi * k * n / N)
    return np.dot(e, x)

def fftfreq(n, d=1.0):
    val = 1.0 / (n * d)
    results = np.empty(n, float)
    N = (n-1) // 2 + 1
    p1 = np.arange(0, N)
    results[:N] = p1
    p2 = np.arange(-(n//2), 0)
    results[N:] = p2
    return results * val

def compute_and_plot_dft(N, T, ax):
    n = np.arange(N)
    x = np.cos(np.pi * n * T)
    X = dft(x)
    freq = fftfreq(N, T)
    ax.plot(freq, np.abs(X))
    ax.set_title(f'N = {N}, T = {T}')
    ax.set_xlabel('Frequency (Hz)')
    ax.set_ylabel('|X[k]|')
    ax.grid(True)
```



که اگر بخواهیم کمی تحلیل کنیم:

اثر تغییر N (تعداد نمونه):

با افزایش N:

- وضوح فرکانس بهبود یافته (peaks باریک تر و مشخص تر)
- peak magnitudes بالاتر
- شکل کلی طیف مشابه باقی می ماند

اثر تغییر T (دوره نمونه برداری):

با افزایش T:

- peaks به فرکانس صفر نزدیک تر می شوند
- محدوده فرکانس کوچک می شود
- دامنه peaks افزایش می یابد

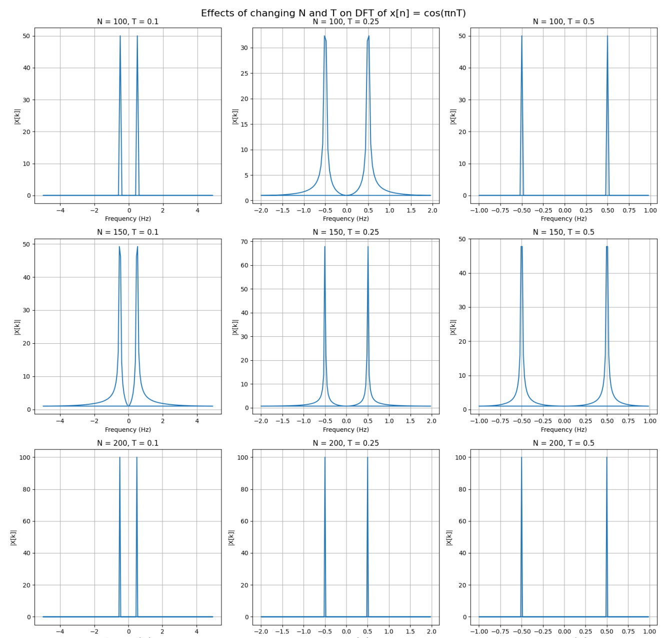
با افزایش N در ستون‌ها، peak magnitudes افزایش می‌یابد و وضوح کلی بهبود می‌یابد.

با افزایش T در ردیف‌ها، peak ها از هم جدا می‌شوند و محدوده فرکانس باریک می‌شود و روی فرکانس‌های پایین‌تر تمرکز می‌کند.

- For T = 0.25: Peaks at ±2 Hz
- For T = 0.5: Peaks at ±1 Hz
- For T = 0.8: Peaks at ±0.625 Hz

این با مکان های peak مورد انتظار $\pm\frac{1}{2T}$ برای تابع کسینوس سازگار است.

می توان این کار را برای مقادیر N, T دلخواه و مختلف نیز انجام داد که نتیجه مشابه می گیرد ولی به طور کلی خیلی وابسته به مقدار T,N دارد که شکل به چطوری می شود چون که در رابطه هم T , N هر دو با هم تاثیرگذار هستند و رابطه بین آنها در شکل هم سهیم است و وابسته به مقدار T,N ممکن است تعداد زیادی نقطه ناصفر داشته باشیم یا تعداد محدودی باشند و خیلی نمی توان تصمیم کلی ای بدون دانستن دقیق T,N و ارتباطی که باهم میتوانند روی کل بگذارند نظر قطعی داد.



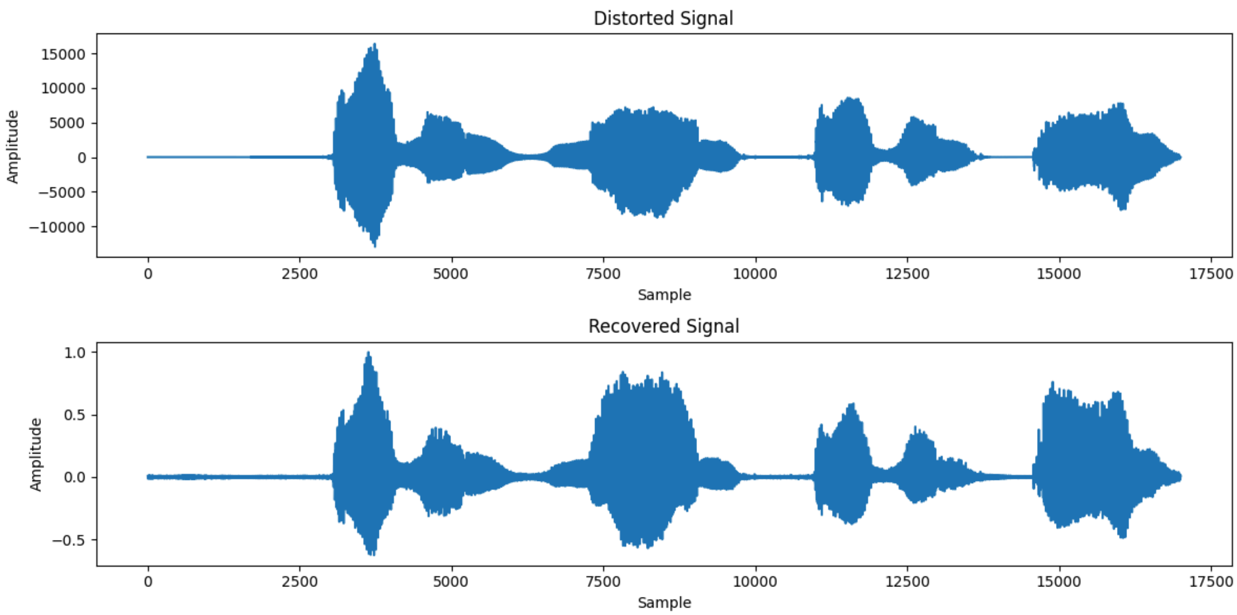
(3

برای این سوال من 2 روش را امتحان کردم که روش دوم بهتر بود و نتیجه بهتری داشت.

الف) چون باید اندازه های فایل ها یکسان میشد، من یکی را بر حسب طول مینیمم تریم کردم، که به این حالت شد:

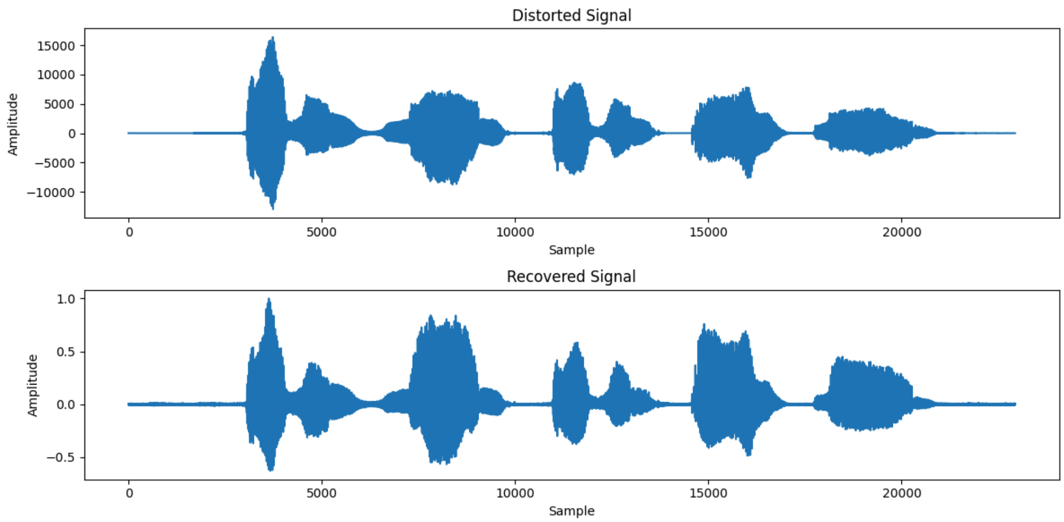
سیگنال های صوتی تمیز (clean1.wav) و تحریف شده (distorted1.wav، distorted2.wav) را لود کردم و همه سیگنال‌ها نرخ نمونه‌برداری یکسانی دارند. برای جلوگیری از مشکلات پردازش، سیگنال‌ها را به همان طول تریم کردم. FFT هر دو حالت clean1 و distorted1 را محاسبه کردم. پاسخ فرکانسی را با تقسیم FFT distorted1 بر FFT clean1 محاسبه کردم. سپس معکوس پاسخ فرکانسی محاسبه شده را محاسبه کردم و برای بازیابی سیگنال، پاسخ فرکانس معکوس را به FFT از distorted2 اعمال کردم.در نهایت هم از FFT معکوس برای تبدیل نتیجه به حوزه زمان استفاده کردم.

این روش فرض می‌کند که distortion به صورت linear and time-invariant است و به ما امکان می‌دهد آن را با یک پاسخ فرکانسی واحد مشخص کنیم. با اعمال معکوس این پاسخ به یک سیگنال distorted جدید، هدف ما حذف اثرات distortion و بازیابی سیگنال تمیز اصلی است. اثربخشی این رویکرد بستگی به این دارد که پاسخ فرکانسی محاسبه شده چقدر distortion واقعی سیستم را نشان می‌دهد، و چقدر distortion در سیگنال‌های مختلف پردازش شده توسط یک سیستم مشابه است.



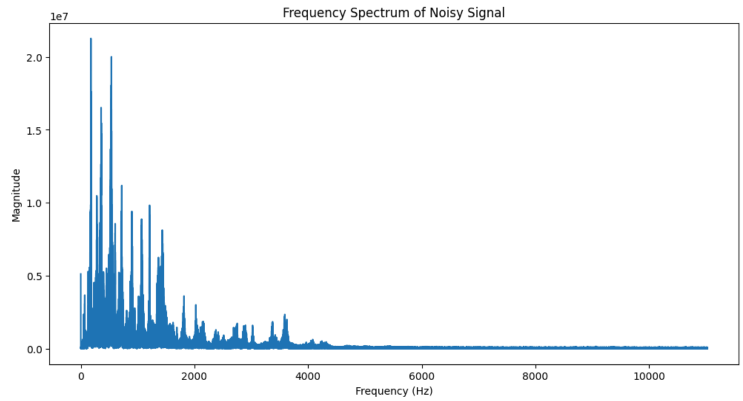
ب) اما یک راه بهتر به جای کوتاه کردن تمام سیگنال ها به حداقل طول، می توانیم از zero-padding استفاده کنیم تا مطمئن شویم همه سیگنال ها دارای طول یکسان هستند بدون اینکه هیچ اطلاعاتی از دست بدهند. علاوه بر این، ما با افزودن یک مقدار اپسیلون کوچک هنگام محاسبه پاسخ فرکانس معکوس، regularization را اجرا می‌کنیم، که به جلوگیری از تقسیم بر صفر و کاهش noise amplification در سیگنال بازیابی شده کمک می‌کند. در این روش ابتدا فایل ها را به نوع شناور تبدیل کرده و بعد zero-padding تمام سیگنال‌ها به طول یکسان شده و بعد پاسخ فرکانسی سیستم محاسبه شده و بعد پاسخ فرکانس معکوس با regularization محاسبه شده و بعد سیگنال clean2 از طریق ضرب در حوزه فرکانس بازیابی شده و FFT معکوس برای به دست آوردن سیگنال بازیابی شده در حوزه زمان اعمال می شود و در نهایت normalizing و ذخیره سیگنال بازیابی شده. فایل خروجی این مرحله کیفیت بالایی دارد و مشکل راهکار قبلی را دیگر ندارد:

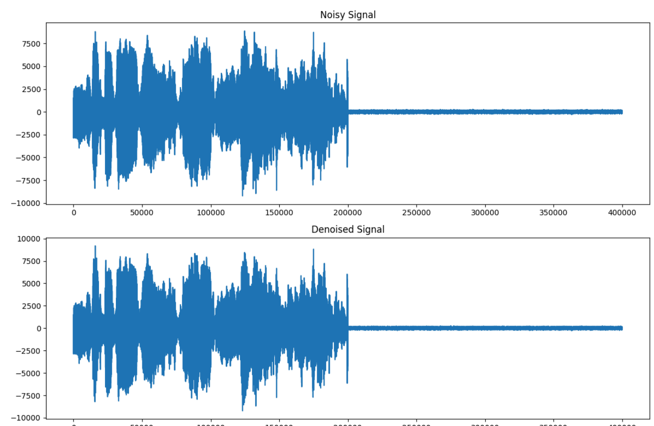
```python
def zero_pad(signals):
    max_length = max(len(s) for s in signals)
    return [np.pad(s, (0, max_length - len(s)), 'constant') for s in signals]

fs_clean1, clean1 = read_wav('clean1.wav')
fs_distorted1, distorted1 = read_wav('distorted1.wav')
fs_distorted2, distorted2 = read_wav('distorted2.wav')

# Ensure all signals have the same sampling rate
assert fs_clean1 == fs_distorted1 == fs_distorted2, "Sampling rates must be the same for all signals"

# Zero-pad signals to the same length
clean1, distorted1, distorted2 = zero_pad([clean1, distorted1, distorted2])

# Calculate frequency response of the system
freq_response = np.fft.fft(distorted1) / np.fft.fft(clean1)

# Apply regularization to avoid division by zero
epsilon = 1e-6
inv_freq_response = 1 / (freq_response + epsilon)

# Recover clean2 signal
recovered2_freq = np.fft.fft(distorted2) * inv_freq_response
recovered2 = np.real(np.fft.ifft(recovered2_freq))

# Normalize recovered signal
recovered2 = recovered2 / np.max(np.abs(recovered2))

wavfile.write('recovered2.wav', fs_clean1, (recovered2 * 32767).astype(np.int16))
```



4) این سوال هم من چندین بار مختلف امتحان کردم و روش های مختلفی را در نوتبوک گذاشته ام. در روش اول یک فایل صوتی نویزی را ابتدا می خوانیم و FFT را انجام می دهیم و طیف را رسم کرده و یک فیلتر bandstop را برای حذف نویز در محدوده فرکانسی خاص اعمال کرده و صدای denoise را در یک فایل جدید ذخیره می کند.

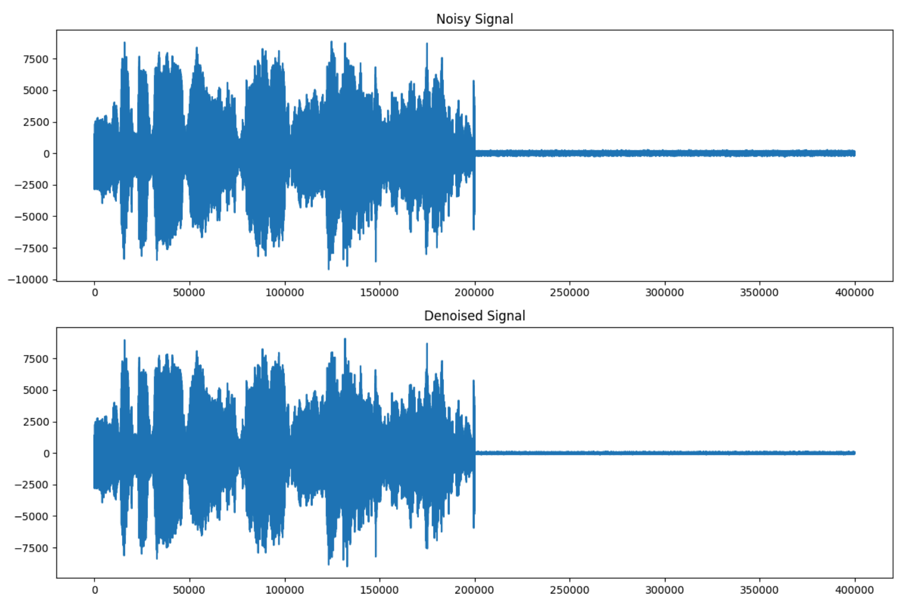
```python
sample_rate, noisy_signal = wav.read('noisy1.wav')

noisy_signal = noisy_signal.astype(float)

# Compute the FFT of the noisy signal
fft_result = fft.fft(noisy_signal)
freqs = fft.fftfreq(len(noisy_signal), 1/sample_rate)

plt.figure(figsize=(12, 6))
plt.plot(freqs[:len(freqs)//2], np.abs(fft_result)[:len(freqs)//2])
plt.xlabel('Frequency (Hz)')
plt.ylabel('Magnitude')
plt.title('Frequency Spectrum of Noisy Signal')
plt.show()

def apply_bandstop_filter(signal, lowcut, highcut, fs, order=5):
    nyquist = 0.5 * fs
    low = lowcut / nyquist
    high = highcut / nyquist
    b, a = butter(order, [low, high], btype='bandstop')
    return filtfilt(b, a, signal)

# Apply the bandstop filter
noise_freq_low = 2000  # Example frequency
noise_freq_high = 3000  # Example frequency

denoised_signal = apply_bandstop_filter(noisy_signal, noise_freq_low, noise_freq_high, sample_rate)

wav.write('denoise1.wav', sample_rate, denoised_signal.astype(np.int16))
```

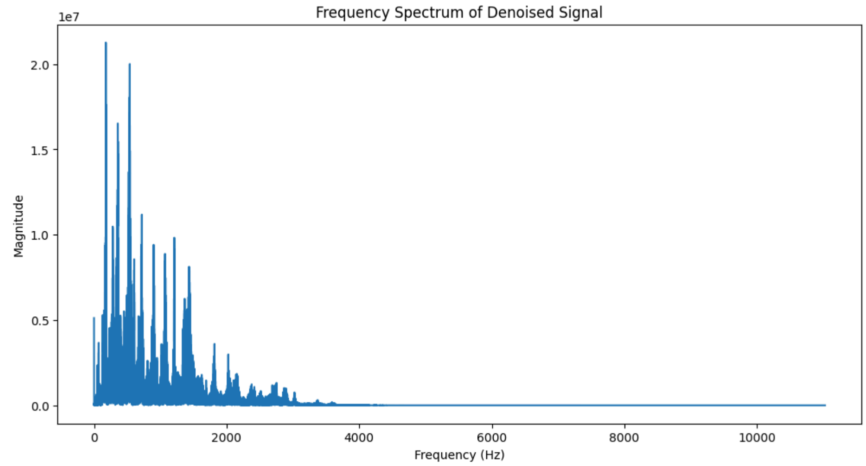




در اینجا نویز قابل توجهی در محدوده فرکانس بالاتر، به ویژه در حدود 4000-3000 هرتز وجود دارد و فیلتر بهتر lowpass
filter است برای اینکار پس دوباره کد را به این صورت تغییر می دهیم

```python
sample_rate, noisy_signal = wav.read('noisy1.wav')

# Convert to float for processing
noisy_signal = noisy_signal.astype(float)

# Compute the FFT of the noisy signal
fft_result = fft.fft(noisy_signal)
freqs = fft.fftfreq(len(noisy_signal), 1/sample_rate)

plt.figure(figsize=(12, 6))
plt.plot(freqs[:len(freqs)//2], np.abs(fft_result)[:len(freqs)//2])
plt.xlabel('Frequency (Hz)')
plt.ylabel('Magnitude')
plt.title('Frequency Spectrum of Noisy Signal')
plt.show()

def apply_lowpass_filter(signal, cutoff, fs, order=6):
    nyquist = 0.5 * fs
    normal_cutoff = cutoff / nyquist
    b, a = butter(order, normal_cutoff, btype='low', analog=False)
    return filtfilt(b, a, signal)

# Apply the lowpass filter
cutoff_freq = 3000  # Cutoff frequency at 3000 Hz

denoised_signal = apply_lowpass_filter(noisy_signal, cutoff_freq, sample_rate)

# Save the denoised signal
wav.write('denoise2.wav', sample_rate, denoised_signal.astype(np.int16))
```



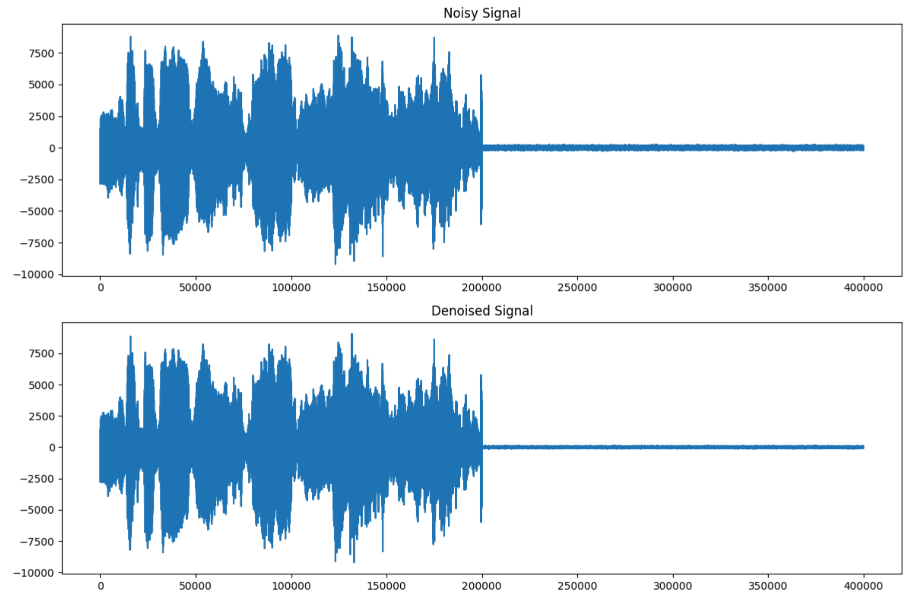


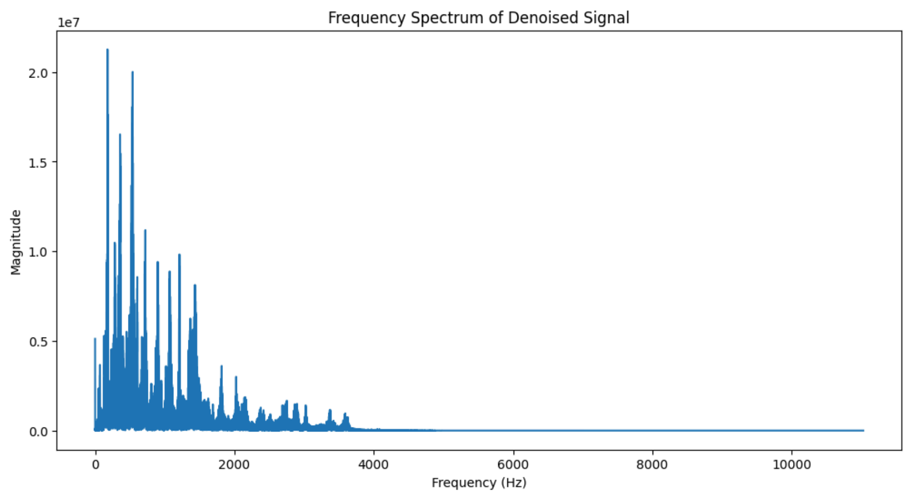
که الآن خیلی بهتر شده است و برای ورژن نهایی به این صورت کار می کنیم:

```
sample_rate, noisy_signal = wav.read('noisy1.wav')

noisy_signal = noisy_signal.astype(float)

# Compute the FFT of the noisy signal
fft_result = fft.fft(noisy_signal)
freqs = fft.fftfreq(len(noisy_signal), 1/sample_rate)

# Plot the frequency spectrum
plt.figure(figsize=(12, 6))
plt.plot(freqs[:len(freqs)//2], np.abs(fft_result)[:len(freqs)//2])
plt.xlabel('Frequency (Hz)')
plt.ylabel('Magnitude')
plt.title('Frequency Spectrum of Noisy Signal')
plt.show()

def apply_lowpass_filter(signal, cutoff, fs, order=6):
    nyquist = 0.5 * fs
    normal_cutoff = cutoff / nyquist
    b, a = butter(order, normal_cutoff, btype='low', analog=False)
    return filtfilt(b, a, signal)

# Apply the lowpass filter
cutoff_freq = 3500  # Cutoff frequency at 3500 Hz

denoised_signal = apply_lowpass_filter(noisy_signal, cutoff_freq, sample_rate)

wav.write('denoise2.wav', sample_rate, denoised_signal.astype(np.int16))

# Plot the original and denoised signals
plt.figure(figsize=(12, 8))
plt.subplot(2, 1, 1)
plt.plot(noisy_signal)
plt.title('Noisy Signal')
plt.subplot(2, 1, 2)
plt.plot(denoised_signal)
plt.title('Denoised Signal')
plt.tight_layout()
plt.show()
```

اصلی ترین تابع در اینجا apply_lowpass_filter است که با فیلتر Butterworth طراحی شده از تابع butter از SciPy استفاده می کند. این فیلتر به گونه‌ای طراحی شده است که فرکانس‌های بالاتر از یک cutoff مشخص (3500 هرتز) را attenuate کند و به طور موثر نویز فرکانس بالا را حذف کند. تابع filtfilt این فیلتر را روی سیگنال اعمال می کند و باعث zero phase distortion می شود. سپس سیگنال فیلتر شده به عنوان یک فایل WAV جدید ذخیره می شود. اطلاعات بیشتر در مورد این فیلتر در زیر است:

https://en.wikipedia.org/wiki/Butterworth_filter


Noisy Signal
7500
5000
2500
0
−2500
−5000
−7500
−10000
0
50000
100000
150000
200000
250000
300000
350000
400000
Denoised Signal
7500
5000
2500
0
−2500
−5000
−7500
−10000
0
50000
100000
150000
200000
250000
300000
350000
400000


Frequency Spectrum of Denoised Signal
1e7
2.0
1.5
1.0
0.5
0.0
Magnitude
0
2000
4000
6000
8000
10000
Frequency (Hz)