Linear Algebra & Eigensystems Homework 4

MinWook Kang

February 22, 2023

1 Fourier Transform

1.1 Introduction

선형 대수학은 모든 수학 분야의 중심이다.

- 선, 평면 및 회전과 같은 기본 물체를 정의하는 것을 포함하여 현대 기하학의 기본이다.
- 수학적 분석의 한 분야인 기능 분석은 함수 공간에 선형대수학을 적용하는 것을 볼 수 있다.

우리가 이번 섹션에 다뤄볼 것은 바로 선형 대수 방정식 세트에서 모르는 미지수를 푸는 것이다.

2 Differentiation of Functions

2.1 Problem Recognition

$x \in [0, 2\pi)$내의 25개의 포인트를 정하여, FFT를 통해 함수를 미분하는 문제이다. 함수는 다음과 같다.

$$u(x) = \max\left\{0, 1 - \frac{|x - \pi|}{2}\right\} \tag{1}$$

이때, 우리가 FFT를 이용하여 구한 미분값과 exact solution을 비교하고 정확성을 확인하는 문제이다.

2.2 Development of a solution

우선 FFT의 Spectral Differentiation을 이용하려면, 우리가 구할려고 하는 함수가 해당 범위에서 주기성을 이루는지를 확인해야 한다. 각각의 함수에 대해 범위를 지정해주고 그것에 대한 그래프를 먼저 그려보고, 그 그래프가 주기성을 가지면, 아래의 절차를 통해 함수의 미분값을 구할 수 있다.

우선, 주기 함수 $u(x)$에 대해, u_j는 N번 샘플링 된 이산화 격자 점, $x_j = .0, 1, ..., N-1$의 함수값이고, 역 이산화 푸리에 변환에 따르면,

$$u_j = \frac{\Delta k}{2\pi} \sum_{l=0}^{N-1} \hat{u}_l e^{ix_j k_l} \tag{2}$$

그리고 v_j를 $u'(x_j)$로 두면, 위의 식으로부터 우리는 아래를 얻는다.

$$u_j = \frac{\Delta k}{2\pi} \sum_{l=0}^{N-1} \hat{u}_l e^{ix_j k_l} \tag{3}$$

따라서 1차 미분 값은 $\hat{v}_l = ik_l\hat{u}_l$이고 이는 $\hat{v}_l = ik_l\hat{u}_l$로, 푸리에 계수 의 역 DFT이다. 마찬가지

방식으로 2차 미분 값을 구하게 되면,

$$w_j = \frac{\Delta k}{2\pi} \sum_{l=0}^{N-1} \underbrace{-k_l^2 \hat{u}_l}_{\hat{w}_l} e^{ix_j k_l} = \frac{\Delta k}{2\pi} \sum_{l=0}^{N-1} \hat{w}_l e^{ix_j k_l} \tag{4}$$

이고, 2차 미분 값 $w_j = u''(x_j)$ 그리고 이는 $\hat{w}_l = -k_l^2 \hat{u}_l$로, 푸리에 계수의 역 DFT이다. 우리는 v_j와 w_j를 푸리에 계수의 역 DFT로 구했고, 이것을 통해 함수의 1차, 2차 미분값을 찾을 것이다. 여기서 홀수 N에 대해서 k_l은 $l = 0, 1, ...M, M+1, ..., N-2, N-1$ 일때 $k = 0, \Delta k, ..., M\Delta k, -M\Delta k, ..., -2\Delta k, -\Delta k$로 추정한다. 여기서 M은 우리가 구현할 파이썬에서 int(N/2)에 해당한다.

N = 25로 두고 주어진 범위, $x \in [0, 2\pi)$에서 L과 xj를 정의 한다. 그리고 첫번째 함수는 max 함수에 따라 $1 - \frac{|x-\pi|}{2}$가 0 보다 커지기 전까지 0을 나타내는 함수이다. 1번 함수를 그대로 u_j에 정의하여, 1차와 2차 미분 방정식을 풀어주면 다음과 같다.

```
N = 25
xlim = [0, 2 * np.pi]
L  = xlim[1] - xlim[0]
xj = np.linspace(*xlim, N + 1)[:-1]

#definition of function 1
uj = [max(0, 1 - (abs(x - np.pi)/ 2)) for x in xj]
Uj = fft.fft(uj)
kj = np.hstack([
    np.arange(0,   N/2),      # k > 0 domain
    np.arange(-int(N/2), 0), # k < 0 domain
]) * 2*np.pi/L

# 1st-order derivatives
Vj = 1j * kj * Uj
vj = fft.ifft(Vj)

# 2nd-order derivatives
Wj = -1 * kj**2 * Uj
wj = fft.ifft(Wj)

# exact solution and vj.real solution plot
plt.figure(figsize=[11, 4])

plt.subplot(1, 2, 1)
```

```
plt.plot(xj, vj.real, "--r")
plt.legend(["$u'(x_j)$", "$v_j$"])
plt.xlabel("$x$")
plt.title("$u'(x)$")
plt.grid()

plt.subplot(1, 2, 2)
plt.plot(xj, wj.real, "--r")
plt.legend(["$u''(x_j)$", "$w_j$"])
plt.xlabel("$x$")
plt.title("$u''(x)$")
plt.grid()

plt.tight_layout()
plt.savefig("Differntiation_1.pdf")
```

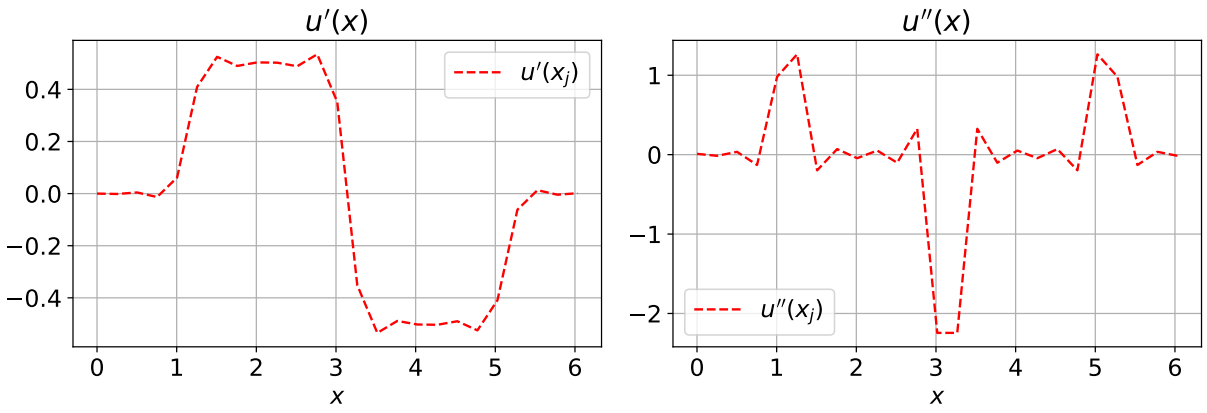

Figure 1: Derivatives of $\max\left\{0, 1 - \frac{|x-\pi|}{2}\right\}$

그리고 2번 함수같은 경우는, $sin(x)$가 지수함수 안에 들어간 형태로 마찬가지로 u_j에 대입하게 되면,

```
N = 25
xlim = [0, 2 * np.pi]
L  = xlim[1] - xlim[0]
xj = np.linspace(*xlim, N + 1)[:-1]

#definition of function 2
uj = [np.exp(np.sin(x)) for x in xj]
Uj = fft.fft(uj)
kj = np.hstack([
    np.arange(0,   N/2),     # k > 0 domain
    np.arange(-int(N/2), 0), # k < 0 domain
]) * 2*np.pi/L

# 1st-order derivatives
Vj = 1j * kj * Uj
vj = fft.ifft(Vj)

# 2nd-order derivatives
Wj = -1 * kj**2 * Uj
wj = fft.ifft(Wj)

# exact solution and vj.real solution plot
plt.figure(figsize=[11, 4])

plt.subplot(1, 2, 1)
plt.plot(xj, np.cos(xj) * np.exp(np.sin(xj)), "-k",
         xj, vj.real, "--r")

plt.legend(["$u'(x_j)$", "$v_j$"])
plt.xlabel("$x$")
plt.title("$u'(x)$")
plt.grid()

plt.subplot(1, 2, 2)
plt.plot(xj, ((np.cos(xj))**2 - np.sin(xj)) * np.exp(np.sin(xj)), "-k",
         xj, wj.real, "--r")
plt.legend(["$u''(x_j)$", "$w_j$"])
```

```
plt.xlabel("$x$")
plt.title("$u''(x)$")
plt.grid()
plt.tight_layout()
plt.savefig("Differntiation_2.pdf")


# Compare the exact solutions and vj, uj
print("-"*70)
print("1st-order derivatives accuracy")
print("-"*70)
error1 = np.cos(xj) * np.exp(np.sin(xj)) - vj.real
for i, err in enumerate(error1):
    print(f"if x = {xj[i]:.8f}, u - wj: {err}")
print("-"*70)
print("2st-order derivatives accuracy")
print("-"*70)
error2 = ((np.cos(xj))**2 - np.sin(xj)) * np.exp(np.sin(xj)) - wj.real
for i, err in enumerate(error2):
    print(f"if x = {xj[i]:.8f}, u - wj: {err}")

#Output
----------------------------------------------------------------------
1st-order derivatives accuracy
----------------------------------------------------------------------
if x = 0.00000000, u - wj: 9.999778782798785e-13
if x = 0.25132741, u - wj: -1.0031975250512914e-12
if x = 0.50265482, u - wj: 9.9298347322474e-13
if x = 0.75398224, u - wj: -9.636735853746359e-13
if x = 1.00530965, u - wj: 9.128253708468037e-13
if x = 1.25663706, u - wj: -8.423262087831063e-13
if x = 1.50796447, u - wj: 7.576994587310537e-13
if x = 1.75929189, u - wj: -6.540323838066797e-13
if x = 2.01061930, u - wj: 5.362377208939506e-13
if x = 2.26194671, u - wj: -4.1611158962950867e-13
if x = 2.51327412, u - wj: 2.877698079828406e-13
if x = 2.76460154, u - wj: -1.5476508963274682e-13
if x = 3.01592895, u - wj: 2.731148640577885e-14
if x = 3.26725636, u - wj: 9.725553695716371e-14
```

```
if x = 3.51858377, u - wj: -2.1471713296250527e-13
if x = 3.76991118, u - wj: 3.26183524634871e-13
if x = 4.02123860, u - wj: -4.330424907550423e-13
if x = 4.27256601, u - wj: 5.312417172831374e-13
if x = 4.52389342, u - wj: -6.192407697724889e-13
if x = 4.77522083, u - wj: 7.001239865633835e-13
if x = 5.02654825, u - wj: -7.737976925881185e-13
if x = 5.27787566, u - wj: 8.387457395286901e-13
if x = 5.52920307, u - wj: -8.93729534823251e-13
if x = 5.78053048, u - wj: 9.401368572525826e-13
if x = 6.03185789, u - wj: -9.777734177873754e-13
--------------------------------------------------------------------------------
2st-order derivatives accuracy
--------------------------------------------------------------------------------
if x = 0.00000000, u - wj: 9.059419880941277e-14
if x = 0.25132741, u - wj: 3.11972669919669e-14
if x = 0.50265482, u - wj: -1.7502665983215593e-13
if x = 0.75398224, u - wj: 3.106404022901188e-13
if x = 1.00530965, u - wj: -4.531930386519889e-13
if x = 1.25663706, u - wj: 6.09734485124136e-13
if x = 1.50796447, u - wj: -7.407408020299044e-13
if x = 1.75929189, u - wj: 8.477663016037695e-13
if x = 2.01061930, u - wj: -9.51017042893909e-13
if x = 2.26194671, u - wj: 1.0146328222049306e-12
if x = 2.51327412, u - wj: -1.0266093530830744e-12
if x = 2.76460154, u - wj: 1.0239586956117819e-12
if x = 3.01592895, u - wj: -1.0091927293842673e-12
if x = 3.26725636, u - wj: 9.742207041085749e-13
if x = 3.51858377, u - wj: -9.218181773462675e-13
if x = 3.76991118, u - wj: 8.505418591653324e-13
if x = 4.02123860, u - wj: -7.844835892001356e-13
if x = 4.27256601, u - wj: 7.344680419407723e-13
if x = 4.52389342, u - wj: -6.808442698513772e-13
if x = 4.77522083, u - wj: 6.167288901792745e-13
if x = 5.02654825, u - wj: -5.47950573803746e-13
if x = 5.27787566, u - wj: 4.745648318760232e-13
if x = 5.52920307, u - wj: -3.872457909892546e-13
if x = 5.78053048, u - wj: 2.836619827917275e-13
if x = 6.03185789, u - wj: -1.829647544582258e-13
```

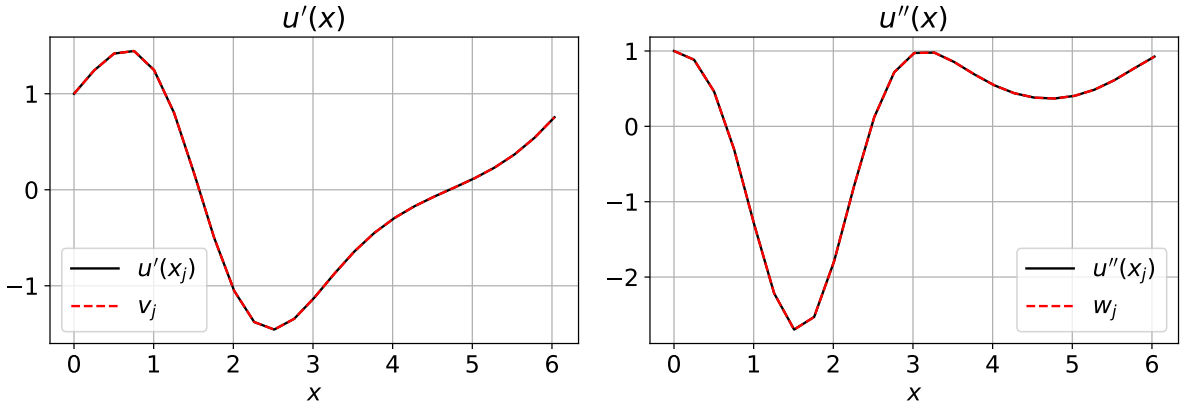

Figure 2: Derivatives of $e^{sin(x)}$

2.3 Execution and Assessment

실제로 미분한 Exact Solution과 FFT를 이용하여 구한 미분 값의 오차가 매우 작은 것을 볼 수 있다. 이것으로 우리는, 미분이 쉽게 불가능한 함수도, FFT와 Spectrum Differentiaion을 통해 미분값을 찾을 수 있다는 것을 보았다.

3 Differentiation of Functions

3.1 Problem Recognition

3.2 Development of a solution

3.3 Execution and Assessment

4 Differentiation of Functions

4.1 Problem Recognition

4.2 Development of a solution

4.3 Execution and Assessment

5 Do-Re-Mi-Fa-So-La-Ti-Do by FFT Module

5.1 Problem Recognition

note와 frequency, amplitude 가 주어졌을때, 첫음을 C_4를 기준으로 $Do - Re - Mi - Fa - So - La - Ti - Do$를 연주해야 한다. 앞서 배운 코드를 잠시 확인하면, 먼저 note와 frequency 그리고 amplitude 리스트를 정의한 후, 주파수와 진폭에 대한 그래프를 확인하면 다음과 같다. 그리고 우리는,

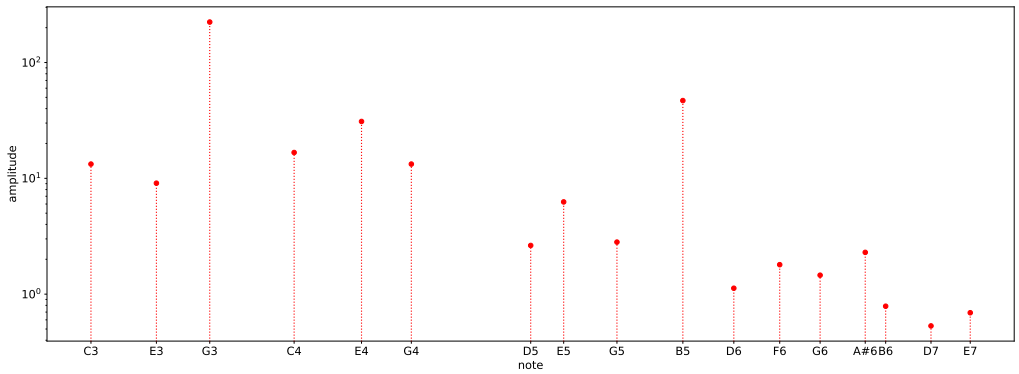

Figure 3: Example Amplitude - C Major

아래의 식을 통하여, 코드의 2초 길이의 모노 오디오 샘플을 생성할 수 있다.

$$u(x) = \sum_{i=1}^{N \text{ notes}} A_i \cos(2\pi f_i t + \phi_i) \tag{5}$$

이때, A_i 그리고 f_i는 i번째 노트의 진폭과 주파수를 나타낸다. 또한 Phase Factor을 무작위의 $0 \leq \phi_i < 2\pi$로 둠으로써, 코드를 연주할 수 있게 된다. 이를 이용하여, 우리가 원하는 음역대인 C_4에서 시작해 각각의 음에 따라 연속적이게 들리게 해야 한다.

5.2 Development of a solution

문제를 풀기 위해서, 우선 고려해야 할 것은 다음과 같다.

- 오디오 샘플에서 원하는 도, 레 와 같은 각각의 사운드를 나타내야 한다.
- 각각의 사운드를 만들 때에는, Filter을 이용하여 amplitude를 지우고, 그것을 위해서는 FFT를 이용하여야 한다.
- 나타낸 각각의 사운드를 합쳐 연속적인 사운드를 만들어야 한다.

선행된 예제에서는 Audio 함수를 사용했기 때문에, 마찬가지로 Audio 함수를 사용하여 오디오 샘플을 만들어 줄 것이다. 주어진 note에 따른 frequency와 amplitude에 대해, B_3부터 $C_{\sharp 5}$까지의 음역대를 골랐다. 이를 각각의 리스트에 대해서 note, frequency 그리고 amplitude에 각각 넣어준뒤, note에 대한 amplitude의 그래프는 다음과 같다.

```
note = ["B3", "C4", "C_shop4", "D4", "D_shop4", "E4", "F4", "F_shop4", "G4",
        "G_shop4", "A4", "A_shop4", "B4", "C5", "C_shop5",

]
frequency = [ # in Hz
    246.94, 261.63, 277.18, 293.66, 311.13, 329.63, 349.23, 369.99, 392.00,
    415.30, 440.00, 466.16, 493.88, 523.25, 554.37
]
amplitude = [
    139.71, 131.87, 124.47, 117.48, 110.89, 104.66, 98.79, 93.24, 88.01,
    83.07, 78.41, 74.01, 69.85, 65.93,62.23


]

plt.figure(figsize=[30, 10])
ax = plt.subplot()
plt.loglog(frequency, amplitude, "or")
for x, y in zip(frequency, amplitude):
    plt.loglog([x, x], [0, y], ":r")
plt.xlabel("note")
```

```
plt.ylabel("amplitude")
plt.xticks(frequency, note)
ax.tick_params(axis='x', which='minor', bottom=False)
plt.savefig("B3_to_Cshop5_Figure1")
```

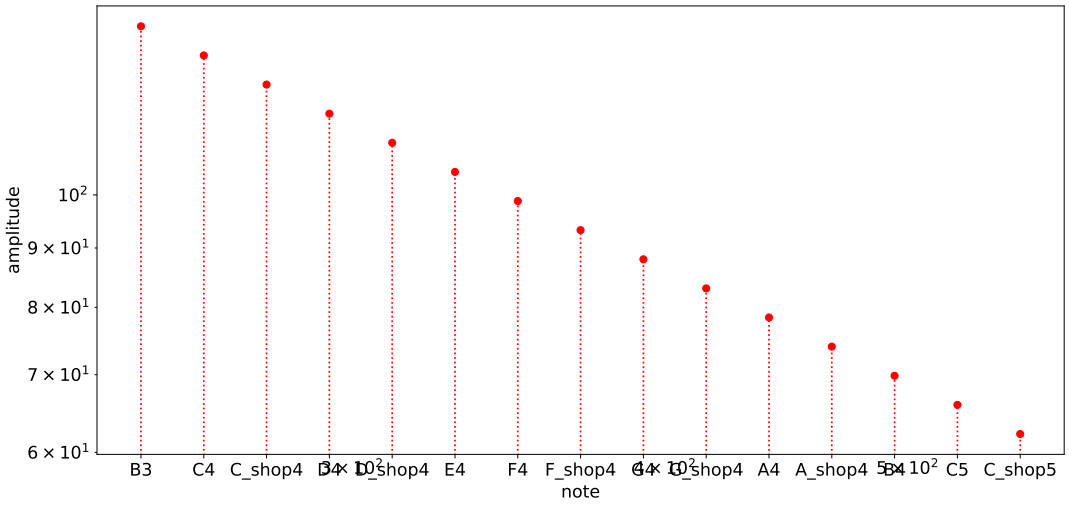

Figure 4: B_3 to $C_{\sharp 5}$ note and amplitude

그리고 이 파형을 소리를 내려면, 우선 위 공식에 맞게 식을 세워주고, 무작위 위상에 대한 샘플을 만들 것이므로, random 함수를 사용하여 표현한다.

```
# number of samples per second,
sampling_frequency = 48000 // 8

# then, maximum frequency that can be resolved
max_freq = 0.5 * sampling_frequency

# discrete time domain
max_time = 1 # in seconds
time = np.linspace(0, max_time, sampling_frequency * max_time + 1)[:-1]

# random phases for different notes
phase = np.random.rand(len(time)) * 2*np.pi

# wave samples
```

```
wave  = sum(A * np.cos(2*np.pi*f*time + ph)
              for A, f, ph in zip(amplitude, frequency, phase))
```

이렇게까지 하면 우리는 위에서 정한, B_3부터 $C_{\sharp 5}$까지의 음역대의 모든 wave가 재생이 될 것이다. 하지만 우리는 순차적으로 연속적인 도레미파 솔라시도의 소리를 원하기 때문에, 나머지 파장대의 소리를 제거할 필요가 있다.

우선, 도 라는 소리를 내기 위해서는 도 이외에 모든 파장대의 소리를 제거해야 한다. 그러기 위해선, 먼저 FFT를 통해 푸리에 함수로 만들어 준다.

```
Wj = fft.fft(wave)
N  = len(Wj)

Dk = 2*np.pi / (max(time) - min(time))
kj = np.hstack([
    np.arange(0,    N/2),      # k > 0 domain
    np.arange(-int(N/2), 0), # k < 0 domain
]) * Dk

# kj is 'angular' frequency.
# divide it by 2pi to convert to frequency in Hz
fj = kj / (2*np.pi)

# power spectral density
Pj = np.abs(Wj/N)**2 / Dk

plt.figure(figsize=[30, 10])
ax = plt.subplot()
plt.loglog(fj[:int(N/2)], Pj[:int(N/2)], "-r", lw=1)
plt.xlim([frequency[0]*0.8, frequency[-1]*1.2])
plt.xlabel("note")
plt.ylabel("$P(k)$")
plt.xticks(frequency, note)
ax.tick_params(axis='x', which='minor', bottom=False)
plt.grid(axis="x")
plt.savefig("FFT_figure.pdf")
```

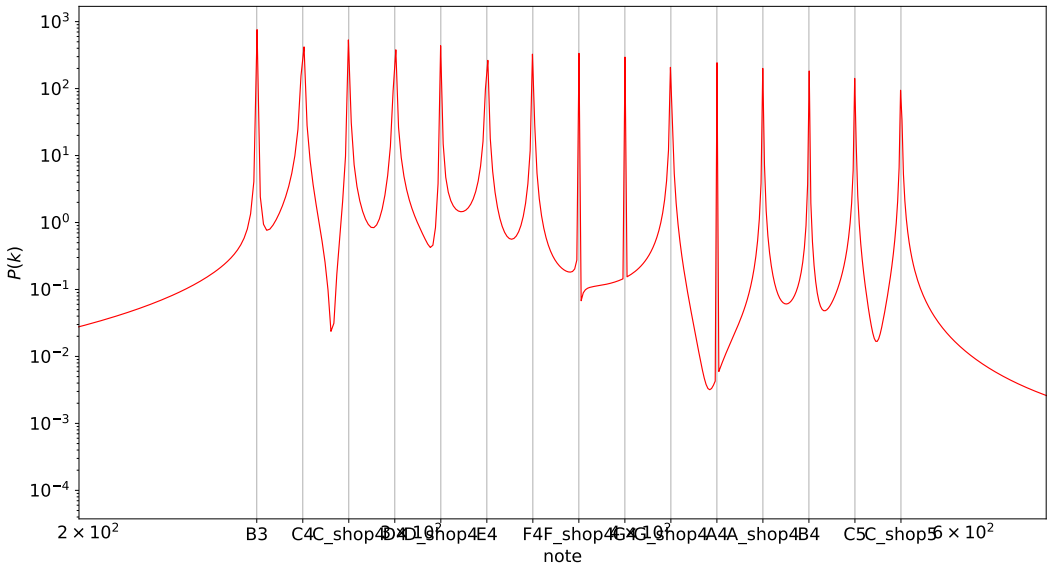

Figure 5: B_3 to $C_{\sharp 5}$ note and amplitude

우리는 "도"의 음역대를 원하기 때문에, Filter을 이용하여 도를 제외한 wave amplitude는 제거하고, 역변환 시켜야 한다. 여기까지 코드를 작성하면 다음과 같다.

```
Major = ["C4", "D4", "E4","F4", "G4", "A4", "B4", "C5",]

def Filter(index):
        freq_cut1 = 0.5 * (frequency[index] + frequency[index + 1])
        freq_cut2 = 0.5 * (frequency[index - 1] + frequency[index])
        indices1 = np.abs(fj) > freq_cut1
        indices2 = np.abs(fj) < freq_cut2
        filtered_Wj = Wj.copy()
        filtered_Wj[indices1,] *= 0
        filtered_Wj[indices2,] *= 0
        return fft.ifft(filtered_Wj).real
```

위 Filter 함수는, 우리가 원하는 음에 해당하는

```
note = [
    "C3", "E3", "G3", "C4", "E4", "G4", "D5", "E5", "G5",
    "B5", "D6", "F6", "G6", "A#6", "B6", "D7", "E7",
]
```

note 리스트의 해당 index를 입력시키게 되면, 그 음을 제외한 모든 wave amplitude를 제거하는 함수이다. 이렇게 까지 하여, 도라는 음역대를 재생할 수 있게 되었다.

나머지 소리를 연속해서 내기 위해서는, 반드시 wave 하나에 우리가 담고자 하는 $Do-Re-Mi-Fa-So-La-Ti-Do$ wave 파형이 다 들어있어야 하는데, 따라서 이 함수를 각각의 레, 미, 파, ... , 도(C_5)까지 반복시킬 필요가 있다. 하지만, 문제는 Audio 파일에는 하나의 wave만 담을 수 있기 때문에, 여러개의 소리를 덧붙일려면 일반적인 합의 방식을 사용하면 안된다. 우선 우리가 구한 이 wave 변수는 1차원 array로 이루어져 있는데 여기서 array를 그저 더하게 되면 wave amplitude는 모든 음역대의 소리가 합쳐지게 된다. 이를 좀더 구체화 시키면,

```
filtered_wave1 = Filter(1)
#x = np.linspace(0, time, len(filtered_wave1))
plt.figure(figsize = (9, 9))
plt.plot(time, filtered_wave1, '.r', label = '$C_4$ wave')
plt.legend()
plt.grid()
plt.xlabel("time")
plt.ylabel("wave amplitude value")
plt.savefig("wave_amp_C4.pdf")
```

우리가 구한 도에 대한 wave amplitude를 $t = 0$에서 1까지를 보여준 그래프이다. 이때, 위상이 여러개로 보이는 이유는 우리가 0 에서 2π까지 무작위로 만들어진 위상에 대한 그래프이기 때문에 여러개의 amplitude wave를 볼 수 있게 된다. 여기서 만약, 합의 방식으로 함수를 더하게 되버리면, 소리를 들어보면, 도와 레가 합쳐진 소리가 들리게 되는데, 실제로 그래프를 그려보면,

```
filtered_wave1 = Filter(1)
filtered_wave3 = Filter(3)
#x = np.linspace(0, time, len(filtered_wave1))
plt.figure(figsize = (9, 9))
plt.plot(time, filtered_wave1, '.r', label = '$C_4$ wave')
plt.plot(time, filtered_wave3, '.b', label = '$D_4$ wave')
plt.legend()
plt.grid()
plt.xlabel("time")
plt.ylabel("wave amplitude value")
```

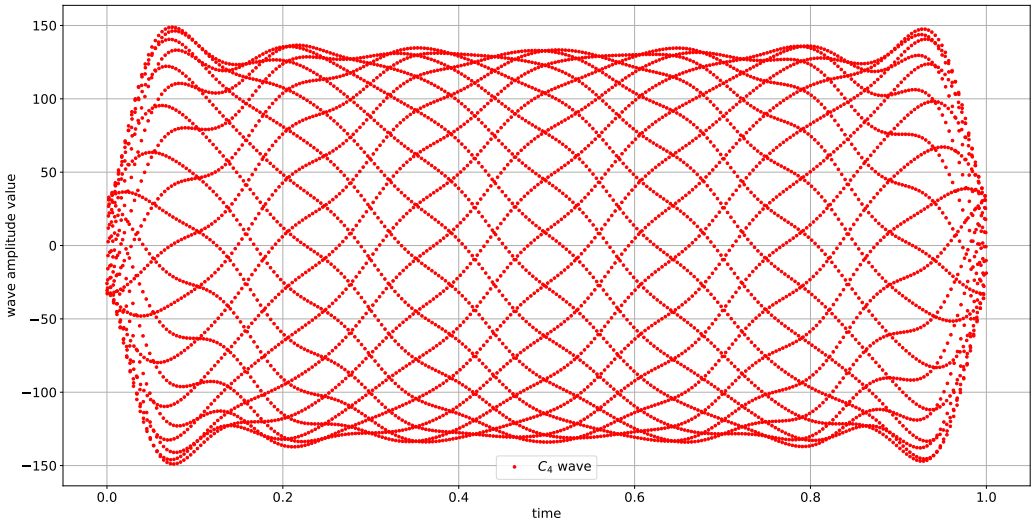

Figure 6: C_4 amplitude

```
11 plt.savefig("wave_amp_C4+D4.pdf")
```

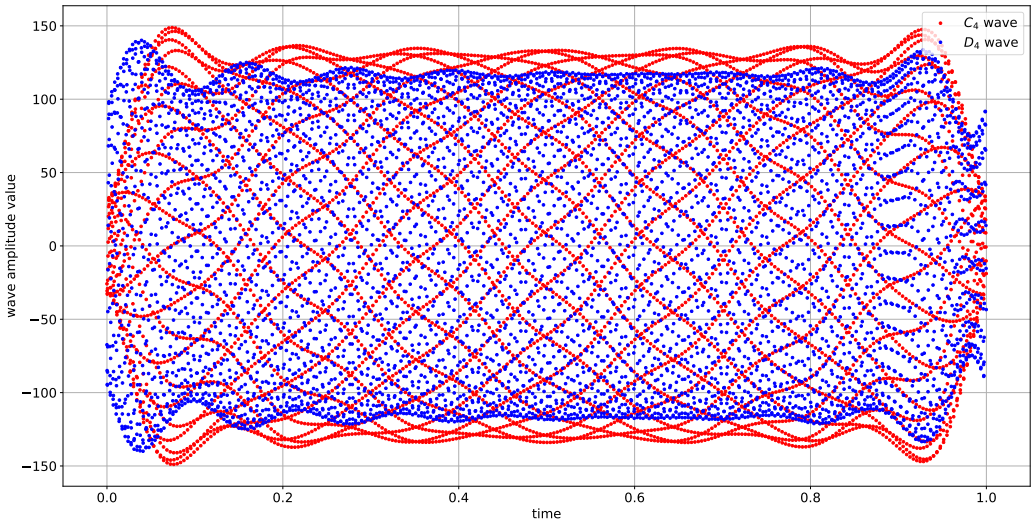

Figure 7: $C_4 + D_4$ amplitude

두 파장대의 amplitude wave 가 합쳐져 각각의 소리를 연속적으로 내지 않고, 한번에 재생되게 되면서 새로운 wave가 형성되게 된다. 따라서 소리를 듣게 되면, 도와 레가 합쳐진 소리를 들을 수 있게

된다. 각각의 파장을 연속적으로 재생시키려면 겹치지 않게 wave를 각각 이어주어야 한다. 우리가 앞서 봤듯, wave는 1D array로 되어있기 때문에, 이를 이어주기 위해서는 np.concatenate를 이용하고, 이때 이어주는 방향은 axis = 0 으로 x축 방향으로 파장을 시간에 따라 붙여주면 된다. 따라서 우리는 도와 레의 amplitude wave를 붙여준 것을 그래프로 확인하게 되면,

```
filtered_wave1 = Filter(1)
filtered_wave3 = Filter(3)

filtered_wave_sum = np.concatenate((filtered_wave1, filtered_wave3), axis = 0)

# time
x_sum = np.linspace(0, 2 * max_time, 2 * (sampling_frequency * max_time + 1) - 1)
    [:-1]
x_C4 = time
x_D4 = np.linspace(1, 2, (sampling_frequency * max_time))

# plot C4 + D4
plt.figure(figsize = (20, 10))
plt.plot(x_sum, filtered_wave_sum, ':k', label = '$C_4$ + $D_4$ wave', alpha = 0.5)
plt.plot(x_C4, filtered_wave1, '.r', label = '$C_4$ wave', alpha = 0.25)
plt.plot(x_D4, filtered_wave3, '.b',  label = '$D_4$ wave',
         alpha = 0.25)
plt.legend()
plt.grid()
plt.xlabel("time")
plt.ylabel("wave amplitude value")
plt.savefig("wave_amp_independence.pdf")
Audio(filtered_wave_sum, rate=sampling_frequency)
```

우리가 예상했던대로, np.concatenate를 이용하면 빨간색과 파란색 두 파장이 겹치지 않게 시간 $t = 1$일때, D_4에 해당하는 wave amplitude가 합쳐지면서, 하나의 검은 새로운 wave amplitude가 만들어 진 것을 볼 수 있으며, 시간에 대해 각각 독립적이게 연속적인 형상을 띄는 것을 볼 수 있다. 또한, Audio 함수를 통하여 직접 듣게 되면, 우리가 원하는 도, 레 의 독립적이고 연속적인 사운드를 들을 수 있다. 우리는 이것을 통하여 실제 Audio wave에도 적용한다.

```
# array box
filtered_wave_sum = np.zeros(1)
```

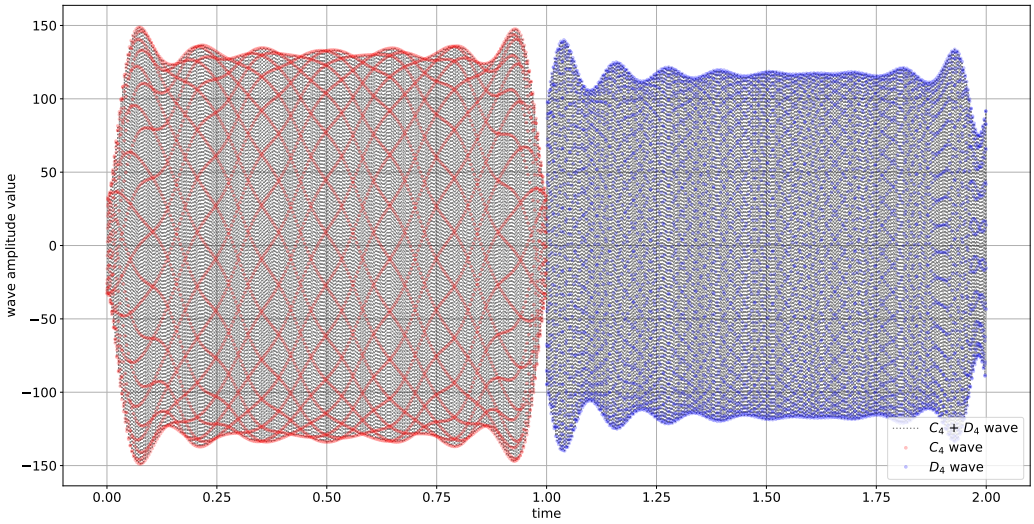

Figure 8: $C_4 + D_4$ independece amplitude

```
# filtered code put in box
for code in Major:
    index = [i for i, n in enumerate(note) if n == code][0]

    # Filtering
    filtered_wave = Filter(index)
    filtered_wave_sum = np.concatenate((filtered_wave_sum, filtered_wave), axis = 0)

Audio(filtered_wave_sum, rate=sampling_frequency)
```

fitered wave sum 변수를 먼저 빈 1차원의 array로 만들어 준 뒤, 우리가 Filter을 통해 만든 각각의 wave amplitude를 np.concatenate를 통하여 이어주고, 최종적으로 만들어진 fitered wave sum를 재생하면, 하나의 오디오에서 연속적인 음을 나타낼 수 있게 된다.

5.3 Execution and Assessment

우리는 원하는 사운드를 얻기위해 FFT로 변환시켜 푸리에 함수로 구한 뒤, Filter을 적용함으로써 역변환 시켜 원하는 사운드를 얻는 문제를 풀어보았다. 또한 우리가 원하고자 하는 wave amplitude는 파형을 더하거나 독립적이게 array에서 합하게 해줌으로써, 만약 원한다면 화음이나, 독립적인 사운드를 내는 방식도 가능한 것을 확인하였다.