Week 11

Discrete Fourier Transform

theory

- Discrete Fourier Transform

$$H_n \equiv \sum_{k=0}^{N-1} h_k \mathrm{e}^{-\frac{2\pi i k n}{N}}$$

$$\equiv \sum_{k=0}^{N-1} h_k [\cos(2\pi kn/N) - i \sin 2\pi kn/N]$$

$$where, \qquad n = 0, 1, 2, \ldots, N-1$$

$$H \equiv [h_0 \quad \cdots \quad h_{N-1}] \begin{bmatrix} 0 & \cdots & 0 \\ \exp(-2\pi i \cdot 1 \cdot 0/N) & & \exp(-2\pi i \cdot 1 \cdot (N-1)/N) \\ \vdots & \ddots & \vdots \\ \exp(-2\pi i \cdot (N-2) \cdot 0/N) & & \exp(-2\pi i \cdot (N-2) \cdot (N-1)/N) \\ \exp(-2\pi i \cdot (N-1) \cdot 0/N) & \cdots & \exp(-2\pi i \cdot (N-1) \cdot (N-1)/N) \end{bmatrix}$$

- Inverse Fourier Transform

$$h \equiv \frac{1}{N} * [H_0 \quad \cdots \quad H_{N-1}] \begin{bmatrix} 0 & \cdots & 0 \\ \exp(2\pi i \cdot 1 \cdot 0/N) & & \exp(2\pi i \cdot 1 \cdot (N-1)/N) \\ \vdots & \ddots & \vdots \\ \exp(2\pi i \cdot (N-2) \cdot 0/N) & & \exp(2\pi i \cdot (N-2) \cdot (N-1)/N) \\ \exp(2\pi i \cdot (N-1) \cdot 0/N) & \cdots & \exp(2\pi i \cdot (N-1) \cdot (N-1)/N) \end{bmatrix}$$

Code preview – hw13

```python
# define function for FT
# let 'centre' to 'True' to set 0 as center
# data >> 1d array with frequency data from real space
# return 1d-array, frequency + real and imaginary part separately
def fourier_tf(data, centre = False):
    # n >> size of data
    n = len(data)

    # gap >> index data of given n-elements array
    # cpt >> repeated calculation for FT
    gap = np.arange(n)
    tmp = gap.reshape((n, 1))
    cal_arr = tmp * gap
    cpt = 2 * np.pi * cal_arr / n

    # calculate FT
    # loop has been used since memory is not enough to store n-n array with data
    R_data = data @ np.cos(cpt)
    I_data = -data @ np.sin(cpt)

    # centre >> If True, shift to set 0 as center. default is False
    div_n = n//2
    result = np.zeros((3, n))

    result[1] = R_data
    result[2] = I_data

    R_data = np.roll(R_data, div_n)
    I_data = np.roll(I_data, div_n)

    gap = (gap - div_n) / n

    if centre:
        result[0] = gap
        result[1] = R_data
        result[2] = I_data
    else:
        result[0] = np.roll(gap, div_n)

    # return value is 2d array with frequency and FT results
    # return Real and Imaginary part seperately
    return result[0], result[1], result[2]
```

이산 푸리에 변환을 위한 함수 선언.
Frequency와 real, imaginary 값을 따로 반환함.

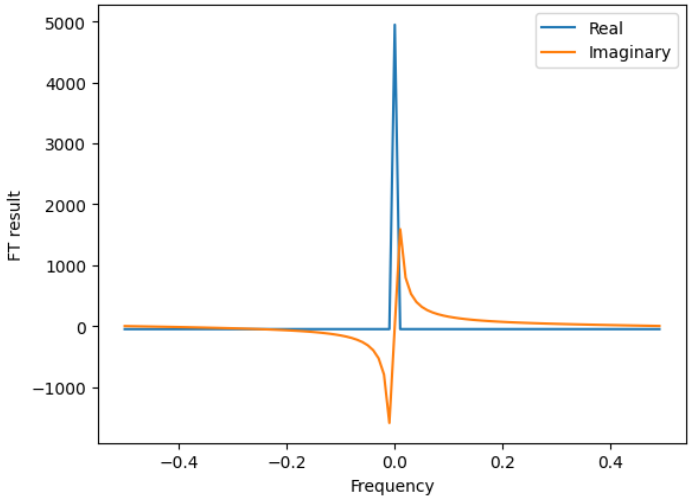
```python
# do inverse FT with complex data and size of array n
def inv_tf(data ,n):
    # making array for calculation
    row = np.arange(n)
    col = row.reshape((n, 1))
    arr = row * col
    arr = arr * 2 * np.pi / n
    arr = np.cos(arr) + 1j * np.sin(arr)

    result = np.real(arr @ data) / n
    return result
```

역 푸리에 변환을 위한 함수 선언
원본 데이터 복구를 위한 작업이므로
결과 amp만 길이 n의 배열로 반환.

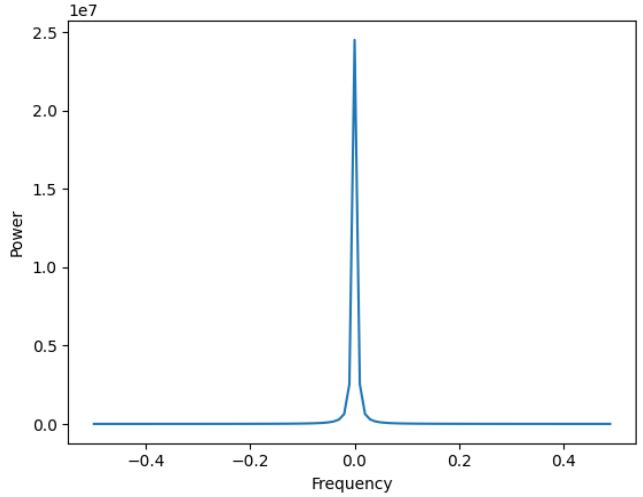
```python
RS_low = np.copy(RS_nc)
IS_low = np.copy(IS_nc)

n_list = ["99", "75", "50" ,"25", "10"]

low_result = np.zeros((5, n))
err_low = np.zeros(5)

for k in range(5):
    RS_low[int(n_list[k]):] = 0
    IS_low[int(n_list[k]):] = 0
    low_result[k] = inv_tf(RS_low + 1j * IS_low, n)
    err_low[k] = np.mean((h_k - low_result[k])**2) ** (0.5)
```

일부 푸리에 계수가 손실되었을 때의
역 푸리에 변환 결과를 보기 위한 코드
남겨진 데이터에 따라 얼마나 변하는지
관찰하기 위해서 n_list에 따라 계산

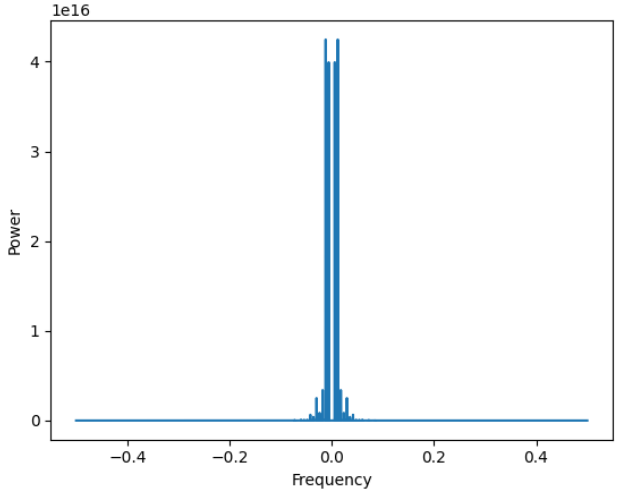
result – hw13



FT result



Power spectrum of FT result



Power spectrum(“Do262.wav”)

좌측 상단은 주어진 0~99까지의 정수로 진행한
FT 결과이다.
계산 결과를 통해 Power Spectrum을 계산한 결과가
우측 상단의 그래프이다.

Power Spectrum은 신호가 주파수 별로 가진 에너지를
나타낸다.
이때 0Hz 에서의 FT 값은 real space에서의 신호의 적분값과
같다. 따라서 이는 신호의 평균 크기나 신호의 편향 정도를
나타낸다.

우리가 사용한 데이터는 단순하게 증가하는 신호이므로
FT결과 데이터가 0Hz 주변의 낮은 주파수에 모여있는
모습을 보인다.
이를 hw14의 “도”음과 비교해보면,
0Hz에서의 값이 상대적으로 없음을 확인할 수 있다.
이는 “도”음은 진동하며, 반복되는 신호이기 때문에 특정한
상수 성분이나 편향이 우리의 신호보다 적기 때문이다.

따라서, 신호가 주어진 것과 같이 선형적인 모습을 보이면
0Hz에서의 값이 큰 부분을 차지한다.

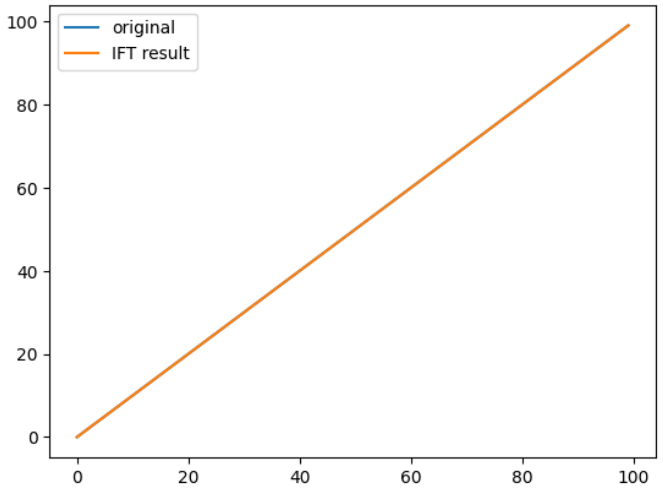
```
cmp_PS = PS[1][np.where(PS[0] == 0)]/np.sum(PS[1]) * 100
print(str(cmp_PS) + "%")
```

✓ 0.0s

```
[74.62311558]%
```

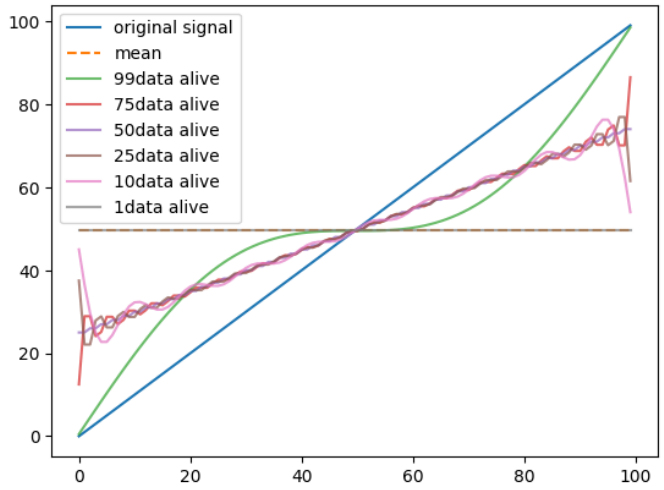
상단과 같이 0Hz에서의 파워가 74.6%로 dominant함을
관찰할 수 있다.

result – hw13



IFT result with all data

```
MSE is 6.396444083330331e-25
```



IFT result with less data

```
RMSE (signal) with 99   : 11.255805366367504
RMSE (signal) with 75   : 14.298857222026992
RMSE (signal) with 50   : 14.439529078193644
RMSE (signal) with 25   : 14.853553326805645
RMSE (signal) with 10   : 15.718661652652102
RMSE (signal) with 1    : 28.86607004772212
-----------------------------------------------
RMSE (mean) with 99     : 21.28780253251505
RMSE (mean) with 75     : 14.828285350378838
RMSE (mean) with 50     : 14.430869689661819
RMSE (mean) with 25     : 14.290112590736076
RMSE (mean) with 10     : 13.97824113696527
RMSE (mean) with 1      : 0.0
```

두 그래프 모두 직전에 계산한 FT 결과를 가지고 IFT를 수행한 결과이다.

모든 푸리에 계수를 사용하여 IFT를 수행한 결과,
복원된 신호는 원본 신호와 거의 유사함을 보였다.
MSE를 통해 오차를 분석한 결과 6e-25로 사실상 원본 신호와 같다는 것을 알 수 있었다.

반면 사용한 푸리에 계수가 적어질수록 다른 모습을 보였다.
0Hz는 신호의 평균 세기와 같으므로 낮은 주파수 영역만 남길수록,
사용된 푸리에 계수가 적어질수록 원본 신호의 평균값에 가까운 신호로 복원되었다.

이를 RMSE를 통해 확인해보면,
더 많은 데이터가 사용될수록 원본 신호에 가까워지며,
더 적은 낮은 주파수의 신호가 사용될수록 평균값에 가까워지는 것을 확인할 수 있었다.

극단적으로 1개의 데이터만 남긴 경우(0Hz), 복원된 신호가 기존 신호의 평균과 완전히 동일한 모습을 확인할 수 있었다.
이를 통해 0Hz에서의 값은 전체 신호의 평균임을 확인할 수 있었다.

Code preview – hw14

```python
# define function for FT
# let 'centre' to 'True' to set 0 as center
# data >> 1d array with frequency data from real space
# return 1d-array, frequency + real and imaginary part separately
def fourier_tf(data, centre = False):
    # n >> size of data
    n = len(data)

    # gap >> index data of given n-elements array
    # cpt >> repeated calculation for FT
    gap = np.arange(n)
    tmp = gap.reshape((n, 1))
    cal_arr = tmp * gap
    cpt = 2 * np.pi * cal_arr / n

    # calculate FT
    # loop has been used since memory is not enough to store n-n array with data
    R_data = data @ np.cos(cpt)
    I_data = -data @ np.sin(cpt)

    # centre >> If True, shift to set 0 as center. default is False
    div_n = n//2
    result = np.zeros((3, n))

    result[1] = R_data
    result[2] = I_data

    R_data = np.roll(R_data, div_n)
    I_data = np.roll(I_data, div_n)

    gap = (gap - div_n) / n

    if centre:
        result[0] = gap
        result[1] = R_data
        result[2] = I_data
    else:
        result[0] = np.roll(gap, div_n)

    # return value is 2d array with frequency and FT results
    # return Real and Imaginary part seperately
    return result[0], result[1], result[2]
```

이산 푸리에 변환을 위한 함수 선언.
Frequency와 real, imaginary 값을 따로 반환함.

```python
def read_wav(filename):
    with wave.open(filename, 'rb') as do:
        frame_rate = do.getframerate()
        n_frame = do.getnframes()
        length = n_frame / frame_rate
        data = do.readframes(n_frame)
        data = np.frombuffer(data, dtype=np.int16)

    time = np.linspace(0, length, len(data))
    do.close()
    return time, data, frame_rate
```

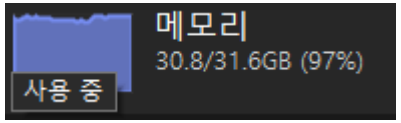
Python에서 기본으로 제공하는 wave 라이브러리를 통해 Wav 파일을 읽어옴.

함수를 선언하여 이번 과제에서 사용할 데이터를 반환함.
⇒신호의 길이, 샘플 당 진폭, 샘플 간격

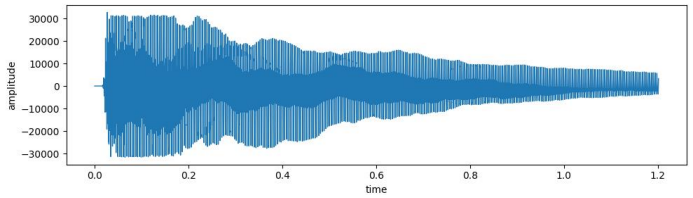
다루는 데이터의 크기가 크기 때문에 .flatten 대신 frombuffer 함수를 사용하여 메모리를 절약함.

```python
# For general use, restricted 'for' loop is used. ex) 52432, 2d array is too large for matrix computation.
for i in range(n):
    inside = cpt * i
    R_data[i] = data @ (np.cos(inside)).T
    I_data[i] = -data @ (np.sin(inside)).T
```

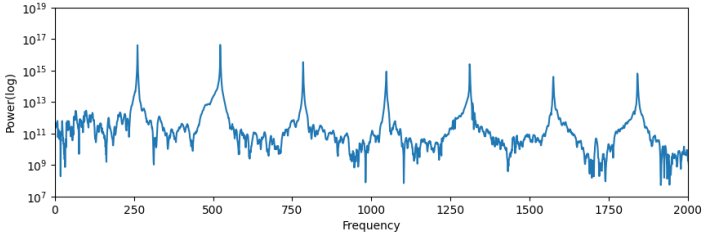
신호 데이터가 너무 큰 경우, 행렬 연산을 위해
행렬을 선언하는 과정에서 메모리가 부족해지므로
For loop를 사용하여 연산 당 메모리 사용량을 줄임.

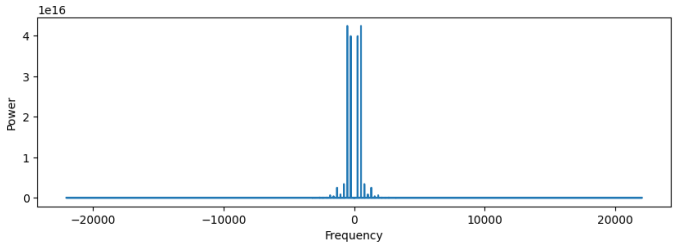


result – hw14



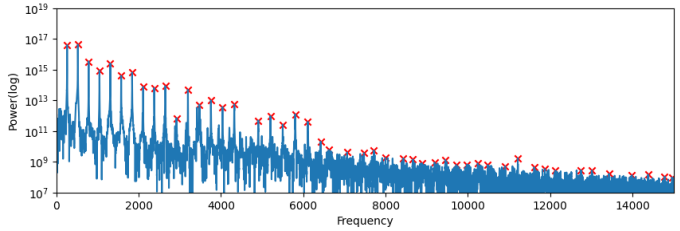
"Do262"의 파형

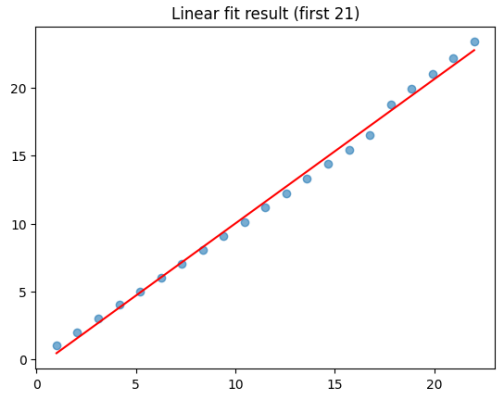




파워 스펙트럼

```
n = 200
local_max_index = []
for i in range(n, len(power_hw13) - n):
    if (power_hw13[i] > np.max(power_hw13[i-n:i])) and (power_hw13[i] > np.max(power_hw13[i+1:i+n+1])):
        if (freq_hw13[i] > 0) and (power_hw13[i] > 1e7):
            local_max_index.append(i)

local_max = [freq_hw13[i] * p_rate for i in local_max_index]
print(local_max)
```






```
기울기    : 1.0631112896808983
MSE       : 0.1993118650750367
```

좌측 상단의 그래프는 주어진 Do262.wav 파일의 파형이다.
이를 푸리에 변환한 결과가 좌측 하단의 두 그래프이다.

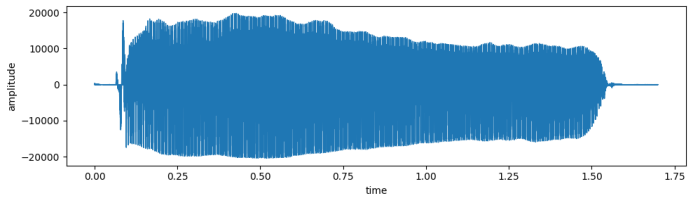
데이터의 분포가 0Hz에서 벗어나 분포하는 것에서 주어진 신호는 평균값을 기준으로 분산되어 있는 형태임을 알 수 있다.
즉, 편향이 일어나지 않은 상태로 말할 수 있다.

Power spectrum에서 가장 dominant한 신호는 ~522.62Hz에서 확인되었다.

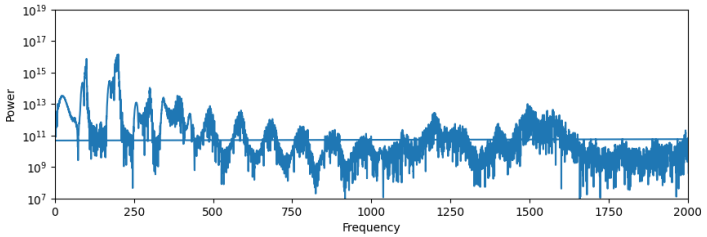
더하여 log scale 그래프에서 여러 개의 peak를 확인할 수 있었다.
어림잡아 각 peak가 200Hz 정도 떨어져있기 때문에 상단의 코드를 활용하여 특정 지점에서 양 옆으로 200만큼 떨어진 범위내에서 최댓값을 가지는 지점을 찾아보면, 각 peak를 잘 찾아내는 모습을 볼 수 있다.
이때 첫 peak는 ~262Hz에서 나타났다.

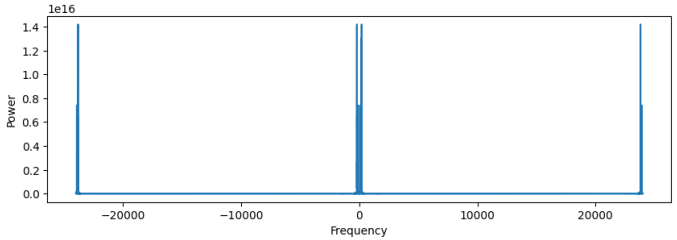
대략 ~6000Hz 정도까지의 데이터를 기준으로 peak가 반복하는 주기를 일차함수로 fitting해보면,
MSE ~0.2정도의 기울기 1인 일차함수이므로 262Hz만큼의 상수배를 이루는 것을 확인하였다.

result – hw14



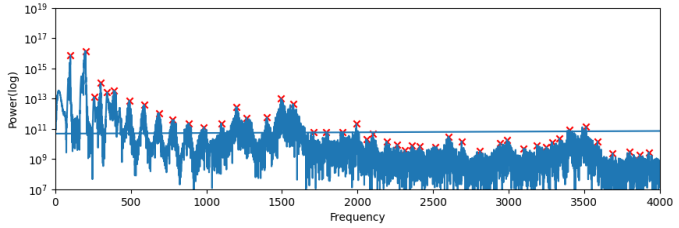
"human"의 파형

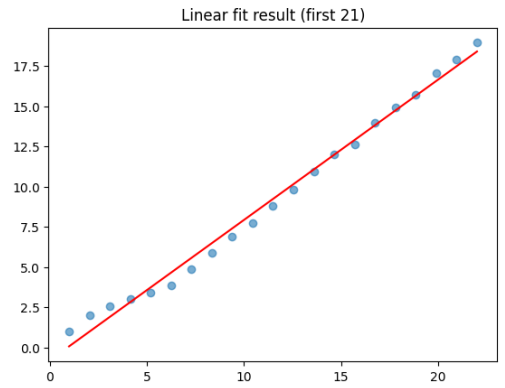




파워 스펙트럼

```
k = 100
local_max_index_h1 = []
for i in range(k, len(h1_power) - k):
    if (h1_power[i] > np.max(h1_power[i-k:i])) and (h1_power[i] > np.max(h1_power[i+1:i+k+1])):
        if (rate_h1[i] > 0) and (h1_power[i] > 1e7):
            local_max_index_h1.append(i)

local_max_h1 = [rate_h1[i] * h1_rate for i in local_max_index_h1]
print(local_max_h1)
```






```
기울기    : 0.8730188956414342
MSE       : 0.27093901476032484
```

좌측 상단의 그래프는 직접 녹음한 음성의 파형이다.
이를 푸리에 변환한 결과가 좌측 하단의 두 그래프이다.

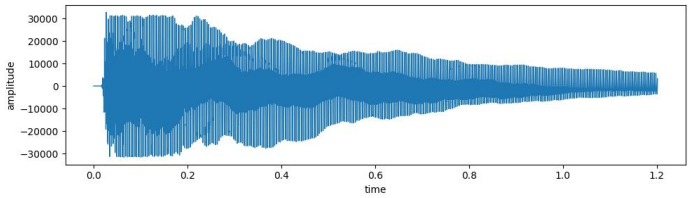
기존 '도' 음과는 달리 0Hz에서 유의미한 신호가 보이는 것으로 보아 신호에 잡음이 상대적으로 많음을 알 수 있다.

Power spectrum에서 가장 dominant한 신호는 ~23799Hz에서 확인되었다. 이는 중심에서 매우 먼 주파수이며, 파워 스펙트럼 그래프의 양 끝 점이였다.

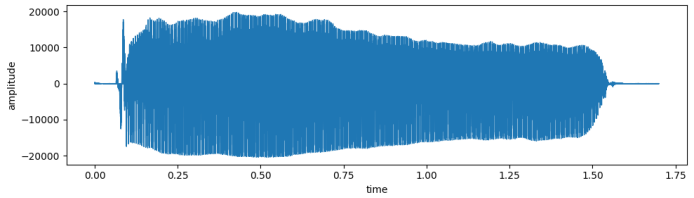
더하여 log scale 그래프에서 '도' 음과 같이 여러 개의 peak를 확인할 수 있었다.
직접 녹음한 경우, peak가 100Hz 정도 떨어져있기 때문에 상단의 코드를 활용하여 특정 지점에서 양 옆으로 100만큼 떨어진 범위내에서 최댓값을 가지는 지점을 찾아보았다.
이때 첫 peak는 ~100Hz에서 나타났다.

기존 데이터와의 비교를 위해서 첫 21개의 peak만 사용하여 일차함수로 예측해보았다.
MSE ~0.3정도의 값으로 예측되는 것에서 신호가 일관된 모습을 보이지 않음을 확인하였다.
기울기의 경우, 피아노가 1.06 정도임에 반해 0.87로 피아노보다는 상수배를 잘 따르지 못하는 모습을 보였다.

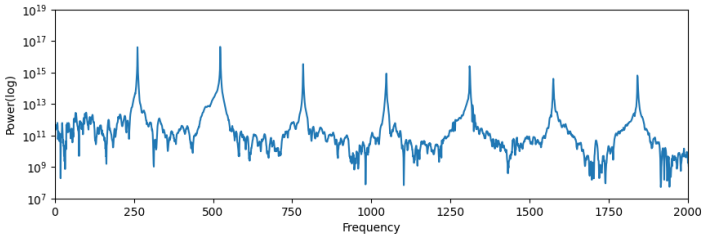
result – hw14

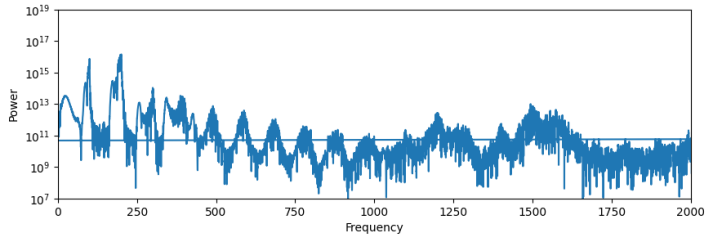


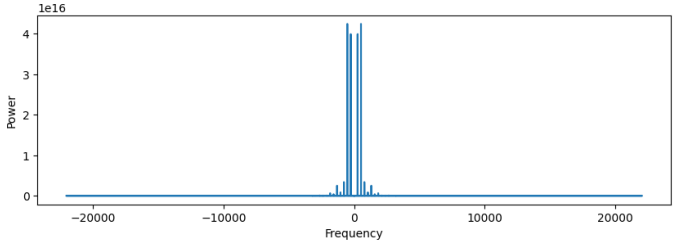
"Do262"의 파형

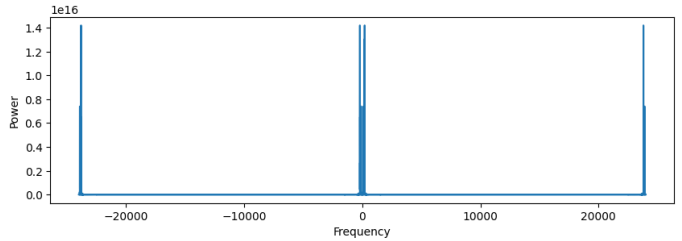


"human"의 파형









두 소리의 파형을 비교해보면 전혀 닮지 않음을 확인할 수 있었다.

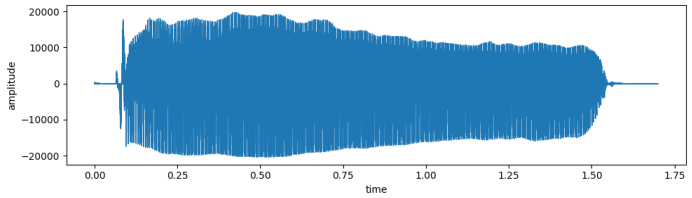
이를 파워 스펙트럼을 통해서 확인해보면 log scale 그래프의 peak 사이의 간격이 피아노의 경우 ~261Hz 정도로 균일한 간격을 보였으나 녹음본의 경우 ~100 또는 ~50Hz 등의 불규칙한 간격을 보였다.
이를 통해 피아노 신호에 포함된 주파수들보다 녹음본에 포함된 주파수들 사이의 상관관계가 더 적음을 확인하였다.

더하여 피아노의 경우 각 peak의 위치가 명확하게 나타났으나 녹음본의 경우에는 peak가 비교적 빠르게 흐릿해지는 경향을 보였다.

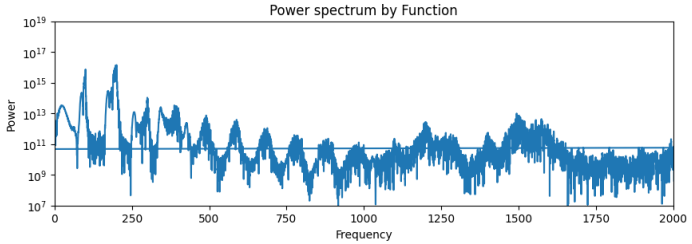
파워 스펙트럼의 0Hz를 기준으로 확인해보면, 피아노의 경우 0Hz 주변으로 각 주파수의 신호가 분포해있으나 녹음본의 경우 0Hz 성분이 유의미하게 존재하며 신호가 나타난 주파수가 0 또는 신호의 끝자락으로 분포해있는 모습을 확인할 수 있었다.

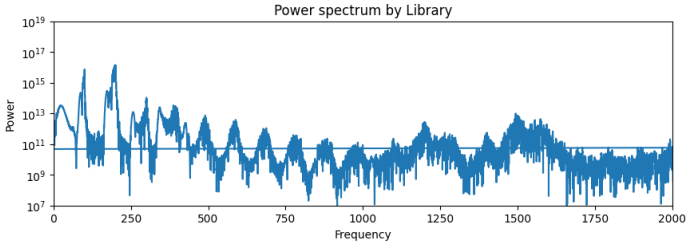
결과적으로 직접 녹음한 데이터에는 피아노 데이터보다 잡음이 많았으며 이를 포함하더라도 처참한 모습을 보였다.

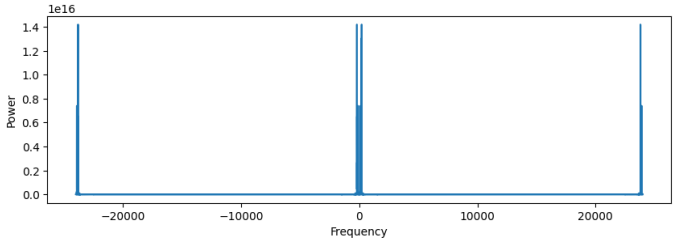
result – hw14

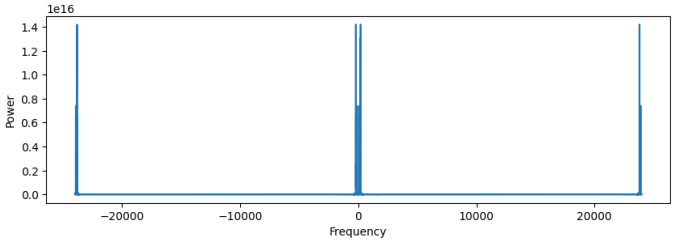


"human"의 파형





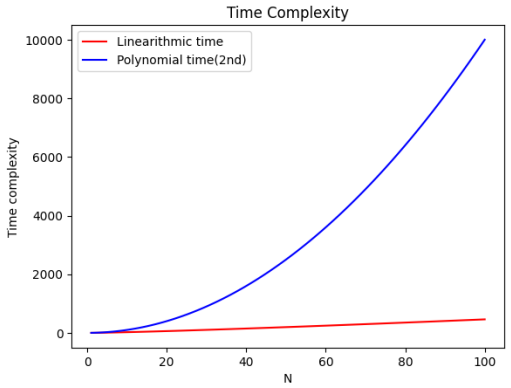





```
# fft 모듈 vs 구현한 함수
diff = h1_tf - lib_result
ft_mse = np.mean(np.abs(diff)**2)
print('MSE:', ft_mse)
```

✓ 0.0s

```
MSE: 4.0242490142605575e-09
```




```
rate_h1, Rh1, Ih1 = fourier_tf(h1_data)
```

✓ 6m 0.0s

```
lib_result = npf.fft(h1_data)
```

✓ 0.0s

라이브러리를 통한 FT와 직접 구현한 FT를 비교해보면, 육안으로는 큰 차이가 없었다.

대체로 계산 속도에서 큰 차이를 확인할 수 있었다. FT의 경우 ~6min 이였고 FFT의 경우 ~0.6ms 정도로 큰 차이를 보였다.

이는 시간 복잡도의 차이에 의해 발생한다.
FT : O(N)
FFT : O(NlogN)
이므로 FT에 활용한 데이터의 배열의 크기를 N으로 두면, N = 163200 일때 시간 복잡도의 차이는 26×10^9정도로 연산 시간이 매우 크게 차이나는 것을 알 수 있다.