

信号処理基礎練習 テキスト

長岡技術科学大学

電気電子情報系 杉田研究室

本テキストは、信号処理の基礎を学ぶためのきっかけを提供するためのものであり、基本的な事項しか記述していない。適宜、参考書などで不明な点を調べて理解を深めること。

分からないことを分からないままにしないように！

なお、実行に当たっては、各自のパソコンに `python` の環境を構築して実行しても構わない。

0. Google Colaboratory の準備

0.1 Google Colaboratory について

Google Colaboratory（以下 Google Colab）は、Google 社が無料で提供している機械学習の教育や研究用の開発環境です。開発環境は Jupyter Notebook に似たインターフェースを持ち、Python の主要なライブラリがプリインストールされています。Google Colab はネットワーク環境とブラウザ（Chrome 推奨）、Google アカウントを持っていればすぐに利用できます。

0.2 Google Colab の基本的な使い方

0.2.1 ノートを作成する

➤ Google Colab サイトから作成

[Google Colab サイト](#) へアクセスすれば、最初のノートブックが表示されます。このノートブックはウェルカムページのようなもので、Google Colab でノートブックを操作する例が記載されています。



図 1：「Colaboratory へようこそ」ノートブック

ページの一番上にメニューが並んでいます。その中の「ファイル」－「ノートブックを新規作成」をクリックして下さい。すると、新しいタブに「Untitled0.ipynb」というノートブックが表示されます。



図 2：ノートブックを新規作成

ノートブックを初めて作成すると、マイドライブ（Google ドライブ）の直下に [Colab Notebooks] というフォルダが生成されます。新規作成したノートブックは、このフォルダに保存されています。



図 3：作ったノートブックはマイドライブに保存される

➤ Google drive から作成

マイドライブの「新規作成」→「その他」→「Google Colaboratory」を選択する。その他で Google Colaboratory が一覧にない場合は、「アプリを追加」でインストールする。

0.2.2 Colab のモジュール、バージョンの確認とインストール

Google Colab にインストールされている python モジュールとバージョンを確認するには、ノートブック上で

```
!pip list
```

としてセルを実行する。実行結果が下に示されるが、モジュール数が多いのでスクロールして確認する必要がある。

colab で pip を使うには pip の前に!を付ければ良い。

```
!pip install 'モジュール名'
```

ただし、この方法のインストールは時間がたつと初期化されるため、初期化ごとに毎回インストール作業が必要になる。そのため、マイドライブ上にモジュール用の適当なフォルダを作成し（例えば、my_modules という名前のフォルダを用意し）、そこにインストールするのが便利である。

「my_modules」へパッケージ：pyaudio をインストールする場合の例

```
!pip install --target /content/drive/MyDrive/Colab¥ Notebooks/my_modules pyaudio
```

0.3 Google drive のマウント

Google Colab 上から Google Drive 上のファイルなどにアクセスするためには、Google Drive をマウントする必要があります。

```
from google.colab import drive  
  
drive.mount('/content/drive', force_remount=True)
```

上記では、/content/drive 配下に Google Drive をマウントするように設定しています。

```
import sys  
  
ROOT_PATH = 'drive/My Drive/Colab Notebooks/ '  
  
sys.path.append(ROOT_PATH)  
  
import my_modules
```

これで ModuleNotFoundError: No module named 'my_module' のようなエラーが出なければ OK。
my_module の中のライブラリ my_utils やさらにその中のメンバーを使いたければたとえば次のように書く。

```
from my_modules import my_utils  
  
from my_toolbox.visualizer import hoge
```

ですが、今回の実習では、既に Google Colab にインストールされているモジュール以外は使うことはないでしょう

0.4 配布データの準備

「B3_python.zip」をダウンロードして解凍して、以下のファイルが含まれていることを確認してください。

- ・ 2つのフォルダ (hrtfs と room_impulse) ,
- ・ 音源ファイル (trumpet_44100.wav, nijiiro.wav)
- ・ 1つの Jupyter ノートブック (拡張子 ipynb)

これらすべてのファイルを、Google ドライブの [Colab Notebooks] の下に入れてください。

(どこに入れても良いのですが、本テキストでは、Colab Notebooks の下にそれらがあることを前提にサンプルコードを作成しています。

とりあえず google colab 使ってみる！

1. Colab Notebooks の下にある「trumpet_44100.wav」を soundfile を使って読み込んでみる。
2. 読み込んだデータをグラフ化してみる。
3. データを再生してみる
4. .wav ファイルとして保存してみる。

■ soundfile のよる音声ファイルの読み込み

```
from google.colab import drive
drive.mount('/content/drive', force_remount=True)
drive_path = '/content/drive/MyDrive/Colab Notebooks/'

import os
import soundfile as sf
file_source = "trumpet_44100.wav"
x0, sr = sf.read(os.path.join(drive_path, file_source))
print(x0.shape)

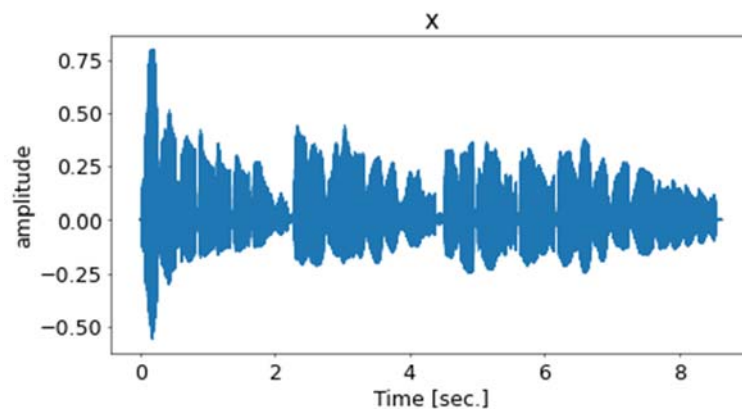
if len(x0.shape) == 2:
    x = 0.5*(x0[:,0]+x0[:,1]) #今回はモノラルで読み込みたいのでステレオのとき左右の平均
else:
    x = x0
```

■ グラフ化

```
import numpy as np
import matplotlib.pyplot as plt
plt.rcParams["font.size"] = 18

fig, ax = plt.subplots(figsize=(10, 5), dpi=50)
dummy_t = np.arange(0, len(x))/sr #横軸を"時間"で表すためのダミー変数
ax.plot(dummy_t, x) #データ x0 のプロット
ax.set_title('x') # タイトル設定
ax.set_xlabel("Time [sec.]") # x 軸のラベル設定
ax.set_ylabel("amplitude") # y 軸のラベル設定
plt.show()
```

正しく実行されれば、以下のようなグラフが作成されるはず。



■ 再生（再生するときは耳を傷めないように音量に注意）

```
import IPython.display
IPython.display.Audio(x.T, rate=sr, autoplay=False, normalize=True)
```

■ オーディオファイル（.wav）として保存

```
y = x0
write_filename = "out.wav"
_format = "WAV"
subtype = 'PCM_24'
sf.write(os.path.join(drive_path, write_filename), y, sr, format=_format,
         subtype=subtype)
```

上記の例だと、Colab Notebooks の下に「out.wav」が生成されているはずである。

1. デジタル信号処理の基礎

振幅 A ，周波数 f ，位相 ϕ の正弦波信号は次式で表せる。

$$x_a(t) = A \sin(2\pi f t + \phi) \quad (1.1)$$

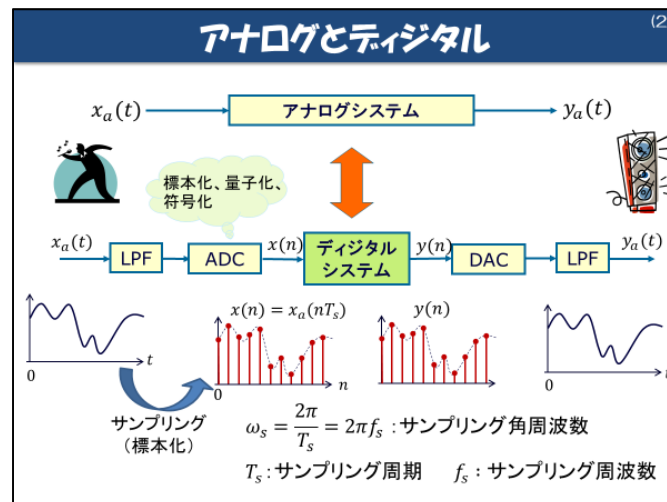
ここで t は連続時間である。

今， $x_a(t)$ をサンプリング周波数 f_s で標本化したとする（すなわち， $T_s = 1/f_s$ 間隔でデータを取り出す）と，その離散時間信号は次式で表せる。

$$x_a(nT_s) = A \sin(2\pi f nT_s + \phi) = A \sin\left(2\pi f \frac{n}{f_s} + \phi\right) \quad (1.2)$$

ただし， n は整数である。以降，離散時間信号 $x_a(nT_s)$ を $x(n)$ で表記する。

$$x(n) = A \sin\left(2\pi f \frac{n}{f_s} + \phi\right) \quad (1.3)$$



ナイキストのサンプリング定理

* 元のアナログ信号の情報を完全に保存するための条件
(スペクトル形状)

『入力信号(元のアナログ信号)に含まれる最も高い角周波数成分を ω_m とすると、2倍の角周波数 ($2\omega_m$) よりも高い角周波数 ω_s でサンプリングすれば、標本化信号(デジタル信号)から元のアナログ信号を完全に復元できる』

$$\omega_s > 2\omega_m$$

サンプリング角周波数 ω_s が決まっている場合は、入力信号を $\omega_s/2$ 以下に制限すれば、元の信号を完全に復元できることを保証する。

入力信号の上限 $\frac{\omega_s}{2}$: ナイキスト角周波数

■ $\omega_s = 2\pi f_s$, $\omega_m = 2\pi f_m$ なので、周波数で言えば、

$$f_s > 2f_m$$

入力信号の上限 $\frac{f_s}{2}$: ナイキスト周波数

デジタルの世界では，サンプリング周波数を f_s とすると， $f_s/2$ までの信号しか正しく扱えないことに注意が必要である。

<離散時間信号の生成>

振幅 $A = 0.1$, 位相 $\phi = 0$, サンプルング周波数 $f_s = 8000$ Hz, 信号の周波数 $f = 440$ Hzとして, 0~3 秒の正弦波信号を生成してみる。

```
import numpy as np
import math
import matplotlib.pyplot as plt
plt.rcParams["font.size"] = 18

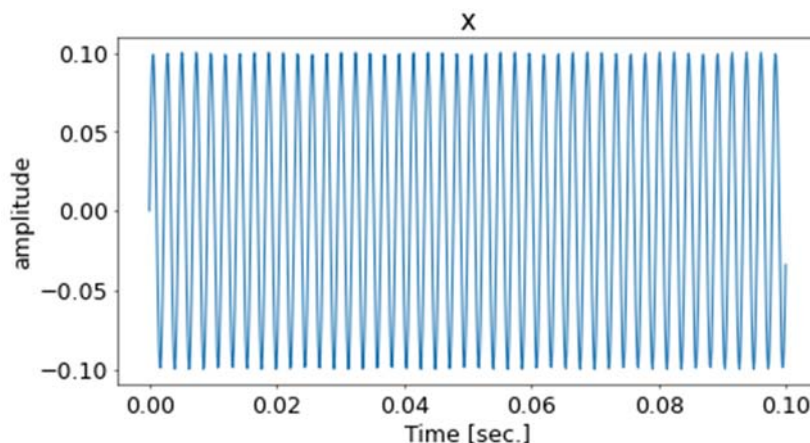
sr = 8000 # サンプルングレート  $f_s$ 
f = 440 # 正弦波の周波数
n = np.arange(0, 3*sr) #  $0 \leq n < 3*sr$  の範囲で間隔 1 のデータを生成
x = 0.1*np.sin(2.0*np.pi*f*n/sr) # 振幅 0.1, 周波数 440Hz

plt_st = 0
plt_end = math.ceil(0.1*sr)
fig, ax = plt.subplots(figsize=(10, 5), dpi=50)
ax.plot(n[plt_st:plt_end]/sr, x[plt_st:plt_end])
ax.set_title('x')
ax.set_xlabel("Time [sec.]")
ax.set_ylabel("amplitude")
plt.show()
```

正しく実行されれば, 以下のようなグラフが作成されるはずである。ここでは, 0~0.1 秒までのデータしかプロットしていないが,

`plt_end = len(x)`

などと修正すればすべてのデータ（上記の場合は 3 秒分のデータ）がプロットされる。



【信号生成の練習】

以下では、すべてサンプリング周波数 $f_s = 8000$ Hz とする。

練習 1-1

3 つの異なる周波数： $f_1 = 220$ Hz, $f_2 = 440$ Hz, $f_3 = 660$ Hz から成る 3 秒間の複合信号を生成せよ。

$$x(n) = x_1(n) + x_2(n) + x_3(n)$$

ただし、 $x_1(n) = A_1 \sin\left(2\pi f_1 \frac{n}{f_s}\right)$, $x_2(n) = A_2 \sin\left(2\pi f_2 \frac{n}{f_s}\right)$, $x_3(n) = A_3 \sin\left(2\pi f_3 \frac{n}{f_s}\right)$ とする。

また、振幅 $A_1 = A_2 = A_3 = 0.1$, とする。

練習 1-2

信号の周波数成分が $f_1 = 220$ Hz, $f_2 = 440$ Hz, $f_3 = 660$ Hz と 1 秒間毎に変化する信号を生成せよ。

それぞれ 1 秒間の $x_1(n)$, $x_2(n)$, $x_3(n)$ を生成し、

$$x = \text{np.concatenate}([x1, x2, x3], 0)$$

を利用することで結合できる。

練習 1-3

python では、

$$\text{np.random.normal}(\text{mean}, \text{std}, N)$$

を利用することで、正規分布に従って平均 mean , 標準偏差 std の N 個のランダム信号を生成できる。

これを利用して、平均 0, 平均パワー 0.5, 3 秒間のホワイトノイズを生成せよ。

また、生成した信号が、平均 0 で平均パワー 0.5 であることを確認せよ。

2. フーリエ変換

フーリエ変換はデータ解析手法のひとつであり、一般的には時間領域のデータを周波数領域へ変換するためのアルゴリズムとして利用される。信号処理の分野においては、周波数解析(スペクトル解析)やデジタルフィルタの高速化のために用いられる重要な技術である。Python でフーリエ変換するには SciPy にある `fft`, `ifft` 関数や `rfft`, `irfft` 関数が便利である。以下は、`fft`, `ifft` 関数を利用した例である。FFT の対象信号は、振幅 1, 周波数 1000Hz の正弦波信号 1 秒間であり、サンプリング周波数 8000Hz である。

```
import numpy as np
import math
import matplotlib.pyplot as plt
from scipy.fft import fft, ifft
plt.rcParams["font.size"] = 18

sr = 8000 # サンプリングレート
f = 1000 # 正弦波の周波数
n = np.arange(0, sr) # 0 <= n < sr の範囲で間隔 1 のデータを生成
x1 = 1*np.sin(2.0*np.pi*f*n/sr) # 振幅 1, 周波数 1000Hz

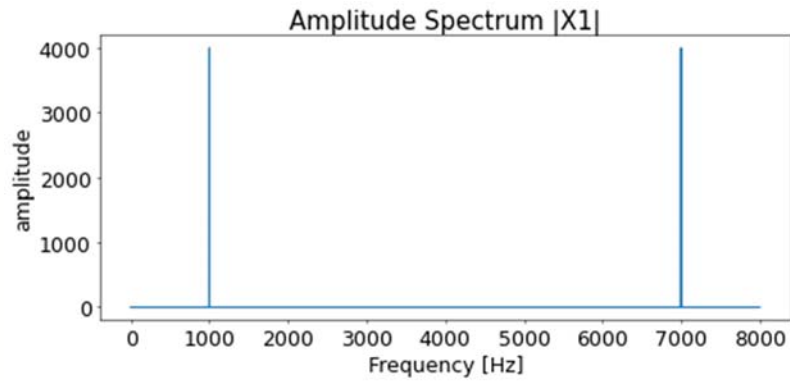
X1 = fft(x1) # 信号 x1 のスペクトル X1 (x1 の全データを使用)

fig, axes = plt.subplots(figsize=(10,5), tight_layout=True, dpi=50)
freq1 = np.arange(0, len(X1))/len(X1) * sr # 横軸を"周波数"で表すためのダミー変数
axes.plot(freq1, np.abs(X1)) # 振幅スペクトルの表示
axes.set_title('Amplitude Spectrum |X1|')
axes.set_xlabel("Frequency [Hz]")
axes.set_ylabel("amplitude")

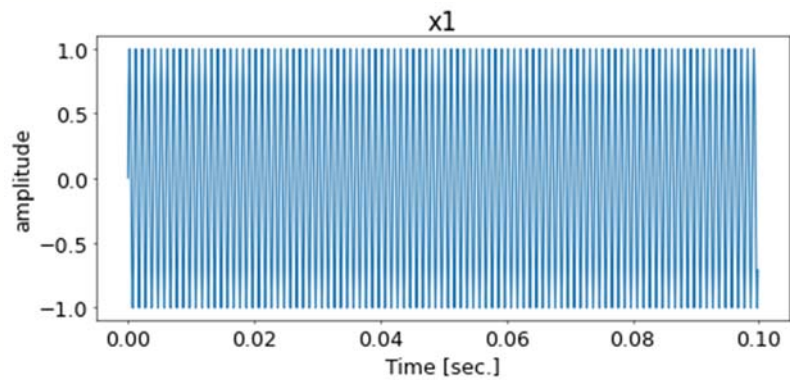
reconst_x1 = ifft(X1)

plt_st = 0
plt_end = math.ceil(0.1*sr)
fig, axes = plt.subplots(figsize=(10,5), tight_layout=True, dpi=50)
axes.plot(n[plt_st:plt_end]/sr, x1[plt_st:plt_end])
axes.set_title('x1')
axes.set_xlabel("Time [sec.]")
axes.set_ylabel("amplitude")

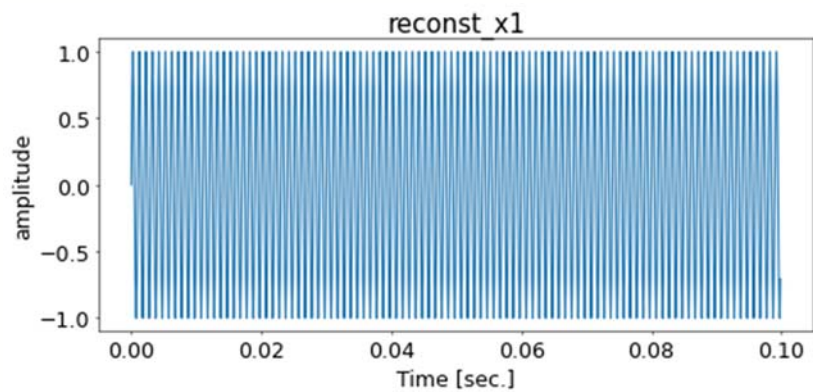
fig, axes = plt.subplots(figsize=(10,5), tight_layout=True, dpi=50)
axes.plot(n[plt_st:plt_end]/sr, np.real(reconst_x1[plt_st:plt_end]))
axes.set_title('reconst_x1')
axes.set_xlabel("Time [sec.]")
axes.set_ylabel("amplitude")
```



(a) 信号 x1 の振幅スペクトル



(b) 信号 x1 の時間波形 (0.1 秒間のみ表示)



(c) スペクトル X1 の逆 FFT

図 2-1: フーリエ変換、逆変換

フーリエ変換の定義式はいくつかあるが、その一つは以下であり、python の fft, ifft では以下の定義式が用いられている。

$$\text{離散フーリエ変換: } X(k) = \sum_{n=0}^{N-1} x(n) e^{-j\frac{2\pi nk}{N}}, \quad k = 0, 1, \dots, N-1 \quad (2.1)$$

$$\text{逆離散フーリエ変換: } x(n) = \frac{1}{N} \sum_{k=0}^{N-1} X(k) e^{j\frac{2\pi nk}{N}}, \quad n = 0, 1, \dots, N-1 \quad (2.2)$$

ここで、 $X(k) = |X(k)|e^{j\phi(k)}$ としたとき、 $|X(k)|$ を振幅スペクトル、 $\phi(k)$ を位相スペクトルと呼ぶ。また、 k は周波数軸上の離散点を意味する。詳細な説明は参考書等に譲るが、式(2.1)より、角周波数軸上 $[0, 2\pi)$ を等間隔に離散化した N 点についてスペクトルが評価されることが分かる。

* 角周波数軸上 $[0, 2\pi)$ は周波数で言えば $[0, f_s)$ である。

ところで、図 2-1(a)を見ると 1000Hz と 7000Hz の所に大きなスペクトルを持っていることが確認できる。ここで、以下の疑問を生じるのではないだろうか。

疑問 1 元々の信号は 1000Hz なのに、なぜ 7000Hz にもスペクトルが立つのか？

疑問 2 生成した 1000Hz の信号の振幅は 1 なのに、なぜスペクトルの大きさが 4000 なのか？

疑問 1 については、「①複素スペクトルでは、正の周波数と負の周波数を用いて表現される」、「②離散時間信号のスペクトルは、サンプリング周波数毎に繰り返される」、という 2 つの事項を思い出せば直ちに納得できるであろう。生成した信号の周波数は 1000Hz であるが、複素スペクトルでは $\pm 1000\text{Hz}$ で表現される（ちなみに、その時の振幅値は実スペクトルの振幅値の $1/2$ である）。また、今回はサンプリング周波数が 8000Hz なので、 $\pm 1000\text{Hz}$ のスペクトルがサンプリング周波数毎に繰り返されるためである。したがって、7000Hz は -1000Hz と同じ役割である。

疑問 2 については、離散フーリエ変換、離散逆フーリエ変換の定義に依存する。もし次式で定義されるならばどうだろうか？

$$\text{離散フーリエ変換：} X(k) = \frac{1}{N} \sum_{n=0}^{N-1} x(n) e^{-j\frac{2\pi nk}{N}}, \quad k = 0, 1, \dots, N-1$$

$$\text{逆離散フーリエ変換：} x(n) = \sum_{k=0}^{N-1} X(k) e^{j\frac{2\pi nk}{N}}, \quad n = 0, 1, \dots, N-1$$

式(2.1)-(2.2)との違いは、 $1/N$ をフーリエ変換の方に付けるか、逆フーリエ変換の方に付けるかだけである。先の例ではサンプリング周波数 8000Hz で 1 秒間のデータを FFT しているので、データ数 N は 8000 である。したがって、もし上記の定義式を使用したとすれば、1000Hz と 7000Hz のスペクトルの大きさは、 $4000/8000 = 0.5$ と算出されるはずで、確かに、実スペクトルの振幅値の $1/2$ となる。

何らかのツールを使って FFT 処理する際は、そのツールではどのような定義式が用いられているか、は最低限確認しておく必要がある。

<信号の一部を切り出してフーリエ変換>

先の例では、元の信号のすべてのデータを使ってフーリエ変換を行った。しかし、時間的に変化する信号に対しては、そのデータの一部を切り出してフーリエ変換を行うのが一般的である。

以下はデータを切り出すために使用する窓関数の一例（矩形窓，ハン窓）である。フーリエ変換では、切り出した信号を 1 周期とする周期性が仮定されるため、例えば、矩形窓で切り出した場合、切り出し位置によっては信号の両端で不連続性を生じることになる。一方、ハン窓は信号の両端を 0 に減衰させる作用を持つため、この不連続性は回避できる。

窓関数は他にも色々な種類があり、どの窓を使用するかは目的・用途による。また、フーリエ変換はもともと定常信号に対する解析手法であるため、切り出す際の窓サイズも対象となる信号の性質や目的等に応じて適切な設定が必要であることに留意する。

```
import scipy
import matplotlib.pyplot as plt

win_size = 64 #窓のサイズ(切り出すデータのサイズ)
window1 = scipy.signal.windows.boxcar(win_size)

fig, ax = plt.subplots()
ax.plot(window1, color="black", label="boxcar")

window2 = scipy.signal.windows.hann(win_size)
ax.plot(window2, color="red", label="hann")
ax.set_ylabel("Amplitude")
ax.set_xlabel("Sample")

ax.legend(loc=0)
plt.show()
```

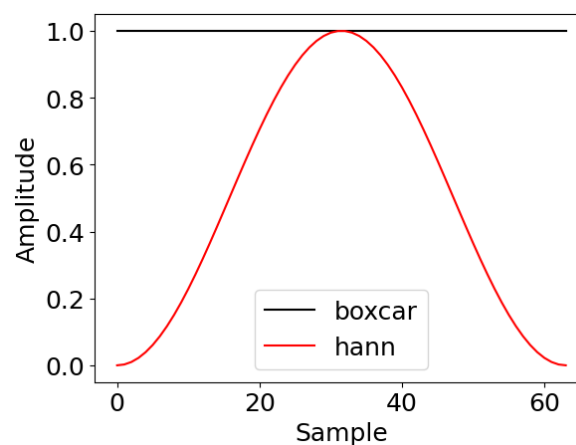


図 2-2: 窓関数

【データの一部を切り抜いて FFT】

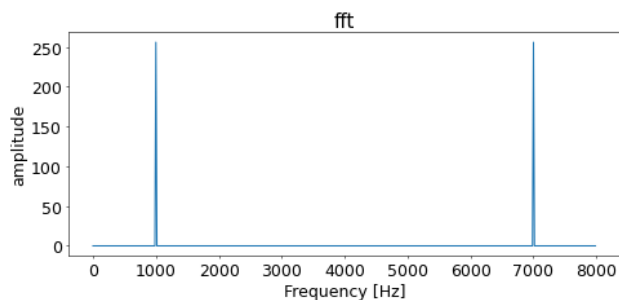
```
import numpy as np
import matplotlib.pyplot as plt
import scipy
from scipy.fft import fft, ifft
plt.rcParams["font.size"] = 18

sr = 8000 # サンプリングレート
f = 1000 # 正弦波の周波数
n = np.arange(0, sr) # 0 <= n < sr の範囲で間隔 1 のデータを生成
x1 = 1*np.sin(2.0*np.pi*f*n/sr) # 振幅 1, 周波数 1000Hz
st = 0 # データの切り出しの位置
win_size = 512
win = scipy.signal.windows.boxcar(win_size) # 窓の種類: 矩形窓
# win = scipy.signal.windows.hann(win_size) # 窓の種類: hann 窓
cut_x = x1[st:st+win_size]*win

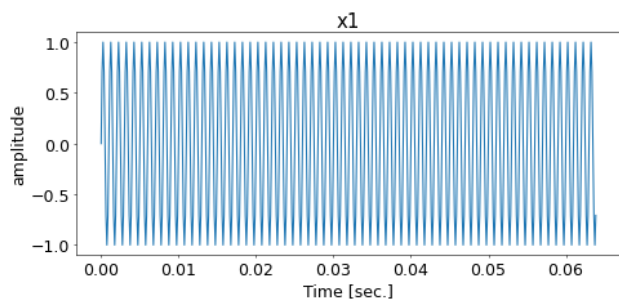
X1 = fft(cut_x)
fig, axes = plt.subplots(figsize=(10,5), tight_layout=True, dpi=50)
freq1 = np.arange(0, len(X1))/len(X1) * sr # 横軸を"周波数"で表すためのダミー変数
axes.plot(freq1, np.abs(X1)) # 振幅スペクトルの表示
axes.set_title('fft')
axes.set_xlabel("Frequency [Hz]")
axes.set_ylabel("amplitude")

reconst_x1 = ifft(X1)
fig, axes = plt.subplots(figsize=(10,5), tight_layout=True, dpi=50)
axes.plot(n[st:st+win_size]/sr, cut_x)
axes.set_title('x1')
axes.set_xlabel("Time [sec.]")
axes.set_ylabel("amplitude")

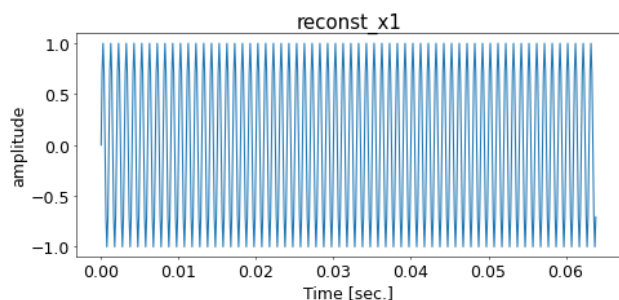
fig, axes = plt.subplots(figsize=(10,5), tight_layout=True, dpi=50)
axes.plot(n[st:st+win_size]/sr, np.real(reconst_x1))
axes.set_title('reconst_x1')
axes.set_xlabel("Time [sec.]")
axes.set_ylabel("amplitude")
```



(a) x1 の振幅スペクトル

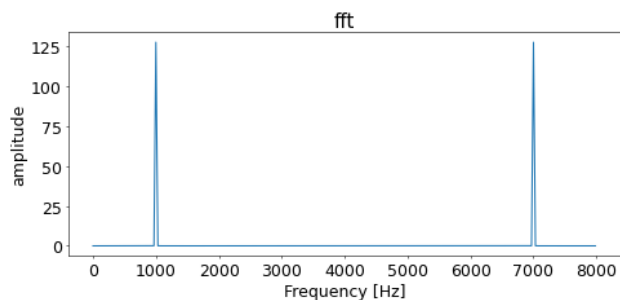


(b) 矩形窓で切り出した信号

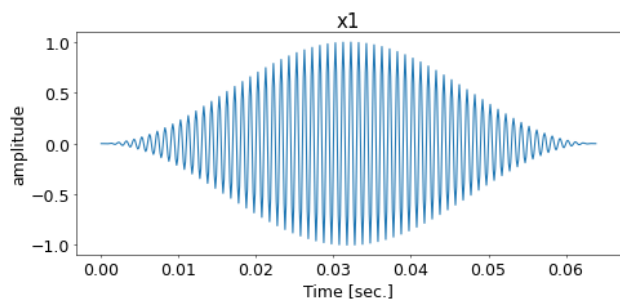


(c) 逆変換で再構成した信号

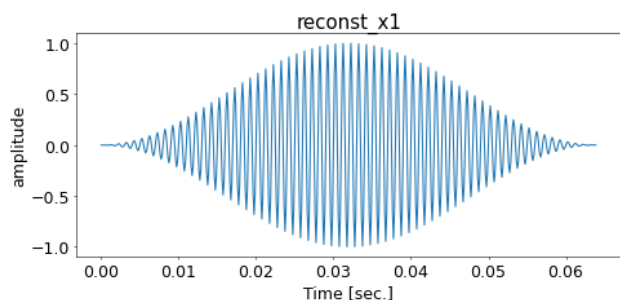
図 2-3: 矩形窓での処理



(a) x1 の振幅スペクトル



(b) hann 窓で切り出した信号



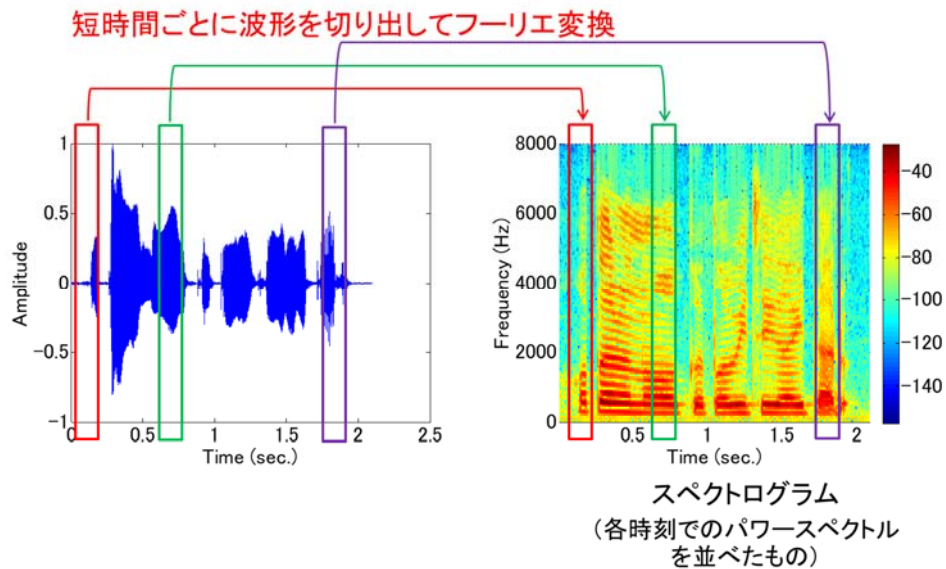
(c) 逆変換で再構成した信号

図 2-4: ハン窓での処理

矩形窓によるスペクトルの大きさが 250 程度（具体的には約 256）になっているのは、先に説明したように FFT する際にデータ点数で割っていないためである。この例では $N = 512$ なので、データ点数で割れば、元の信号の振幅値 1 の半分である 0.5 となる。一方、hann 窓を用いた場合、その大きさは矩形窓を用いた場合の約半分の値になっている。これは、hann 窓の面積が矩形窓の半分になっており、切り出した際に信号のパワーが半分に減少しているためである。いずれにせよ、信号を FFT する際には、窓関数の影響も含まれていることを認識しておく必要がある。

<スペクトログラム>

窓で切り出したデータに対して周波数分析を行い、その切り出し位置をずらしながら連続して行う。色によって信号成分の強さを表すことで、音の時間的な変化、音色、高さ、大きさなどを同時に読み取ることができる。



```
frq, t, Pxx = scipy.signal.stft(xx, fs=sr) #周波数、時間、強さの3つの情報が帰ってくる  
Pxx = 10 * np.log(np.abs(Pxx)) #対数表示に直す  
plt.pcolormesh(t, frq, Pxx, cmap = 'jet')  
plt.show()  
plt.close()
```

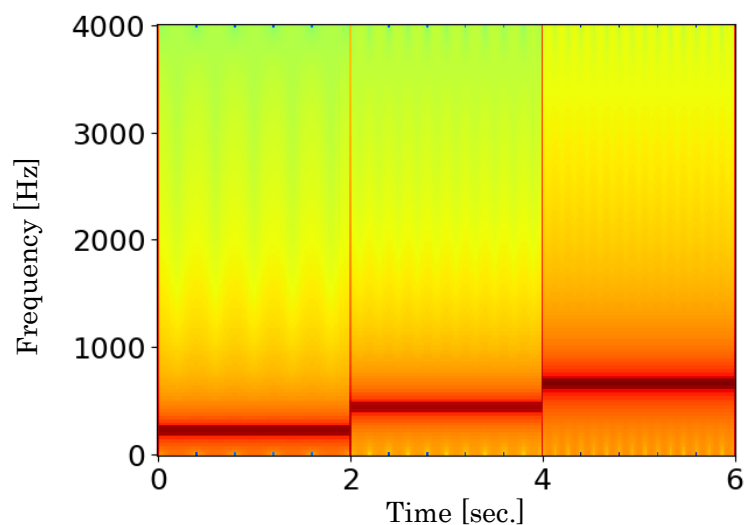


図 2-5: スペクトログラム

【周波数解析と周波数領域処理に関する練習】

練習 2-1

練習 1-1, 1-2, 1-3 で生成した信号の全サンプルを用いてフーリエ変換し，それぞれの振幅スペクトルをグラフ化せよ。

練習 2-2

練習 1-2 で生成した信号 $x(n)$ に関して，0 秒，1 秒，2 秒の位置を基準として，それぞれそこから 512 サンプル分のデータを抜き出してフーリエ変換せよ。また，切り出した部分のそれぞれの時間波形と振幅スペクトルをグラフ化せよ。

練習 2-3

複素スペクトルにおいて，対応する周波数のスペクトルを 0 に置き換えることで，その周波数成分を除去できる。このことを利用して，帯域が 1000Hz～3000Hz に制限されたホワイトノイズを生成せよ。（練習 1-3 で生成したホワイトノイズをフーリエ変換し，所望の帯域の成分以外を 0 にして逆フーリエ変換する。ただし， $f_s/2$ でスペクトルが対称性を持つことに注意が必要である）

3. デジタルフィルタ

デジタルフィルタは、その構成法の違いから、有限長インパルス応答（FIR：Finite Impulse Response）フィルタと、無限長インパルス応答（IIR：Infinite Impulse Response）フィルタに大別される。それぞれに特徴があるが、ここでは FIR フィルタについて取り上げる。

3.1 時間領域フィルタリング

FIR フィルタによるフィルタリングは、入力信号を $x(n)$ 、出力信号を $y(n)$ とすると、

$$y(n) = \sum_{i=0}^M a(n)x(n-i) \quad (3.1)$$

によって与えられる。ここで、 $a(n)$ はフィルタ係数、 M はフィルタ次数、 $M+1$ のことをタップ数、と呼ぶ。なお、FIR フィルタの場合、フィルタ係数 $a(n)$ を「インパルス応答」とも呼ぶ。

式(3.1)は、離散時間でのたたみ込み（Convolution：コンボリューション）である。なお、詳細な説明は省略するが、たたみ込みには「直線たたみ込み（線形たたみ込み）」と「巡回畳み込み」が存在し、式(3.1)は「直線たたみ込み」であることに注意する。

以下は、FIR フィルタの実装例であり、

$a(n) = \left[\frac{1}{5}, \frac{1}{5}, \frac{1}{5}, \frac{1}{5}, \frac{1}{5} \right]$ の場合、ローパスフィルタの役割を果たす。

ちなみに Python では、簡単にフィルタリングを実現するために、`signal.lfilter` や `signal.convolve` の関数が既に用意されている。

```
ntap = 5 #フィルタのタップ数
a=np.ones(ntap)/ntap #分子係数

#時間領域フィルタリング
y = np.zeros(len(x)) # y を 0 で初期化
buff_x = np.zeros(len(a))

for num in range(0,len(x)):
    buff_x = np.append(x[num], buff_x[0:-1])

    for k in range(0,len(a)):
        y[num] += a[k]*buff_x[k]
```

また, Python でフィルタの周波数特性を確認するには, SciPy の signal にある freqz 関数や group_delay 関数が便利である。以下は, タップ数 5 の移動平均フィルタの周波数特性である。

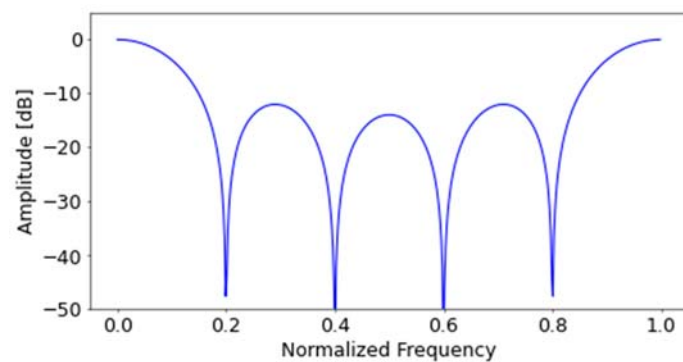
```
import numpy
from scipy import signal
import matplotlib.pyplot as plt
import numpy as np

ntap = 5 #フィルタのタップ数
a=np.ones(ntap)/ntap #分子係数
w, h = signal.freqz(a, 1, whole = True)

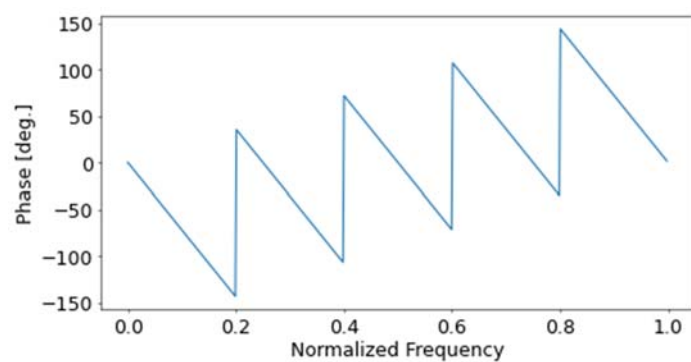
fig, ax = plt.subplots(figsize=(10, 5), dpi=50)
ax.plot(w/(2*np.pi), 20.0 * np.log10(abs(h)), 'b')
plt.ylim(-50, 5)
ax.set_xlabel("Normalized Frequency")
ax.set_ylabel("Amplitude [dB]")

phase = numpy.angle(h) / np.pi * 180.0 # 位相計算
fig, ax = plt.subplots(figsize=(10, 5), dpi=50)
ax.plot(w/(2*np.pi), phase)
ax.set_xlabel("Normalized Frequency")
ax.set_ylabel("Phase [deg.]")

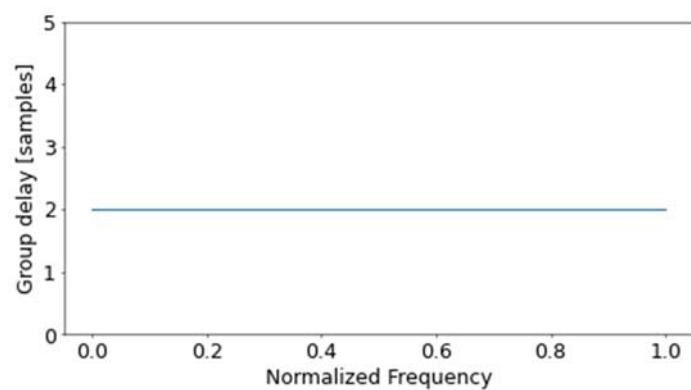
_, gd = signal.group_delay([a,1], w=w)# 群遅延計算
fig, ax = plt.subplots(figsize=(10, 5), dpi=50)
ax.plot(w/(2*np.pi), gd)
plt.ylim(0, ntap)
ax.set_xlabel("Normalized Frequency")
ax.set_ylabel("Group delay [samples]")
```



(a) 振幅特性



(b) 位相特性



(c) 群遅延特性

図 3-1: 周波数特性

練習 3-1

3 つの異なる周波数： $f_1 = 500$ Hz, $f_2 = 1500$ Hz, $f_3 = 2500$ Hz をもつ信号（振幅 $A_1 = A_2 = A_3 = 0.1$, サンプル周波数 $f_s = 8000$ Hz, 時間 3 秒間）を入力信号として,

係数が $a(n) = \left[\frac{1}{5}, \frac{1}{5}, \frac{1}{5}, \frac{1}{5}, \frac{1}{5}\right]$ であるフィルタに通して出力信号を得よ。

また, 入力信号と出力信号, 及びフィルタ係数をフーリエ変換し, その振幅特性をそれぞれ図示せよ。
ただし, フーリエ変換の際の点数は, 入力信号の長さに合わせる。また横軸は周波数とすること。

$a(n) = \left[\frac{1}{5}, \frac{1}{5}, \frac{1}{5}, \frac{1}{5}, \frac{1}{5}\right]$ を入力信号のサイズに合わせてフーリエ変換する（後ろに 0 を詰める）には

例えば, $h = \text{np.zeros}(\text{len}(x))$

$h[0:\text{len}(a)] = a$

などとして, h をフーリエ変換することが考えられる。他にも `np.pad()` を利用して 0 を詰めることも考えられる。

3.2 周波数領域フィルタリング

式(3.1)の時間領域でのたたみ込み処理は、インパルス応答 $a(n)$ の次数が高い場合、非常に計算コストが高い。そこで、時間領域の畳み込みが、周波数領域では掛け算であることを利用して、高速フーリエ変換 (FFT: Fast Fourier Transform) と逆高速フーリエ変換 (IFFT: Inverse FFT) を用いて、周波数領域でフィルタリングする手法がしばしば用いられる。

FFT を記号 $\text{FFT}[\cdot]$ で、また IFFT を記号 $\text{IFFT}[\cdot]$ で表すことにすると、信号 $x_1(n)$ とインパルス応答 $a(n)$ のたたみ込みは、以下の関係が成立つ。

$$X_1(k) = \text{FFT}[x_1(n)], \quad (n, k = 0, 1, 2, \dots, 2N - 1)$$

$$A(k) = \text{FFT}[a(n)], \quad (n, k = 0, 1, 2, \dots, 2N - 1)$$

$$y(k) = \text{IFFT}[X_1(k) \times A(k)], \quad (k = 0, 1, 2, \dots, 2N - 1)$$

ここで、 N は処理するデータの数を表す。もし、 $x_1(n)$ と $a(n)$ の長さが等しくない場合は、ゼロ値を付け加えることで長さを揃える必要がある。

図 3.1 に周波数領域で「直線たたみ込み」を実現するための処理概要を示す。図 3.1 に示すように、周波数フィルタリングでは、元の信号 $x(n)$ をサイズ N の窓で切り出し、それを shift_len ずらしながら処理する。FFT する際に 0 詰めによってサイズを $2N$ とするのは直線たたみ込みを実現するためである（サイズ N のまま処理した場合は巡回畳み込みに対応する）。また、 shift_len のサイズは窓の種類や処理目的などによって異なるが、一般には「矩形窓を使用した場合は N 」, 「ハン窓を使用した場合には $N/2$ 」とすることが多い。

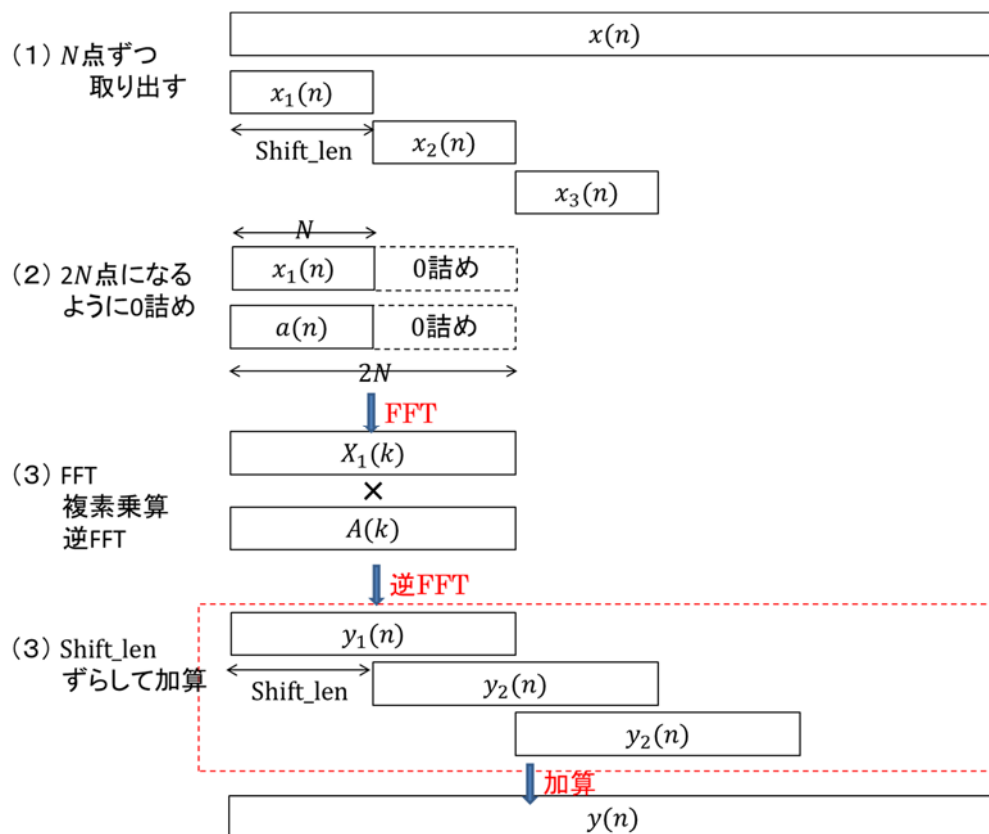


図 3.1 FFT, IFFT による周波数領域フィルタリングの概要

練習 3-2

下記のサンプルプログラムは、入力信号を矩形窓（窓サイズ 512）で切り出し、窓を 512 サンプルずつシフトしながら周波数領域で処理するプログラム例である。ただし、フィルタ係数 $a(n) = [1, 0, 0, 0]$ なので、実質周波数領域では何も処理せずに戻すだけである。以下を確認せよ。

- (1) $f_1 = 500$ Hz, 振幅 $A_1 = 0.1$ の正弦波信号を入力信号し、逆 FFT によって得られる出力信号が、入力信号に一致することを確認せよ。
- (2) 窓の種類、シフトサイズを変えて処理し、入力と出力の関係（出力にどのような影響があるか）を確認せよ。
- (3) 3 つの異なる周波数: $f_1 = 500$ Hz, $f_2 = 1500$ Hz, $f_3 = 2500$ Hz をもつ信号（振幅 $A_1 = A_2 = A_3 = 0.1$, サンプル周波数 $f_s = 8000$ Hz, 時間 3 秒間）を入力信号, 係数を $a(n) = \left[\frac{1}{5}, \frac{1}{5}, \frac{1}{5}, \frac{1}{5}\right]$ とし得られる出力信号が、先の時間領域処理の結果を一致することを確認せよ。

```
N = 512      # 窓のサイズ
shift_len = N # 窓のシフトサイズ
w = scipy.signal.windows.boxcar(N) # 窓の種類(矩形窓)
#w = scipy.signal.windows.hann(win_size)

x_len = len(x0)      # 元の信号 x の長さ
x = np.pad(x0, [0, N*2], "constant")
a = np.zeros(4) #分子係数
a[0] = 1
h = np.zeros(2*N)
h[0:(len(a))] = a
H = rfft(h)

y = np.zeros((len(x)))

n_frame = x_len // shift_len + 1 #フレーム数
for i in range(n_frame):
    # 取り出した x を FFT する
    x_N = np.pad(w*x[i*shift_len:i*shift_len+N], [0, N], 'constant') # 0 埋め
    X = rfft(x_N)
    # フィルタ H と X を掛ける
    Y = X * H
    # 逆 FFT をして足し合わせる
    y[i*shift_len:i*shift_len+2*N] += irfft(Y)
```


練習 3-3

フォルダ「room_impulse」には、実際の教室やホールで収録されたインパルス応答データが保存されている。

- class_sample.wav
- octa_sample.wav
- great_sample.wav

すべて、サンプリング周波数は44.1 kHz である。

適当な wav ファイル（例えば、配布した trumpet.wav）とインパルス応答のたたみ込み処理を周波数領域で実現するプログラムを作成せよ。

4. HRTF による3D サウンド

4.1 HRTF(Head Related Transfer Function:頭部伝達関数)

音波は鼓膜に届くまでに頭や耳介、あるいは胴体の影響を受ける。このような、頭部周辺による入射音波の物理特性の変化を周波数領域で表現したものを頭部伝達関数という(頭部伝達関数を逆フーリエ変換した時間領域信号は、頭部インパルス応答と呼ばれる)。例えば、図 4.1 のように右側に音源があるとき、音源と左耳の間には何もないため、音はあまり変化なく右耳に伝わります。一方、音源と左耳の間には頭や鼻という障害物があるため、左耳に伝わる音は減衰したり、右耳と比べると遅れて音が届いたりする。

HRTF は角度によって異なり、この HRTF (頭部伝達関数) を好きな音源に「たたみ込み処理」することで様々な角度・高さに音を移動させることができる。ただし、HRTF には人の頭や耳の特性が含まれているため人によって同じ角度に対しても特性は異なる。それゆえ、他人の頭部伝達関数を使うと前後方向については上手く定位させることができないが、左右方向については上手く定位させることができる(と思う)。

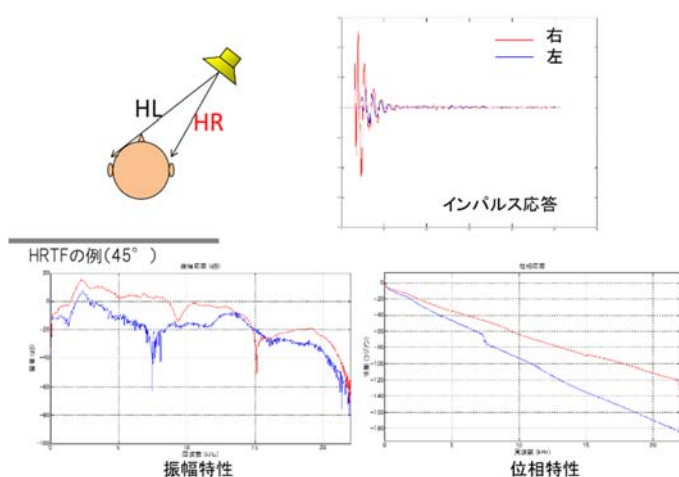
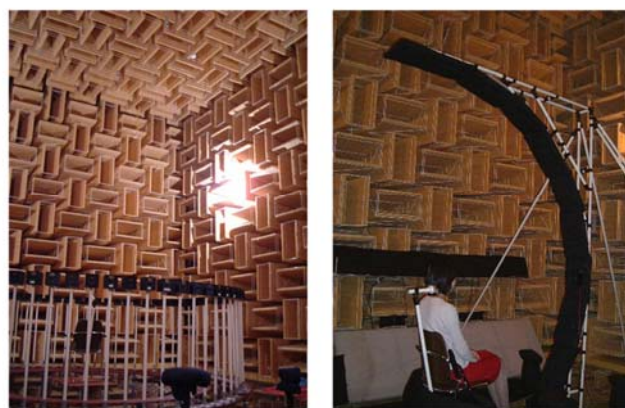


図 4.1 頭部伝達関数の例

無響室(長岡技術科学大学 音響振動工学センター)



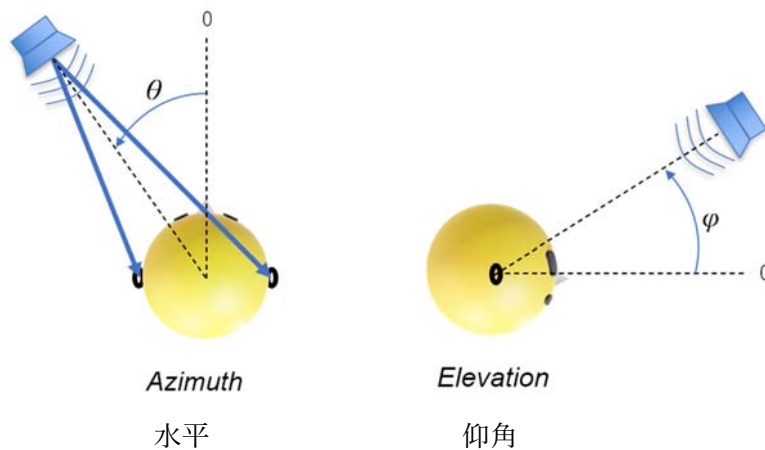
頭部伝達関数の測定風景

図 4.2 頭部伝達関数の測定風景

配布したフォルダ「hrtfs」には、仰角方向に対して $0^{\circ} \sim 90^{\circ}$ まで 5° 間隔、水平方向に対して $0^{\circ} \sim 355^{\circ}$ まで 5° 間隔で測定された頭部インパルス応答(HRIR)が記録されている。例えば、フォルダ「elev0」は仰角 0° の場合であり、その中に $0^{\circ} \sim 355^{\circ}$ まで 5° ごと異なる水平角の HRIR が DAT ファイルとして記録されている。なお、サンプリング周波数は 44.1kHz で、HRIR の長さは 512 点である。

HRTF データベース：<https://sites.google.com/site/takanorinishinomu/research/hrtf/database-j>

また配布した python プログラム「3D_sound_template.ipynb」は、仰角 0° で水平角 $0^{\circ} \sim 355^{\circ}$ に対して 5° 間隔で記録された頭部インパルス応答(HRIR)を読み込み、それらをフーリエ変換することで頭部伝達関数(HRTF)を得ている。例えば、 $\text{HRTF_L}[0,:]$, $\text{HRTF_R}[0,:]$ は 0° 方向における左右の HRTF, $\text{HRTF_L}[1,:]$, $\text{HRTF_R}[1,:]$ は 5° 方向における左右の HRTF, といった具合で、全 72 方向の HRTF が生成される。



それを踏まえて、以下のプログラムを作成せよ。

練習 4-1

指定したある一方向の HRTF を用いて、入力信号を周波数領域フィルタリングするプログラムを作成せよ。

練習 4-2

フレームごとに使用する HRTF が一つずつ切り替わる（最初フレームは 0° 方向、次のフレーム 5° 方向、その次のフレーム 10° 方向、 \dots 355° 方向、 0° 方向、 \dots という感じで切り替わる）プログラムを作成せよ。

練習 4-3

音源が、頭の周りを任意の速度で周るプログラムを作成せよ。

ヒント： 任意の速度で回転させる場合、現在処理しているフレームにどの方向の HRTF を掛ければ良いかを計算する必要がある。しかし、今回の HRTF データベースは 5° 間隔でしか存在しないため、例えば、処理すべき HRTF の角度が 8° の場合、存在する 5° 間隔の HRTF から新たに生成する（補間する）必要がある。音源位置の計算と HRTF の生成方法は以下を参照。

【音源位置の計算】

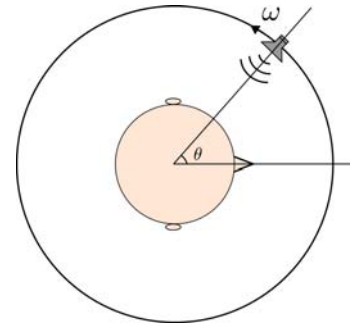
音源が頭の周りをまわっている様子を右図に示す。

音源が図のように角速度 ω [deg/s] で周っているとき、サンプル番号 n のときの音源方向位置 θ [deg] は次式で求まる。

$$\theta = \omega \frac{n}{f_s}$$

ここで、 f_s はサンプリング周波数である。

ただし、音源方向の角度が 360° 以上の場合、プログラム上扱いにくいので、 0° 以上 360° 未満となるようにする必要がある。



音源位置の計算

【HRTF の補間】

ここでは、線形補間法を用いてデータベースに存在しない HRTF の補間を行う。計算は非常に簡単で、次式で所望方向の HRTF を作成できる。

$$H[k] = rH_1[k] + (1-r)H_2[k]$$

ここで、 r は $0 \leq r \leq 1$ は 2 つの HRTF の内分比を意味する。例えば、音源方向角度が 8° の場合、 H_1 は 5° 方向の HRTF、 H_2 は 10° 方向の HRTF を用いて、

$$r = \frac{H_2 - H}{H_2 - H_1} = \frac{10 - 8}{10 - 5} = \frac{2}{5} = 0.4$$

とすればよい。

5. 適応フィルタ

図 5-1 に示すように、フィルタの出力 $y(n)$ と所望信号 $d(n)$ の誤差 $e(n)$ が小さくなるように（すなわち、 $y(n)$ が $d(n)$ に近づくように）、与えられた手順に従ってフィルタ係数を逐次更新するフィルタを「適応フィルタ」と呼ぶ。エコーキャンセラやノイズキャンセラ、未知システムの実験同定など、多くの場面で利用される。

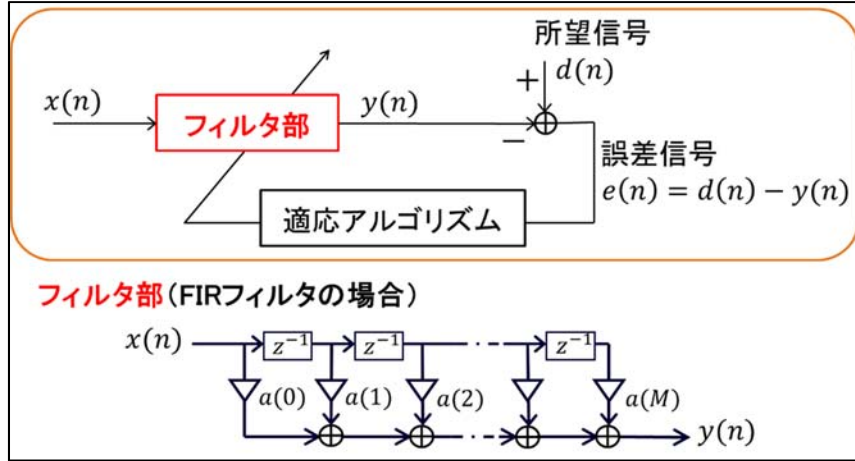


図 5-1: 適応フィルタの概要

係数更新のための代表的な適応アルゴリズムとして：

- LMS アルゴリズム (Least Mean Square)
- NLMS アルゴリズム (Normalized LMS)
- RLS アルゴリズム (Recursive Least Square)

などが知られている。以下では、NLMS アルゴリズムによるフィルタ係数の更新方法を簡単に示す。

今、フィルタへの入力信号 $x(n)$ 、出力を $y(n)$ とすると、次数 M の FIR フィルタの差分方程式は次式で表わせる。

$$y(n) = \sum_{i=0}^M a_n(i)x(n-i) = \mathbf{a}_n^T \mathbf{x}_n = \mathbf{x}_n^T \mathbf{a}_n \quad (5.1)$$

ここで、 M はフィルタ次数であり、 $a_n(i)$ は時刻 n でのフィルタ係数を意味する。また、係数ベクトル \mathbf{a}_n と入力信号ベクトル \mathbf{x}_n はそれぞれ以下の通りである。

$$\mathbf{a}_n = [a_n(0) \ a_n(1) \ a_n(2) \ \cdots \ a_n(M)]^T \quad (5.2)$$

$$\mathbf{x}_n = [x(n) \ x(n-1) \ x(n-2) \ \cdots \ x(n-M)]^T \quad (5.3)$$

このとき、NLMS アルゴリズムを用いると、時刻 $n+1$ でのフィルタ係数 \mathbf{a}_{n+1} は次式によって更新される。

$$\mathbf{a}_{n+1} = \mathbf{a}_n + \mu \frac{1}{\beta + \|\mathbf{x}_n\|} \mathbf{x}_n e(n) \quad (5.4)$$

$$\|\mathbf{x}_n\| = \mathbf{x}_n^T \mathbf{x}_n = x(n)^2 + x(n-1)^2 + \cdots + x(n-M)^2 \quad (5.5)$$

式(5.4)において、 $e(n) = y(n) - d(n)$ 、 μ はステップサイズであり $0 < \mu < 2$ 、また β は小さな正の実数である。

練習 5-1 ～システム同定～

図 5-2 はシステム同定のブロック図である。特性の不明な未知システム(Unknown System)があり, その入力信号 $x(n)$ と出力信号 $d(n)$ は観測できるものとする。もし, 同じ入力信号 $x(n)$ を適応フィルタ (ADF)に与え, その出力 $y(n)$ が $d(n)$ と同じになるならば, その ADF は, 未知システムと同じ特性をもつ (働きをする) と考えられる。これを「システム同定」と呼ぶ。

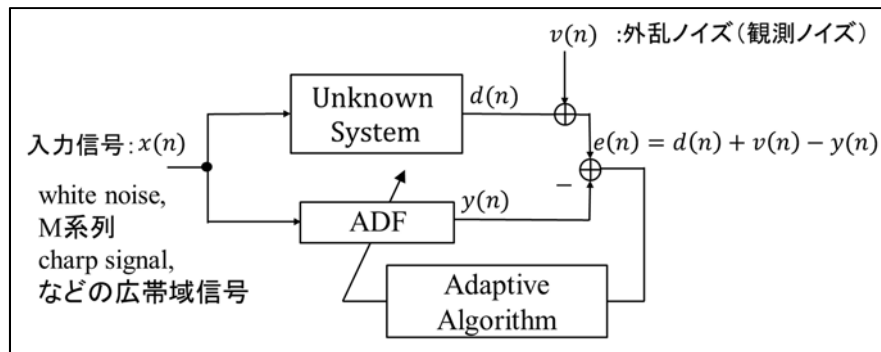


図 5-2：システム同定のブロック図

NLMS アルゴリズム用いてシステム同定するためのプログラムを作成せよ。

ここでは, 未知システム, ADF, 入力信号, 及び外乱信号の各条件は以下のとおりとする。

【条件】

- 未知システム(Unknown System)のインパルス応答は,

```
a = signal.remez(21, [0, 0.2, 0.3, 0.5], [1, 0])
```

により得られる a であると仮定する。
- ADF (FIR フィルタ) の次数は 20 とする (タップ数で言えば, 21 タップ)。
- 入力信号 $x(n)$ は平均 0, 平均パワー1 のホワイトノイズ 10000 サンプルとする。
- 外乱ノイズ $v(n)$ は入力信号 $x(n)$ とは無相関のホワイトノイズとし, 平均 0, 平均パワー 10^{-4} のホワイトノイズとする。

ステップサイズ μ は 0.02, 0.2, 1.0 の 3 パターンで実行し, その時の収束特性 (横軸を n , 縦軸を $MSE = 20 \log_{10} |e(n)|$ として表示したもので, 各繰返しにおける誤差の推移を示すもの: 図 5-3 のようなもの) を図示せよ。

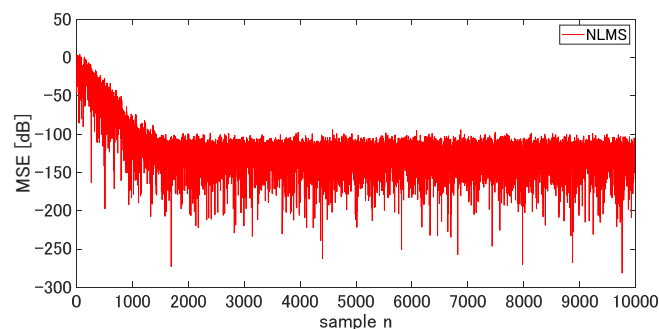


図 5-3: MSE

ひとまず, 以上