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。

1. **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](https://colab.research.google.com/notebooks/welcome.ipynb?hl=ja)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

* 1. **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

* 1. **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

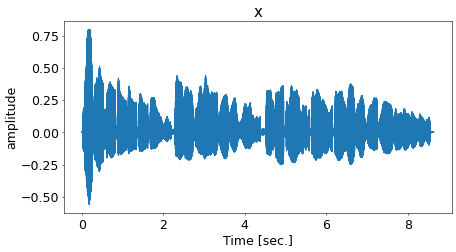
ax.set\_title('x')  # タイトル設定

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．ディジタル信号処理の基礎**

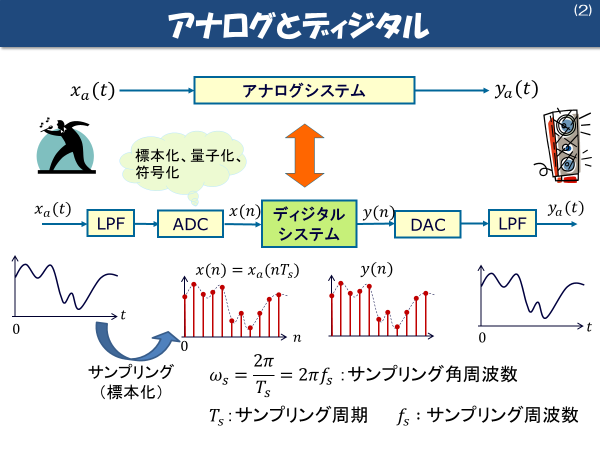
振幅，周波数，位相の正弦波信号は次式で表せる。

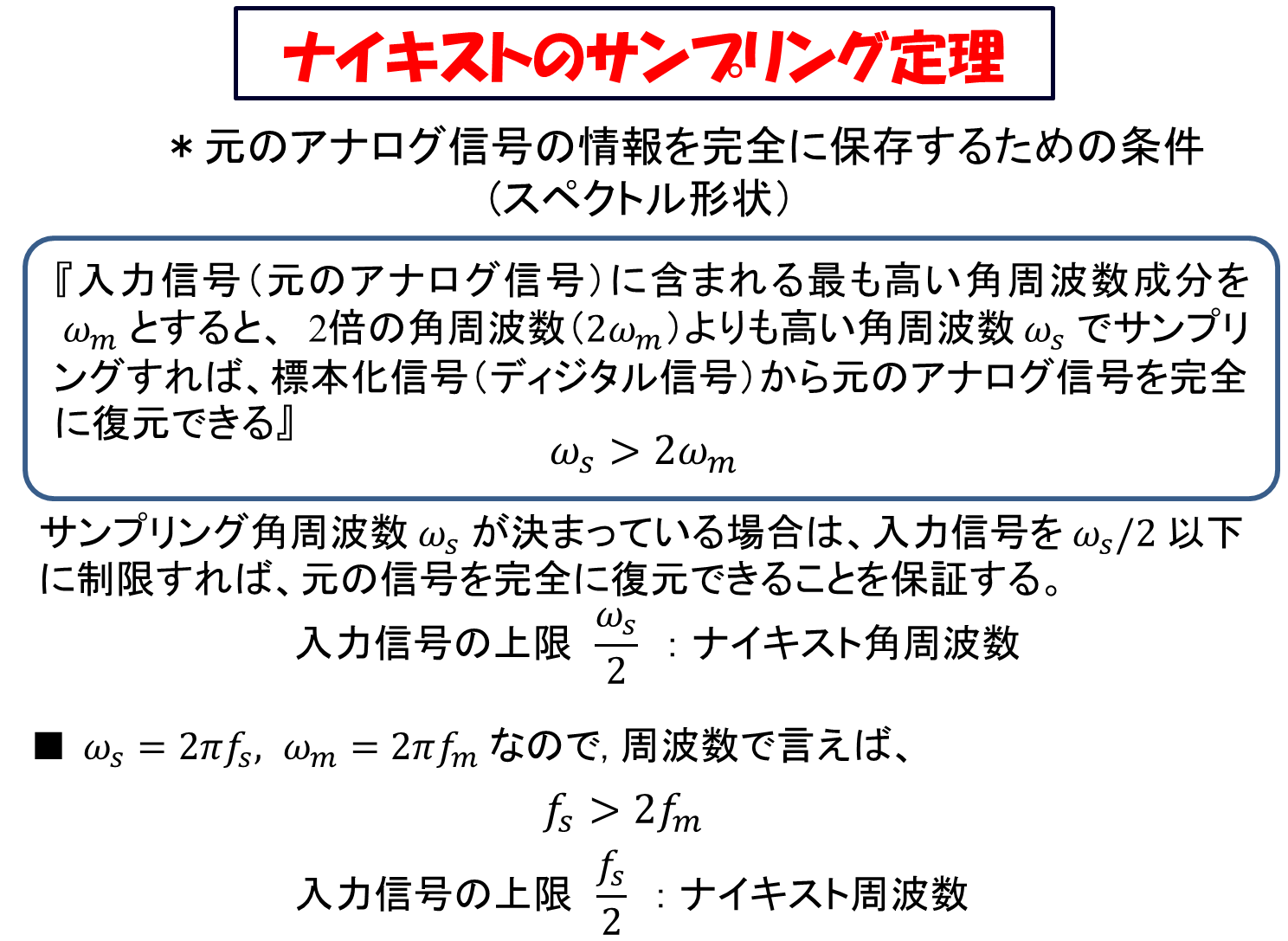
　　　　(1.1)

ここでは連続時間である。

今，をサンプリング周波数 で標本化したとする（すなわち，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。

ただし，以降，離散時間信号をで表記する。





ディジタルの世界では，サンプリング周波数を とすると， 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，位相 ，サンプリング周波数，信号の周波数として，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  # サンプリングレートfs

f = 440  # 正弦波の周波数

n = np.arange(0, 3\*sr) # 0 <= 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.]")

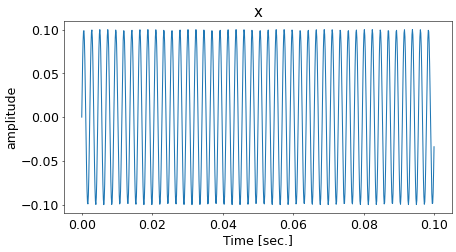
ax.set\_ylabel("amplitude")

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とする。

練習1-1

3つの異なる周波数：，，から成る3秒間の複合信号を生成せよ。

また，振幅，とする。

練習1-2

信号の周波数成分が，， 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=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，

np.random.normal(mean，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)やディ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")

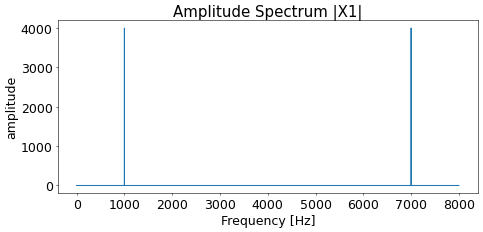
fig, axes = plt.subplots(figsize=(10,5), tight\_layout=True, dpi=50)

axes.plot(n[plt\_st:plt\_end]/sr, np.real(reconst\_x1[plt\_st:plt\_end]))

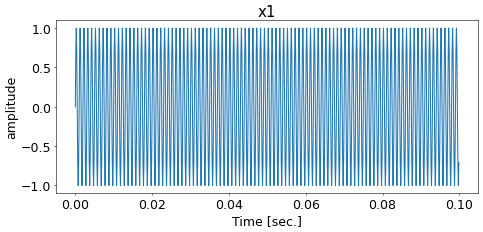
axes.set\_title('reconst\_x1')

axes.set\_xlabel("Time [sec.]")

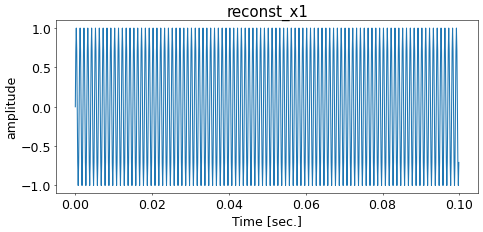
axes.set\_ylabel("amplitude")



(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。

ここで，としたとき，を振幅スペクトル，を位相スペクトルと呼ぶ。また，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，角周波数軸上を等間隔に離散化した点についてスペクトルが評価されることが分かる。

＊角周波数軸上は周波数で言えばである。

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。

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？
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1000Hz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である）。また，今回はサンプリング周波数が8000Hzなので，1000Hz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？

式(2.1)-(2.2)との違いは， 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， と算出されるはずで，確かに，実スペクトルの振幅値のとなる。

　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()

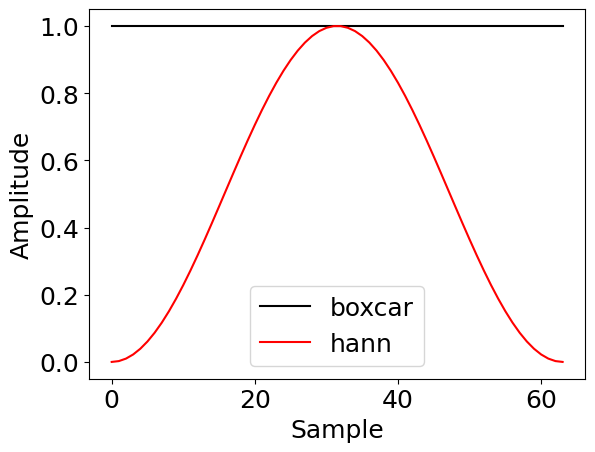


図2-2: 窓関数

【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")

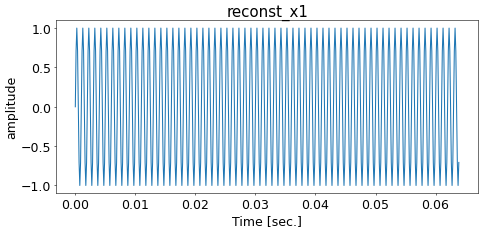
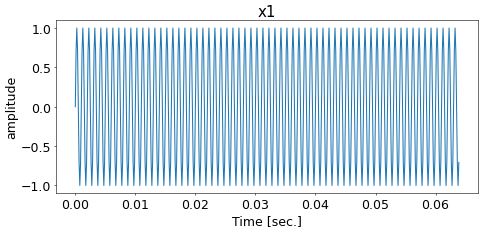
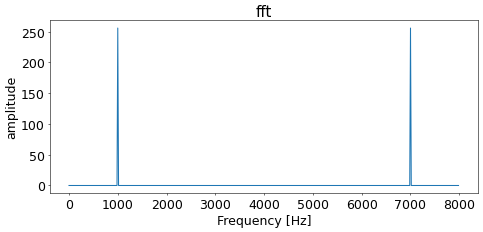
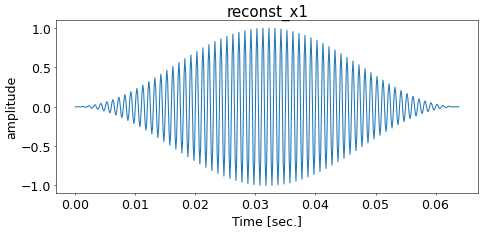
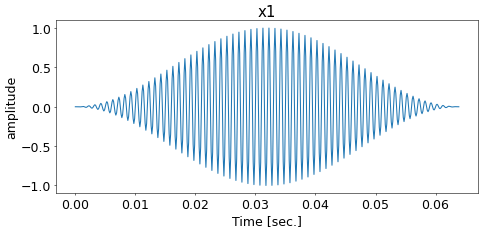
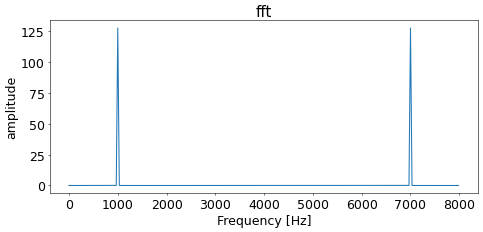


図2-3: 矩形窓での処理

図2-4: ハン窓での処理

(a) x1の振幅スペクトル

(a) x1の振幅スペクトル

(b) 矩形窓で切り出した信号

(b) hann窓で切り出した信号

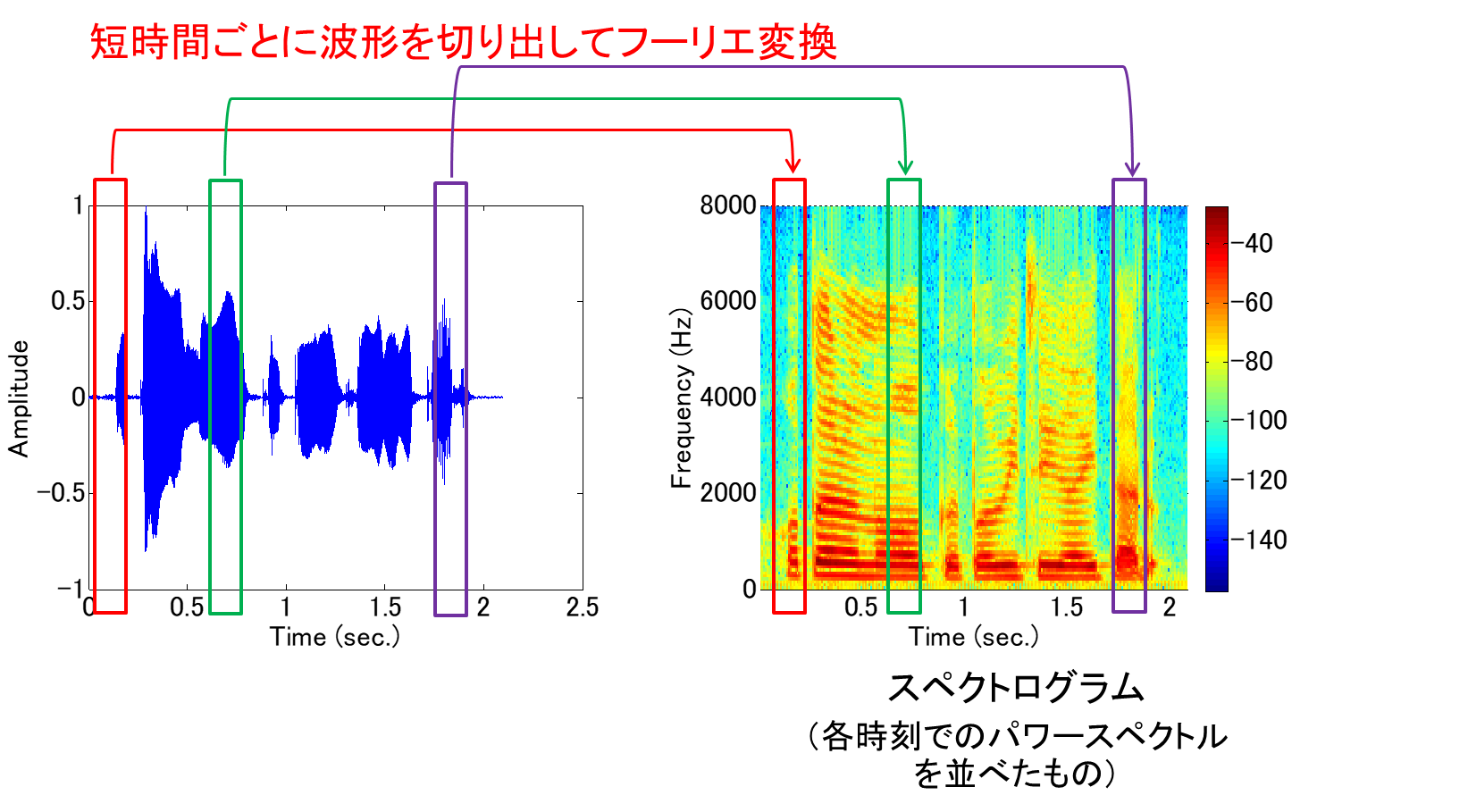
(c) 逆変換で再構成した信号

(c) 逆変換で再構成した信号

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。この例ではなので，データ点数で割れば，元の信号の振幅値の半分であるとなる。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。



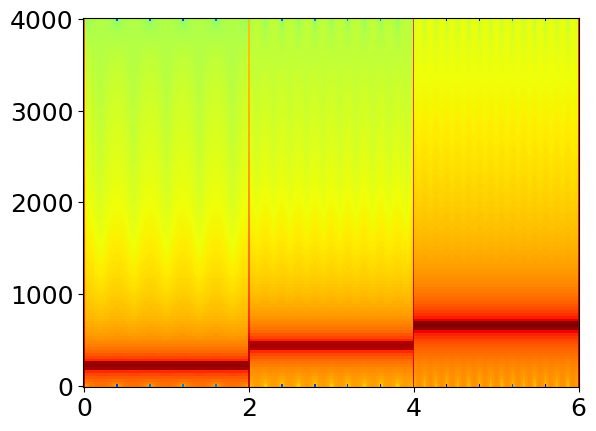
frq, t, Pxx = scipy.signal.stft(xx, fs=sr) #周波数、時間、強さの3つの情報が帰ってくる

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()



Time [sec.]

図2-5: スペクトログラム

Frequency [Hz]

**【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，）

**３．ディジタルフィルタ**

ディジタルフィルタは，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，出力信号をとすると，

によって与えられる。ここで，はフィルタ係数，はフィルタ次数，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。

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

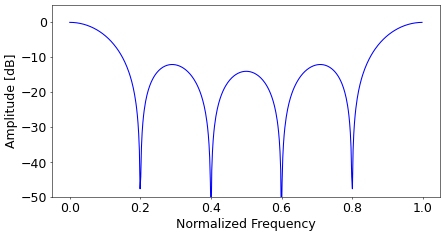
fig, ax = plt.subplots(figsize=(10, 5), dpi=50)

ax.plot(w/(2\*np.pi), gd)

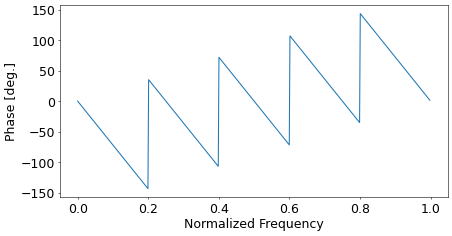
plt.ylim(0, ntap)

ax.set\_xlabel("Normalized Frequency")

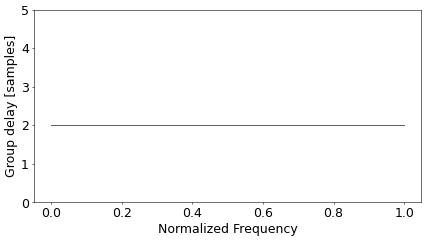
ax.set\_ylabel("Group delay [samples]")



(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：，，をもつ信号（振幅，サンプリング周波数，時間3秒間）を入力信号として，

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こと。また横軸は周波数とすること。

例えば，

などとして，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 FFT[•]で，またIFFTを記号 IFFT[•]で表すことにすると，信号とインパルス応答のたたみ込みは，以下の関係が成立つ。

ここで， は処理するデータの数を表す。もし， と 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，元の信号をサイズの窓で切り出し，それを 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（サイズのまま処理した場合は巡回畳み込みに対応する）。また，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」, 「ハン窓を使用した場合には」とすることが多い。

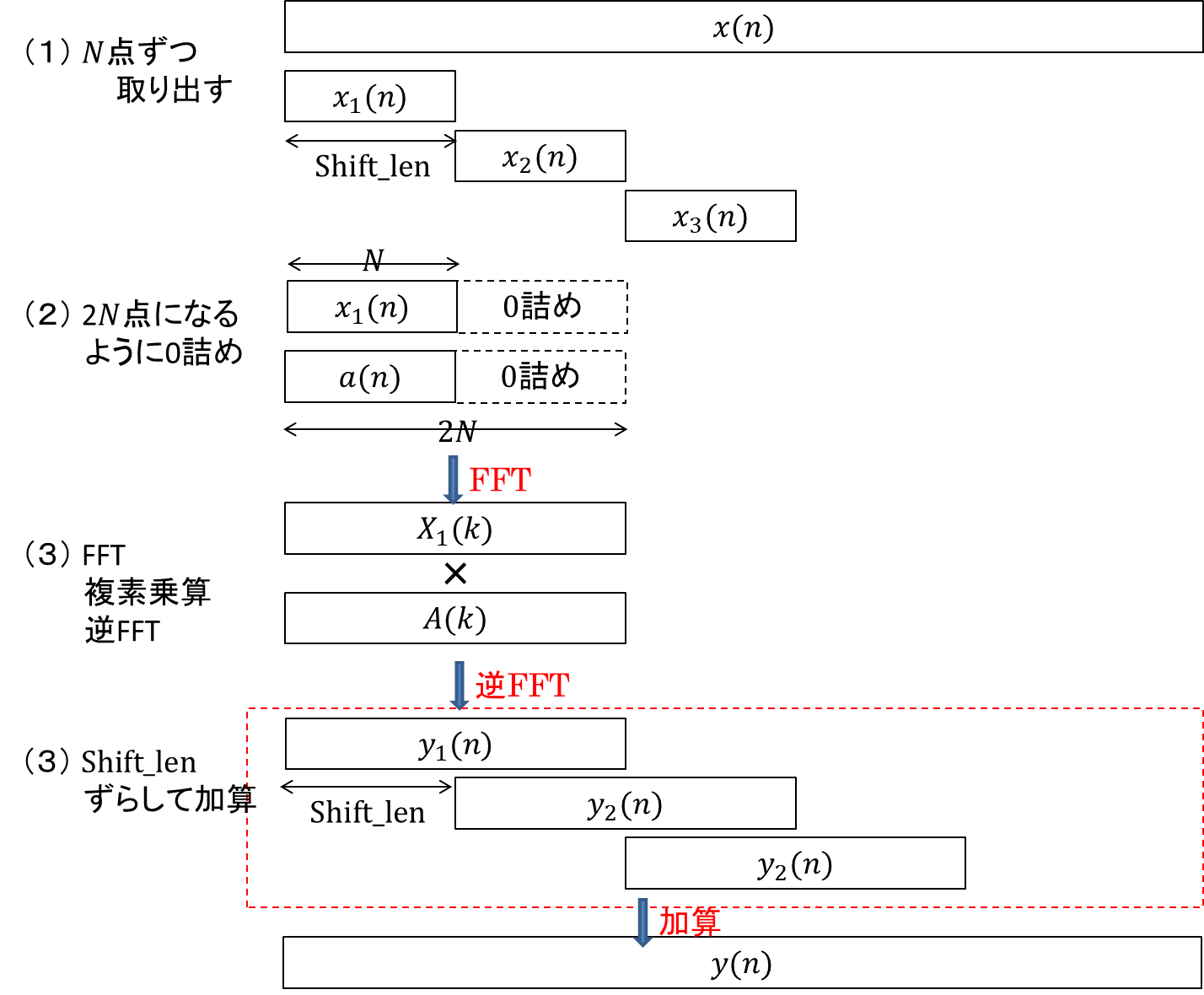


図3.1 FFT, IFFTによる周波数領域フィルタリングの概要

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. ，振幅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：，，をもつ信号（振幅，サンプリング周波数，時間3秒間）を入力信号，係数を として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(ntap) #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音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

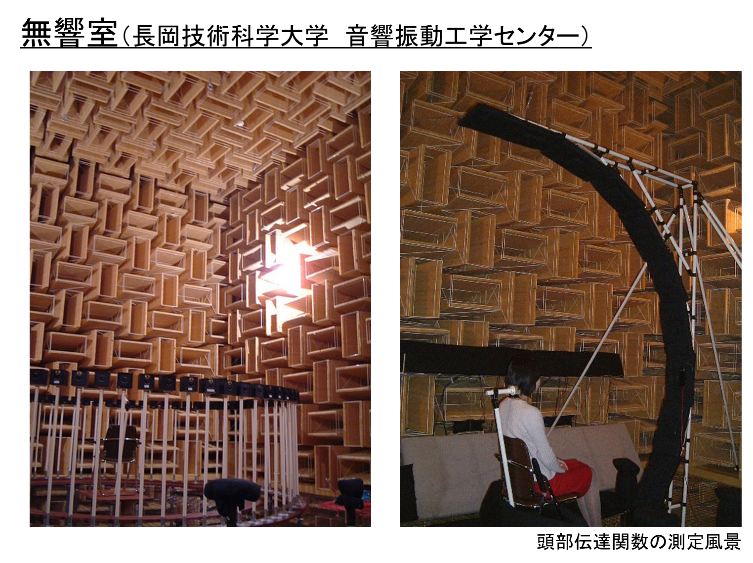
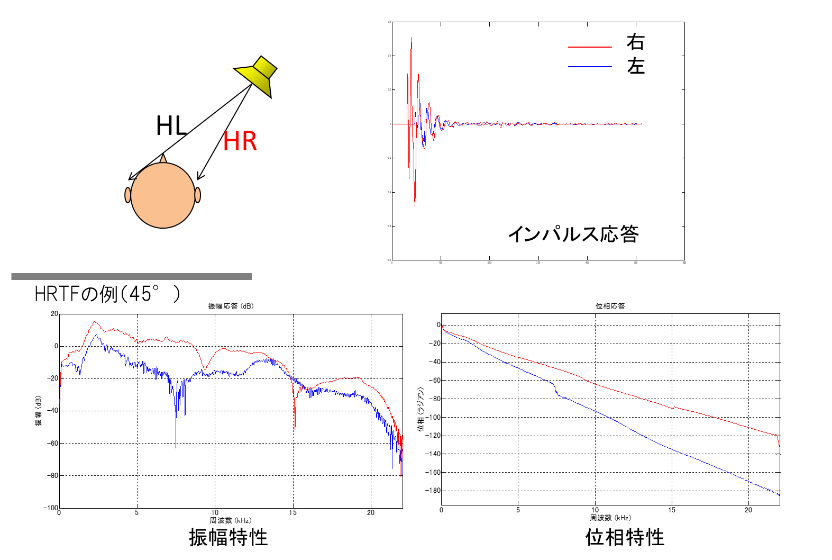
適当なwavファイル（例えば，配布したtrumpet.wav）とインパルス応答のたたみ込み処理を周波数領域で実現するプログラムを作成せよ。

**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。

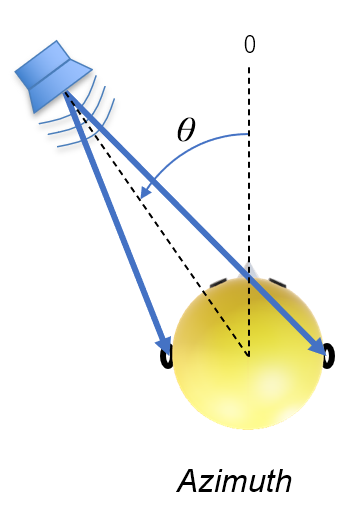
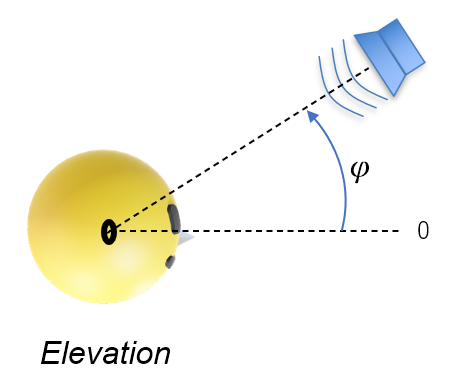
HRTFは角度によって異なり，このHRTF（頭部伝達関数）を好きな音源に「たたみ込み処理」することで様々な角度・高さに音を移動させることができる。ただし，HRTFには人の頭や耳の特性が含まれているため人によって同じ角度に対しても特性は異なる。それゆえ，他人の頭部伝達関数を使うと前後方向については上手く定位させることができないが，左右方向については上手く定位させることができる（と思う）。

　　　　図4.1 頭部伝達関数の例　　　　　　　　　　　　　　　　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°～90°まで5°間隔，水平方向に対して0°～355°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°～355°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°～355°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，HRTF\_L[0,:]，HRTF\_R[0,:]は0°方向における左右のHRTF，HRTF\_L[1,:]，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。

　　　　　　　　　　水平　　　　　　　　　仰角

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。

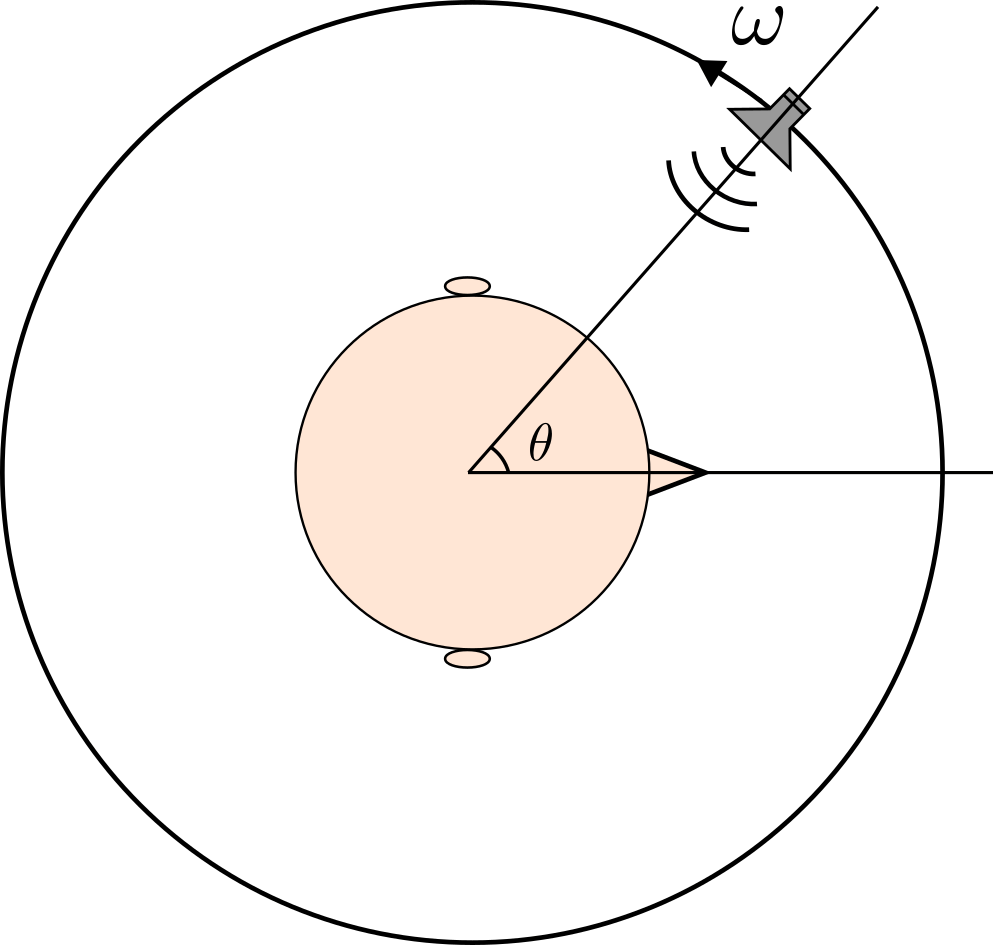
練習4-2

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，・・・355°方向，0°方向，・・・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 [deg/s] で周っているとき，サンプル番号 のときの音源方向位置  [deg] は次式で求まる。

ここで，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。

ここで， はは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，は5°方向のHRTF， は10°方向のHRTFを用い

**5．適応フィルタ**

図5-1に示すように，フィルタの出力と所望信号の誤差が小さくなるように（すなわち，がに近づくように），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，未知システムの特性同定など，多くの場面で利用される。

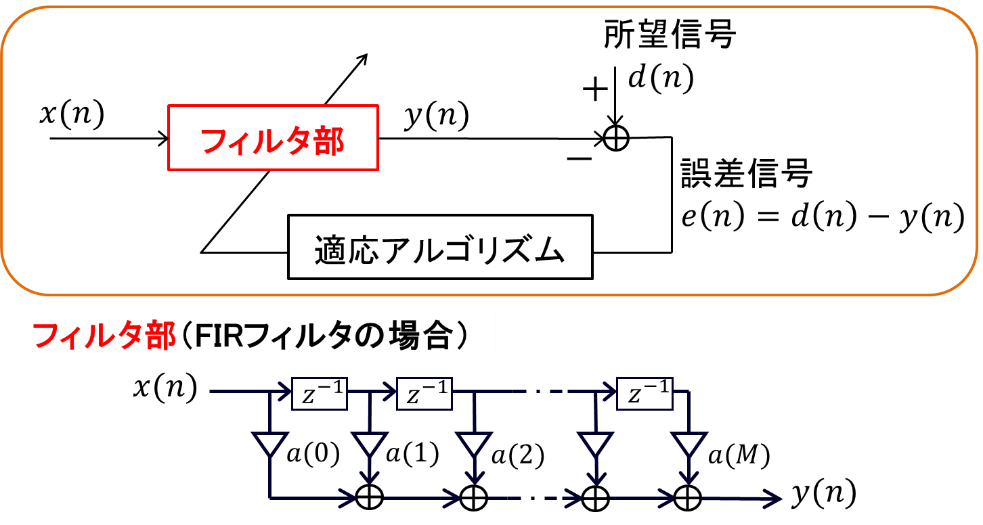


図5-1: 適応フィルタの概要

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，出力をとすると，次数のFIRフィルタの差分方程式は次式で表わせる。

ここで，はフィルタ次数であり， は時刻でのフィルタ係数を意味する。また，係数ベクトルと入力信号ベクトルはそれぞれ以下の通りである。

このとき，NLMSアルゴリズムを用いると，時刻 でのフィルタ係数 は次式によって更新される。

式(5.4)において，はステップサイズであり，また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，その入力信号と出力信号は観測できるものとする。もし，同じ入力信号を適応フィルタ(ADF)に与え，その出力が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。

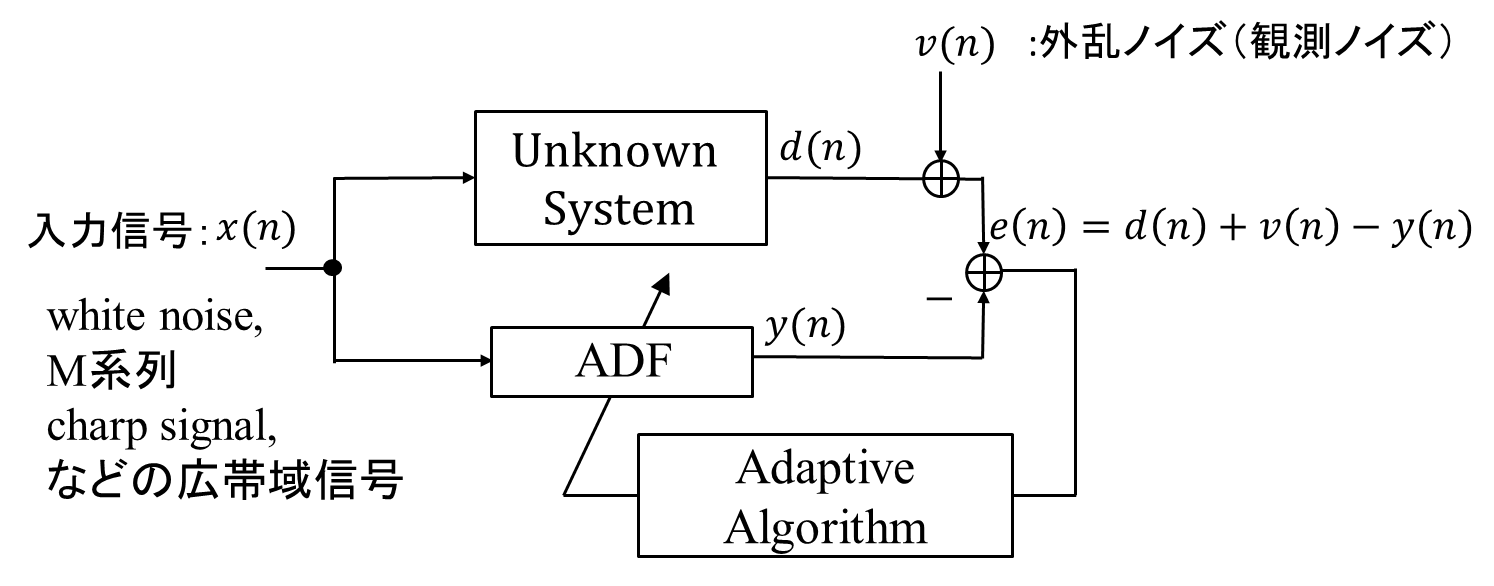


図5-2：システム同定のブロック図

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。
* 外乱ノイズは入力信号とは無相関のホワイトノイズとし，平均0，平均パワーのホワイトノイズとする。

ステップサイズは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，縦軸を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