Processamento Digital de Sinais e Aplicações em Acústica
Tutorial 2 - Sinais e Sistemas

https://github.com/fchirono/Aulas_PDS_Acustica

Objetivos do Tutorial

Ao final desta sessão, você será capaz de:

- Gerar, plotar e realizar operações simples em sinais usando Python.
- Verificar o comportamento de um sistema quanto à sua variância ou invariância temporal.
- Verificar o comportamento não linear de um sistema.
- Calcular a energia total e a potência média de sinais.

Antes de começar

1. Faça o download dos arquivos para este tutorial
2. Inicie a IDE Spyder
3. Copie os arquivos para uma pasta dedicada no seu computador, e selecione esta pasta como o diretório de trabalho do Spyder (aba "Files" no canto superior direito)

Sumário

1 Preliminares em Python **2**
- 1.1 Gerando Sinais . . . 2
- 1.2 Plotando Sinais . . . 3
- 1.3 Operações com Sinais . . . 3

2 Sistemas **4**
- 2.1 Sistemas Variantes/Invariantes no Tempo . . . 4
- 2.2 Sistemas Não Lineares . . . 4
  - 2.2.1 Revisão . . . 4

3 Energia e Potência de Sinais **4**
- 3.1 Revisão . . . 4
- 3.2 Tarefas . . . 5

4 Escrevendo Sinais em Arquivos .WAV **5**

1 Preliminares em Python

Em Python, é muito comum usar módulos externos contendo funções especiais para aplicações específicas. Usaremos inicialmente os módulos `numpy` para computação numérica e `matplotlib.pyplot` para plotagem. Outros módulos, como o `scipy`, também serão usados eventualmente. Esses módulos são importados no início do código, como mostrado abaixo; note que estamos importando o módulo `numpy` usando o identificador `'np'` por conveniência, e que `pyplot` é um submódulo que pertence ao módulo `matplotlib`.

Lembre-se de baixar o arquivo `tutorial1_funcoes.py` do Github e adicioná-lo ao seu diretório de trabalho atual.

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

# Importando explicitamente os modulos 'signal', 'wavfile' e 'loadmat'
import scipy.signal as sss
from scipy.io import wavfile, loadmat

import tutorial1_funcoes as tutorial1
```

Durante o trabalho interativo no console IPython, você pode verificar qual é o diretório de trabalho atual com o comando "mágico" do IPython `%pwd` (acrônimo para *print working directory*), e você pode mudar para um outro diretório de trabalho usando o comando `cd C:\diretorio\desejado` (no Windows). O mesmo princípio se aplica para usuários de Linux e Mac usando a estrutura de pastas apropriada desses sistemas operacionais.

1.1 Gerando Sinais

Gerar um sinal em Python significa criar um vetor (ou, na terminologia Python, um *array Numpy*) de valores. Um exemplo de código para gerar uma onda senoidal com frequência f_0 Hertz, amostrada a cada $\Delta T = 1/fs$ segundo (onde fs é a frequência de amostragem em Hertz) com duração T_{max} segundos é dado abaixo:

```
fs = 44100          # define a frequencia de amostragem (em Hz)
dt = 1./fs      # define o periodo de amostragem (em segundos)
T_max = 1.          # define a duracao do sinal (em segundos)

# cria um array Numpy para as amostras de tempo
t = np.arange(0, T_max, dt)

A = 1.                      # define a amplitude (de pico)
freq = 200.                 # define a frequencia (em Hz)
x_seno = A*np.sin(2*np.pi*freq*t)    # calcula as amostras da onda senoidal
```

(Note o uso da função `np.arange` com um tamanho de passo fracionário, o que não é recomendado por questões de precisão numérica. Como você poderia usar a função `np.linspace` para gerar o mesmo vetor de tempo? Existem pelo menos duas maneiras de fazer isso; você consegue encontrar as duas?)

Outras funções para gerar sinais comumente usados estão incluídas nos módulos `numpy` e `scipy.signal`; algumas sugestões são:

- `np.sin()`,
- `np.cos()`,
- `ss.square()`,

- `ss.sawtooth()` e
- `np.random.randn()` (para gerar ruído branco aleatório).

Leia suas documentações e tente se familiarizar com elas.

1.2 Plotando Sinais

Para plotar os sinais gerados acima, usaremos o módulo `matplotlib.pyplot`. Este módulo fornece uma interface semelhante ao MATLAB capaz de gerar figuras simples, enquanto ao mesmo tempo permite um estilo de plotagem orientado a objetos mais poderoso, se desejado. Focaremos na interface mais simples por enquanto.

Como exemplo, o código abaixo continua a seção anterior e plota a onda senoidal que geramos. Este código primeiro plota o sinal `sine_wave` em sua totalidade (ou seja, de 0 a T_{max}), e depois restringe a janela de visualização aos primeiros 10% do seu comprimento, para que possamos observar os detalhes da onda senoidal.

(Note que isso não é muito eficiente computacionalmente; tente modificar o código para que ele plote apenas um número limitado de amostras desde o início!)

```
plt.figure()                        # abre uma nova figura usando Pyplot
plt.plot(t, x_seno)                 # plota a onda senoidal (eixo y) vs tempo (
   eixo x)
plt.xlabel('Tempo [s]')          # adiciona um rotulo no eixo 'x'
plt.ylabel('Amplitude')
plt.xlim([0, t.max()*0.1])       # ajusta os valores min e max do eixo 'x'
plt.ylim([-A*1.1, A*1.1])        # ajusta os valores min e max do eixo 'y'

# adiciona um titulo mostrando a frequencia da onda
plt.title("Onda senoidal, {:.2f} Hz".format(freq))
```

1.3 Operações com Sinais

Sinais podem ser combinados realizando operações como soma ou subtração. Em Python, isso significa realizar operações em arrays `Numpy`, que gerealmente são operações elemento a elemento.

Tarefas

- Sinal 1: Gere uma onda senoidal em uma frequência fundamental de sua escolha, de duração T_{max} segundos, amostrada a $fs = 44100$ Hz, com uma amplitude arbitrária A;
- Sinal 2: Gere um sinal de ruído branco com a mesma duração e frequência de amostragem do Sinal 1;
- Realize as seguintes operações nos sinais gerados anteriormente:
  - Soma dos dois sinais (e.g. $x[n] + y[n]$);
  - Multiplicação dos dois sinais (e.g. $x[n] \times y[n]$);
  - Raiz quadrada de cada sinal (e.g. $\sqrt{x[n]}$);
  - Elevar as amostras de cada sinal ao quadrado (e.g. $(x[n])^2$).
- Plote os sinais resultantes e verifique se eles concordam com o que você esperaria.

2 Sistemas

2.1 Sistemas Variantes/Invariantes no Tempo

O objetivo deste exercício é verificar se o sistema dado na função `sistema1` (fornecida no módulo `tutorial1_funcoes.py` no Github) é invariante no tempo ou não. A sintaxe para a função é:

```
y = Tutorial1.sistema1(x)
```

onde `x` é o sinal de entrada do sistema, e `y` é o sinal de saída do sistema.

Tarefas

1. Crie um sinal de impulso de amplitude unitária `x_1` com comprimento total de `N = 100` amostras, e uma segunda versão atrasada no tempo `x_2` com o mesmo comprimento da primeira, mas atrasada em `N_0 = 50` amostras;

2. Use os impulsos unitários e analise seus respectivos sinais de saída para determinar se o sistema `Tutorial1.system1(x)` é invariante ou variante no tempo.

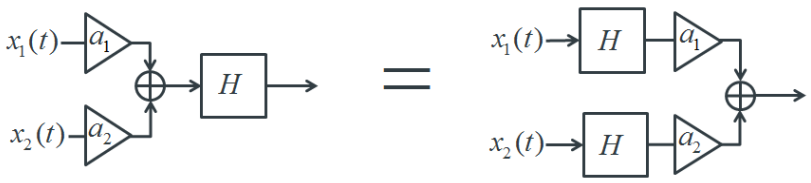
2.2 Sistemas Não Lineares

2.2.1 Revisão

Um sistema H é dito linear se ele satisfaz o princípio da superposição – ou seja, satisfaz aditividade e homogeneidade:

$$H(a_1x_1[n] + a_2x_2[n]) = a_1H(x_1[n]) + a_2H(x_2[n]).$$

Diagrama de blocos:



Tarefas

1. Use um conjunto de sinais para descobrir se o sistema `Tutorial1.sistema(x)` é linear ou não.

3 Energia e Potência de Sinais

3.1 Revisão

Para sinais de tempo contínuo, a energia total E e a potência média P de um sinal são definidas da seguinte forma:

$$\begin{aligned} E &= \lim_{T\to\infty} \int_{-T}^{T} |x(t)|^2 dt \leq \infty \\ P &= \lim_{T\to\infty} \frac{1}{2T} \int_{-T}^{T} |x(t)|^2 dt \end{aligned}$$

Como em Python estamos limitados a sinais de tempo discreto, as definições precisam ser ajustadas adequadamente:

$$E = \lim_{N\to\infty} \sum_{-N}^{N} |x[n]|^2 \leq \infty$$

$$P = \lim_{N\to\infty} \frac{1}{2N+1} \sum_{-N}^{N} |x[n]|^2$$

3.2 Tarefas

1 Calcule a energia total e a potência média do seguinte sinal analiticamente:

$$x(t) = A\sin(2\pi \cdot 53 \cdot t)$$

Dica: $\sin^2(x) = \frac{1}{2}(1 - \cos(2x))$

Carregue os dados contidos no arquivo `EnergiaPotencia.mat` no seu diretório de trabalho. Você precisará ler o arquivo *.mat* usando a função `scipy.io.loadmat`, que retorna um dicionário, e ler as entradas do dicionário `'x1'` e `'x2'` para acessar os dois vetores de dados contidos nele:

```
# Use o modulo 'loadmat' para carregar o arquivo MAT como um dicionario
matfile = loadmat('EnergiaPotencia')

# leia as chaves no dicionario para obter os dados
mat_signal1 = matfile['x1']
mat_signal2 = matfile['x2']
```

2 Determine a energia total e a potência média do sinal dado em `mat_signal1`.

3 Determine a energia total e a potência média do sinal em `mat_signal2`, assumindo que o vetor contém apenas um período de um sinal periódico.
CUIDADO: Atenção ao limite!

4 Escrevendo Sinais em Arquivos `.WAV`

No Processamento Digital de Sinais, tratamos sinais de áudio manipulando suas representações numéricas em um computador digital, que normalmente armazena os dados como vetores de números de ponto flutuante. No entanto, muitas vezes é muito ilustrativo ouvir esses sinais, e portanto é necessário gerar um sinal acústico real.

Uma maneira de gerar sinais de áudio é salvá-los como arquivos Wave, compatível com diversos programas e aparelhos de som. Este é um padrão de formato de arquivo de áudio usado para armazenar um fluxo de bits de áudio, normalmente áudio não comprimido codificado em formato de modulação por código de pulso linear (*Linear Pulse Code Modulation*, ou LPCM), similarmente ao método usado para armazenar música em CDs. Tais arquivos são geralmente reconhecidos por sua extensão `.wav`. Uma versão muito popular para arquivos Wave armazena cada amostra como um valor inteiro de `N_bits` com sinal (positivo/negativo) – o primeiro bit contém o sinal e não é usado para armazenar dados de amplitude. Desta forma, as amplitudes máxima e mínima que uma amostra Wave pode assumir são dadas por $\pm(2^{(N\_bits-1)} - 1)$; por exemplo, áudio de CD utiliza 16 bits por amostra.

No entanto, dados de áudio são frequentemente manipulados usando representação de ponto flutuante (como arrays Numpy), e geralmente são normalizados para ter suas amplitudes máxima e

mínima permitidas iguais a ±1.0: valores acima de +1.0 e abaixo de −1.0 são armazenados como ±1.0, e o sinal torna-se "clipado" (limitado). Portanto, para converter um array Numpy em um arquivo Wave com qualidade de CD é necessário mudar seu tipo de dado de `float` para `int16` e renormalizar seus valores de amplitude de ±1.0 (ponto flutuante) para $\pm(2^{(N\_bits-1)} - 1)$ (inteiro).

O módulo `scipy.io.wavfile` permite ler e escrever arquivos Wave, uma vez que os dados foram convertidos para o tipo correto. Vamos agora escrever um arquivo contendo a onda senoidal que geramos antes como um Wave codificado em 16 bits.

IMPORTANTE: Note que esta onda senoidal possui a amplitude máxima permitida para um sinal Wave, e portanto o som reproduzido será **MUITO ALTO!** Lembre-se de diminuir o volume do seu sistema de reprodução de som (fones de ouvido ou alto-falantes do computador) antes de reproduzir este arquivo, e depois aumente o volume até atingir um nível confortável.

```
N_bits = 16          # Numero de bits

wav_pico = 2**(N_bits-1) - 1     # Amplitude de pico da amostra wav

# converte o tipo da variavel para int de 16 bits e normaliza as amplitudes
   para 'wav_peak'
wav_amostras = np.int16(x_seno * wav_pico)

# Escreve um arquivo chamado 'seno.wav', na frequencia de amostragem 'fs',
# contendo os dados dentro de 'wav_amostras'
wavfile.write('seno.wav', fs, wav_amostras)
```

O mesmo arquivo pode ser lido de volta, convertido de volta para um array Numpy de 64 bits (normalizado para ±1), e reproduzido usando `sounddevice`.

Tente reproduzir a onda senoidal e veja se ela soa como você esperaria. Tente também gerar um arquivo Wave com o ruído branco Gaussiano (lembre-se de limitar sua amplitude a ±1!) e ouça-o também.

```
# Le o arquivo 'seno.wav'
fs_2, wav_amostras_2 = wavfile.read('seno.wav')

# Normaliza a amplitude para +/- 1 e muda o tipo de dado para float 64 bits
x_seno_2 = np.float64(wav_amostras_2 / wav_pico)

# Reproduz valores lidos de 'seno.wav', CUIDADO - Verifique o volume!!!
sd.play(x_seno_2, fs_2)
```