Processamento Digital de Sinais e Aplicações em Acústica
Tutorial 1 - Revisão de Python

https://github.com/fchirono/Aulas_PDS_Acustica

Sumário

1 Instalação e Configuração

Este Tutorial inicial visa fornecer uma breve revisão de conceitos de programação usando a linguagem Python versão 3. Utilizaremos a distribuição *Miniforge*, um instalador e gerenciador de pacotes para linguagem Python de uso livre e gratuito, que pode ser facilmente baixado da internet.

1.1 Miniforge

Para instalar o Miniforge:

- Navegue para o endereço `https://conda-forge.org/download/` e selecione o instalador para o seu Sistema Operacional - este tutorial descreve os passos necessários para instalar o programa em um ambiente Windows[1];

[1] Para outros Sistemas Operacionais, veja as instruções disponíveis em `https://github.com/conda-forge/miniforge` (em inglês).

- Execute o instalador e siga as instruções na tela:
  - Recomenda-se instalar o Miniforge apenas para o seu usuário - selecione a opção *"Just Me"*;
  - Recomenda-se permitir o Miniforge criar atalhos para os pacotes compatíveis - selecione a opção *"Create shortcuts (supported packages only)"*.

Você agora possui o programa *Miniforge Prompt* no seu computador, um prompt de comando específico para criar e manejar os ambientes virtuais (*"virtual environments"*) e pacotes Python. Este prompt utiliza o programa `conda` para instalar e gerenciar pacotes e ambientes através da linha de comando.

- No Windows, os comandos `conda` devem ser executados no *Miniforge Prompt*. Este programa está disponível através de um atalho na sua Área de Trabalho, ou na pasta "Miniforge3" do menu Iniciar;

- Em sistemas Unix, os comandos `conda` podem ser executados diretamente no terminal.

1.2 Ambientes Virtuais

Ao iniciar o programa *Miniforge Prompt*, você verá uma versão modificada do Prompt de Comando do Windows. Este prompt inicia em um ambiente chamado *base* - note que este nome aparece entre parênteses à esquerda da linha de comando.

Ambientes virtuais (*"virtual environments"*) são "espaços" isolados para instalação de pacotes Python, incluindo o gerenciamento de versões e dependências. Ambientes virtuais não interferem entre si, o que evita conflitos entre diferentes pacotes ou versões, mesmo quando instalados em um mesmo computador. Ambientes virtuais também são fáceis de serem criados, apagados e recriados, podendo ser tratados como "descartáveis": caso ocorra algum problema com o seu ambiente virtual, simplesmente apague-o e crie um novo.

Recomendação: não instale pacotes ou modifique o ambiente "base", e crie um ambiente novo para cada projeto!

1.2.1 Criando um ambiente virtual novo

Vamos criar um ambiente virtual novo chamado `tutorial_pds` para este curso. Abra o *Miniforge Prompt* e entre o seguinte comando:

```
conda create -n tutorial_pds
```

Após criado, o novo ambiente virtual pode ser ativado através do comando:

```
conda activate tutorial_pds
```

Vamos agora instalar alguns pacotes básicos, em suas versões mais recentes. Certifique-se que o ambiente `tutorial_pds` encontra-se ativo, e entre o seguinte comando:

```
conda install python numpy scipy matplotlib spyder python-sounddevice
```

Ao final da instalação, você terá um ambiente virtual Python propício para Processamento Digital de Sinais. Para mais instruções de como criar e manejar ambientes virtuais e pacotes Python usando `conda`, veja o conda cheatsheet (em inglês).

1.3 A IDE Spyder

Este curso utiliza o Ambiente de Desenvolvimento Integrado (*Integrated Development Environment* - IDE) *Spyder*. Após a instalação descrita na seção anterior, a IDE Spyder estará disponível através de atalhos na Área de Trabalho e no Menu Iniciar, e os nomes dos atalhos fazem referência ao ambiente virtual no qual o Spyder foi instalado. Para abrir o Spyder dentro deste ambiente virtual e ter acesso aos pacotes recém instalados, simplesmente utilize os atalhos.

Por padrão, o IDE Spyder contém um editor de arquivos no lado esquerdo, um console interativo IPython (*Interactive Python*) no lado inferior direito, e um painel multiuso (*Help / Variable Explorer / Debugger / Plots / Files*) no lado superior direito. O diretório onde o Spyder irá executar os programas pode ser visto (e modificado) através da barra de diretórios no canto superior direito. Uma demonstração das principais funcionalidades do Spyder está disponível dentro do próprio programa, através do menu *Help - Interactive Tour*, e um tutorial mais detalhado pode ser encontrado no menu *Help - Built-in Tutorial*.

Para este curso, você frequentemente precisará baixar e usar funções especiais e arquivos de dados em seus códigos. Recomenda-se salvar estes arquivos no seu computador dentro de uma pasta específica para cada tutorial, e selecionar esta pasta como diretório de trabalho do Spyder através da barra de diretórios no canto superior direito. Certifique-se de que esses arquivos estejam "visíveis" para o seu programa Python atual, configurando o diretório de trabalho do Spyder para a mesma pasta onde eles estão localizados. Lembre-se de sempre verificar se o diretório de trabalho atual do Spyder, exibido no canto superior direito da tela do Spyder, está definido para o local correto; caso contrário, você pode mudar o diretório através da própria barra de diretório ou através do painel *Files*.

1.3.1 Configurando o Spyder

Antes de começar o módulo, recomenda-se que você ajuste algumas opções no Spyder para uma experiência mais interativa e amigável. No Spyder, selecione *Tools - Preferences* e:

- Em *IPython console - Plotting - Graphics backend*, selecione *Automatic* como seu backend gráfico padrão; isso deve permitir que figuras recém criadas apareçam em janelas separadas (ao invés de serem incorporadas no painel *Plots*), permitindo que você as inspecione manualmente usando as ferramentas de zoom;

- Em *Help*, selecione as conexões automáticas para o Editor e para o console IPython; isso permitirá que o painel *Help* (localizado logo acima do console) abra automaticamente a documentação de uma função específica que você começar a escrever no Editor ou no console. A documentação das funções geralmente inclui uma breve descrição da função, os tipos de variáveis de entrada e saída que ela aceita e retorna, e às vezes alguns exemplos de seu uso.

2 Conceitos Fundamentais de Programação em Python

Este curso assume familiaridade com a linguagem de programação Python, e esta seção serve como revisão destes conceitos. Caso você não tenha familiaridade com esta linguagem de programação, há uma enorme quantidade de materiais e aulas sobre programação em Python disponíveis livremente na internet - algumas sugestões encontram-se ao final deste documento. Esta revisão é fortemente inspirada na *Scientific Python Lecture Notes*.

Esta seção contém uma lista de comandos Python para você digitar e executar diretamente no console IPython, linha a linha, e cobrem conceitos fundamentais de programação em Python que utilizaremos neste módulo. Antes de executar os comandos, use seus conhecimentos de linguagem Python para antecipar e explicar o resultado de cada comando.

2.1 Tipos de Variáveis

```
print("Ola, mundo!")

# variavel inteira
a = 3
b = 2*a
type(b)
b

a*b
a**b

# o tipo da variavel pode mudar durante a execucao!

# variavel tipo string
b = "Ola"
type(b)
print(b)

b + b
2*b
```

2.2 Tipos de Variáveis Numéricas

```
a = 4
type(a)

# variavel tipo ponto flutuante - float
b = 2.1
type(b)

# conversao de tipo
c = float(1)
type(c)

# conversao para inteiro nao e o mesmo que arredondamento!
int(2.1)
round(2.1)

int(2.9)
round(2.9)

# variavel tipo numero complexo (partes real e imaginaria)
d = 1.5 + 0.5j
type(d)
d.real
d.imag
```

A divisão inteira (`//`) e resto (`%`) são definidas através da expressão `x == (x//y)*y + (x%y)`.

```
a = 3
b = 2

```

```
a / b

# divisao inteira
a//b

# Se um operando for float, o resultado tambem e float!
float(a)//b

# operador resto
a % b
```

2.3 Listas e Tuplas

Listas são coleções ordenadas de objetos, que podem ser de tipos diferentes. Listas usam colchetes (`[...]`) para criação e indexação, a indexação começa no índice zero, e índices negativos permitem contar de trás para frente a partir do final da lista.

```
L = ["vermelho", "azul", "verde", "preto", 1001.099]

L[1]
L[0]

L[-1]
L[-4]

# elementos da lista podem ser alterados
L[3] = 1000
L
```

O fatiamento resulta em sublistas de elementos regularmente espaçados: `[inicio:fim]` retorna todos os elementos com índices `inicio <= i < fim` (ou seja, `i` varia de `inicio` até `fim-1`, de modo que `fim - inicio` elementos são retornados - o primeiro índice sempre está incluído, e o fim sempre é excluído). Também é possível adicionar um tamanho de passo usando a sintaxe `[inicio:fim:passo]`.

```
L[0:3]

L[2:3]

L[0:4:2]

# do inicio ao fim
L[:]

# do inicio ao fim, em ordem reversa (passo de -1)
L[::-1]
```

Tuplas são construções semelhantes às listas, mas usam parênteses na sua criação e são imutáveis. Tanto listas como tuplas usam colchetes na indexação.

```
t3 = ("Joao", "Joana", 999)

t3[2]

# tuplas nao podem ser alteradas
```

```
t3[1] = 10

# tuplas de elemento unico e tuplas vazias sao validas
t1 = (10,)
t0 = ()
```

2.4 Texto / Strings

Texto (também chamados de *strings*) são "listas de caracteres" e são indexadas como listas. Strings podem ser criadas usando aspas simples (`'...'`) ou aspas duplas (`"..."`).

```
a = "Ola, mundo!"
a[3:6]

a[2:10:2]

a[::3]
```

A formatação de strings permite a criação de strings usando outras variáveis previamente definidas.

```
a = 1
b = 0.1
name = "Joao"
print("Um inteiro: {}; um float: {}; outra string: {}".format(a, b, name))

# use tres casas decimais para float dentro da string
pi = 3.14159265359
"O numero pi pode ser aproximado como {:.3f}".format(pi)

# notacao exponencial com duas casas decimais
"O numero 123456 pode ser aproximado como {:.2e}".format(123456)
```

2.5 Controle de Fluxo

2.5.1 Instruções `if`

Instruções `if` permitem a execução condicional de código quando uma dada condição é verdadeira; testes alternativos podem ser adicionados usando `elif`, e exceções a todos os testes são tratadas usando `else`. Variáveis booleanas (verdadeiro/falso) são definidas com iniciais maiúsculas: (`True` / `False`).

```
condition = True
if condition:
    print("E verdade!")

# um sinal de igual significa atribuicao de variavel;
# dois sinais de igual significa comparacao!
a = 5

if (2+2 == a):
    print("E quatro!")
elif (a < 4) and (a > 0):       # condicoes podem ser concatenadas
    print("E menor que quatro, mas positivo!")
elif a < 0:
    print("E negativo!")
```

```
else:
    print("E maior que quatro!")
```

2.5.2 Instruções `for`

Instruções `for` permitem a execução repetida de um dado trecho de código enquanto itera sobre um contador ou uma sequência.

```
# 'range' tambem segue a notacao (start, stop, step)
for i in range(4, 10, 2):
    print(i)

# tambem pode-se iterar sobre elementos de uma sequencia
for palavra in ("legal", "facil", "divertido"):
    print("PDS e {}".format(palavra))
```

2.6 Funções

Funções permitem o encapsulamento e abstração de uma tarefa, e estimulam a reutilizar código em vez de reescrever código. Chamadas de funções Python usam parênteses para os argumentos da função; funções retornam `None` por padrão, a menos que a função explicitamente declare uma ou mais variáveis de retorno.

```
def dobrar_valor(x):
    """Frase concisa de uma linha descrevendo a funcao.

    Documentacao completa da funcao, que pode conter multiplos
    paragrafos."""

    return x * 2

dobrar_valor?

dobrar_valor(3)

dobrar_valor()
```

Funções Python podem ter argumentos de valor padrão para quando uma função é chamada sem um ou mais de seus argumentos.

```
def elevar_potencia(x = 2, y = 2):
    return x**y

elevar_potencia(3, 3)

elevar_potencia(3)

elevar_potencia()
```

2.7 Numpy

Numpy é o módulo *Python numérico*. Este módulo define um tipo próprio de *array* homogêneo (todos os elementos são do mesmo tipo) e multidimensional, que é útil para cálculos numéricos rápidos. Arrays unidimensionais são equivalentes a vetores (no sentido matemático), e seus elementos podem

ser indexados usando um único índice; arrays bidimensionais são equivalentes a matrizes (no sentido matemático), e seus elementos podem ser indexados usando dois índices; etc. O módulo Numpy deve ser importado antes de ser usado.

```
import numpy as np

# arrays Numpy sao criadas usando listas (e listas de listas) como argumentos
    de entrada
a = np.array([0, 1, 2, 3])        # array 1D
b = np.array([[0, 1, 2], [3, 4, 5]])    # array 2D (2x3)

a
b

# "shape" (formato) do array: tupla contendo numero de elementos em cada
   dimensao
a.shape
b.shape

# verificar tipo dos elementos dentro do array
a.dtype

# indexacao e igual as listas
a[1]
b[0, 1]

b[1]

# fatiamento funciona independentemente em cada dimensao
b[0, :]
b[:, 0]
```

Algumas funções comuns para criar arrays são mostradas abaixo.

```
# arange: (inicio, fim (nao incluido), tamanho do passo)
b = np.arange(1, 9, 2)
b

# linspace: (inicio, fim, numero de pontos)
# ponto final pode ser incluido (True - padrao) ou nao (False)
c = np.linspace(0, 1, 6)
c

c = np.linspace(0, 1, 6, endpoint=False)
c

# zeros: cria um array populado com zeros
# note que a tupla (2, 3) define a dimensao do array e define um unico
   argumento de entrada!
z = np.zeros((2, 3))
z

# pode-se criar uma matriz de 'zeros complexos'
z_complex = np.zeros((3, 3), dtype='complex')
z_complex.dtype
z_complex
```

```
# ones: comportamento similar a 'zeros'
o = np.ones((3, 5))
o

# criar array de numeros aleatorios (distribuicao Gaussiana)
r = np.random.randn(10)
r
```

Arrays podem ser lidos e modificados usando seus índices; cada dimensão requer um índice separado.

```
formato_array = (4, 4)
A = np.zeros(formato_array)
A

A[1, 3] = np.pi
A

b = np.arange(5, 9)
A[2, :] = b
A

d = np.array([1, 2, 3])

# soma com conteudo anterior (em vez de sobrescrever)
A[1:, 0] += d
A
```

Note que a função `np.arange` NÃO é recomendado para passos fracionais, podendo gerar erros devido à precisão numérica. Se intervalos fracionais forem necessários, recomenda-se fortemente usar a função `np.linspace`.

```
# quantos/quais elementos estarao neste array?
a = np.arange(0, 1, 1./10)
a
a.size
a[-1]

# quantos/quais elementos estarao neste array?
b = np.arange(0, 1, 1./49)
b
b.size
b[-1]

# quantos/quais elementos estarao neste array?
c = np.linspace(0, 1, 10)
c
c.size
c[-1]

# quantos/quais elementos estarao neste array?
d = np.linspace(0, 1, 49)
d
d.size
d[-1]
```

Operações matemáticas com arrays são quase sempre realizadas elemento a elemento:

```
x = np.arange(5)

x+2

x*3

x**3

np.sqrt(x)

x*x

np.exp(x)

# logaritmo base 10
np.log10(x)
```

Algumas operações não são realizadas elemento a elemento: entre elas, incluímos o produto escalar (para multiplicação de matrizes quando aplicado a arrays 2D, e produto interno de vetores quando aplicado a arrays 1D) e o produto vetorial (também conhecido como produto externo). O desempenho dessas operações em arrays N-D é mais complexo, tenha cuidado com resultados inesperados.

```
x = np.arange(3)
y = np.arange(4, 7)
A = np.arange(9).reshape((3, 3))

# produto escalar / produto interno
w = np.dot(x, y)
w.shape
w
x @ y

# produto matriz-vetor
np.dot(A, x)

# O caractere '@' pode ser usado para
# produto de matrizes / produto interno (no sentido matematico)
np.dot(A, x) == A @ x

# produto vetorial / produto externo
z = np.outer(x, y)
z.shape
z
```

Arrays Numpy também podem ser concatenados, desde que suas respectivas formas coincidam. Existem outras funções e métodos para concatenar arrays, tente pesquisá-los.

```
x = np.array([[0., 0.1, 0.2], [0.3, 0.4, 0.5]])
y = np.array([[1., 1.1, 1.2], [1.3, 1.4, 1.5]])

x.shape
y.shape

# argumento da funcao e uma tupla '(x, y)'
z1 = np.concatenate((x, y))            # eixo 'zero' por padrao
```

```
z1.shape
z1

z2 = np.concatenate((x, y), axis=1)
z2.shape
z2
```

2.8 Matplotlib / Pyplot

O módulo Matplotlib é um módulo muito poderoso para plotar figuras, mas pode ser trabalhoso de usar. Recomendamos usar a interface `pyplot`, que adota um estilo semelhante ao MATLAB e simplifica alguns dos comandos mais comuns. Ele também deve ser importado antes de ser usado.

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

# dados 1D
x = np.linspace(-np.pi, np.pi, 256, endpoint=True)
y1 = np.cos(x)
y2 = np.sin(x)

plt.figure()
plt.plot(x, y1, label="y1")
plt.plot(x, y2, 'ro', label="y2")

plt.xlabel("Rotulo para o eixo x")
plt.ylabel("Rotulo para o eixo y")
plt.title("Titulo - Grafico Simples")
plt.legend()
plt.savefig("nome_fig.png")
```

2.9 Gravar e Reproduzir Sinais com `sounddevice`

Agora vamos instruí-lo sobre como reproduzir e gravar arquivos de som em seu computador usando Python. Usaremos o módulo `sounddevice`, que deve ser importado antes de ser usado, e o módulo `numpy` para calcular sinais em função do tempo:

```
import sounddevice as sd
import numpy as np
```

IMPORTANTE: O fluxo de dados de áudio será enviado diretamente para a saída de áudio do computador pelo `sounddevice`, e portanto o volume pode estar **MUITO ALTO**! Lembre-se de diminuir o volume do seu sistema de som (fones de ouvido ou alto-falantes do computador) ao mínimo antes de reproduzir qualquer arquivo de áudio, e aumente o volume aos poucos até atingir um nível de audição confortável!

2.9.1 Reproduzindo Sinais

Agora, vamos criar um sinal usando Numpy e reproduzi-lo através da placa de som do computador usando `sounddevice`:

```
# frequencia de amostragem (em Hz)
fs = 44100
```

```
# intervalo de amostragem (em segundos)
dt = 1/fs

# define a duracao total do sinal
T_total = 1.

# numero de amostras no sinal
N_amostras = int(T_total * fs)

# criar um vetor de amostras no tempo usando o comando 'np.linspace'
t = np.linspace(0, T_total-dt, N_amostras)

# cria uma onda senoidal de 500 Hz
f0 = 500
x_seno = np.sin(2*np.pi*f0*t)

# Atenuar o sinal de teste para evitar 'clipping' e erros numericos
ganho = 0.5

# reproduz o sinal de teste na saida de som padrao do computador
sd.play(x_seno*ganho, fs)
```

2.9.2 Gravando Sinais

O módulo `sounddevice` também permite a gravação de dados de áudio. Este próximo exemplo demonstrará como isso pode ser feito gravando alguns segundos de dados, plotando a forma de onda gravada e depois reproduzindo-a para você.

```
# gravar 2 s de dados e salvar no array 'y'
T_duracao = 2
N_amostras = int(T_duracao * fs)

# grava 2 s de audio atravez da entrada de som padrao do computador
# (experimente estalar os dedos ou assobiar)
y = sd.rec(N_amostras, fs, channels=1, blocking=True)

# criar vetor de amostras de tempo
t2 = np.linspace(0, T_duracao-dt, N_amostras)

# plotar vetor 'y'
plt.figure()
plt.plot(t2, y)
plt.xlabel("Tempo [s]")
plt.ylabel("Amplitude")

# reproduzir o arquivo gravado 'y'
sd.play(y, fs)
```

3 Referências

Existe uma ampla variedade de tutoriais e fontes sobre programação em Python na internet, e espera-se que você irá pesquisar mais informações e aprender conceitos novos por conta própria. Algumas referências:

- Introdução à Linguagem Python para Ciências Computacionais e Engenharia: Livro aberto interativo, criado pelo Prof. Hans Fanghor, Univ. de Southampton, Reino Unido, focado em programação científica e engenharia (tradução em português e versão interativa online pelo Prof. Gustavo Oliveira, UFPB)
- Tutorial Oficial da linguagem Python (em português)
- Pense em Python (2a Edição): livro aberto criado por Allen B. Downey, focado em ensinar programação Python para principiantes (tradução em português pela Editora Novatec, disponível em HTML e impresso)
- Programming with Python (Software Carpentry): curso aberto de programação em Python focado em programação científica e análise de dados (em inglês)
- Tutorial `pyplot` (Matplotlib): Introdução à interface `pyplot`, do pacote `matplotlib`, para criar figuras
- Scientific Python Lectures: Notas de aula de Python para uso científico (em inglês)