Travaux Pratiques - OASIS SI101

Introduction à Python pour OASIS et SLI

1 Préliminaires : utilisation de python pour OASIS

Python est langage de programmation que nous utiliserons en mode interactif dans l'interface spyder.

On peut exécuter une commande soit en la tapant directement dans le mode interactif soit en exécutant un script qui est une suite de commandes qui se trouvent dans un fichier dont l'extension est "**.py**". Le mieux est de taper les commandes dans l'éditeur de spyder, sélectionner puis F9 pour exécuter les commandes sélectionnées. Cela permet de revenir sur les commandes et de les modifier en cas d'erreur. On peut aussi lancer toute une cellule avec Ctrl+Entrée.

les données pour le TP : Vous pouvez **soit** charger les données depuis le site pédagogique et décompresser dans un répertoire de votre choix **soit** faire ce qui suit dans un terminal

```
~ladjal/OASIS/charge_donnees_OASIS.sh TP1
```

cela va créer un répertoire OASIS_TP1 dans votre répertoire racine. Vous pouvez le déplacer où vous voulez.

Dans ce répertoire se trouve un fichier scratch_TP1_OASIS.py dans lequel la plupart des commandes sont déjà écrites.

lancer l'environnement spyder [1]

```
~ladjal/OASIS/spyderOASIS
```

Dans spyder ouvrez le fichier scratch_TP1_OASIS.py et exécutez la section de modules à importer.

Ce TP a pour objectif de prendre en main l'utilisation de python pour OASIS et d'effectuer des opérations de manipulation simple sur les SLI (Systèmes Linéaires et Invariants).

Tout le TP peut se faire en mode interactif, python permettant de définir des fonctions en mode interactif. Pendant la partie 1, vous êtes encouragés à taper les commandes et à en comprendre le résultat. En mode tutoriel le résultat d'une commande est affiché. Exemple

```
3+4; # Vous n'êtes pas obligés de taper les commentaires.
3+4  # ou comment utiliser 10^9 opérations par seconde pour calculer 7
```

1.1 Les bases

Sous python un package indispensable est le package numpy abrégé en np dans ce qui suit (**import numpy as np** et cette commande est à taper dans chaque nouvelle console python ouverte. Chaque tableaux numpy peut avoir une ou plusieurs dimensions. Les matrices seront des tableaux de dimension 2. Les vecteurs de dimension 1. Cependant, pour traiter la multiplication matrice vecteur il est utile de considérer un vecteur comme un tableaux à deux dimension dont l'une vaut 1 (suivant si l'on veut une colonne ou une ligne). Il seront par défaut remplis de valeurs de type float64.

1.2 Générer une matrice

Il y a divers moyens de générer une matrice ou un vecteur

— **np.zeros((m,n))** : Génère une matrice de taille $m \times n$ pleine de zéros.

— **np.ones((m,n))** : Cette fois-ci la matrice est pleine de 1.

1. C'est juste un spyder avec le bon environnement pour ce TP, bien sûr, en copiant les données et en installant les bon packages vous pouvez aussi executer ce TP sur n'importe quelle machine.

— **np.arange(a,c,b)** : Génère un vecteur ligne commençant à a et incrémenté de b tant que l'on est encore strictement plus petit que c. Si on tape (a, c) l'incrément par défaut est 1. Essayez

```
np.arange(0,7,3)
np.arange(1,7)
np.arange(0,1.01,0.1)
np.arange(0,1,0.1)
np.arange(5,0,-1)
```

— Entrer directement les valeurs entre crochets :

```
[1, 2 ,3] # ceci est une liste
np.asarray([1,2,3]) # ceci un tableau numpy
np.asarray([1,2,3]).reshape((1,3)) # tableau numpy sous forme de vecteur ligne
np.asarray([1,2,3]).reshape((1,-1)) # idem ci-dessus
np.asarray([[1,2],[3,4]])
np.asarray([1,2,3,4]).reshape((2,2)) # idem ci-dessus
np.asarray([1,2,3,4]).reshape((-1,2)) # idem ci-dessus
```

1.3 Opérations sur les tableaux

— **A+B** : Renvoie la somme des deux tableaux qui doivent être de même taille. **Exception** : Si A ou B est un scalaire, il est ajouté à toutes les entrées de l'autre tableau. L'opération "-" fonctionne de la même manière pour la soustraction

```
np.ones((2,2))+np.ones((2,2))
np.ones((2,2))+1 # on ajoute 1 à toutes les entrées.
```

— **A@B** : Renvoie la multiplication des matrices [2] A et B et doivent avoir des tailles respectives de $m \times n$ et $n \times l$. Le résultat est de taille $m \times l$. Notez que si $m = l = 1$ alors cette opération est un produit scalaire entre le vecteur ligne A et le vecteur colonne B.

```
A=np.asarray([[1,2],[ 3 ,4]])
B=np.asarray([[-2,1],[ 1.5 ,-0.5]])
A@B
B@A
A@np.asarray([[1, 0],[ 0 ,1]])
np.arange(1,20)@np.arange(1,20)  # Bien que arange renvoie un tableau à une seule
                                  # dimension, un vecteur prend la forme nécessaire
                                  #pour permettre le produit matriciel
A@np.arange(3,5)
np.arange(3,5)@A
A.T   # A.T est la transposée de la matrice A
```

— **A*B** : Multiplication point par point. A et B doivent avoir la même taille. Même exception pour les scalaires. Comparez

```
A*B
A@B
```

— **Fonctions unaires** : Numpy dispose d'un grand nombre de fonctions usuelles prédéfinies telles que : sin, cos, exp, tan, log.... Les appliquer à un tableau signifie appliquer la fonction à chaque élément du tableau

```
i=np.complex(0,1) # creer le complexe i
pi=np.pi # donner un nom plus commode à pi
np.exp(i*pi)
idx=np.arange(0,11)
np.exp(2*i*pi*0.123*idx) # Onde de Fourier sur Z de fréquence 0.123,  entre 0 et 10
np.real(np.exp(2*i*pi*0.123*idx)) # Partie réelle de l'onde
plt.plot(np.real(np.exp(2*i*pi*0.123*idx))) # "plot" affiche son argument
```

2. Seulement valable pour les version de python supérieures à 3.5

```
    # voir plus bas pour plus de précisions sur plt.plot
```

— **sum()** : Renvoie la somme des éléments d'un tableau. Si on précise un axe, on obtient la somme suivant cet axe

```
np.ones(10).sum()
np.arange(1,101).sum()
np.ones((2,3)).sum(axis=0) # somme colonne par colonne
np.ones((2,3)).sum(axis=1) # somme ligne par ligne
(np.arange(1,10)**2).sum() # somme du carré des entiers entre 1 et 9
```

— **np.concatenate** : Concaténation, par défaut empile les colonnes. L'argument est un n-uplet des tableaux à concaténer.

```
np.concatenate( (np.ones( (2,3) ) , np.ones((2,3)) ) )
np.concatenate( (np.ones ((2,3)) , np.ones((2,3))), axis=1)
np.concatenate( (np.ones( (3,3) ) , np.ones((2,3)) ) )
np.concatenate( (np.ones( (3,3) ) , np.ones((2,3)) ), axis=1 ) # erreur
```

— **A.reshape(-1)** : Quelque soit la forme de A, renvoie un vecteur monodimensionnel reprenant tous les éléments de A. A est parcourue ligne par ligne. Utile pour s'assurer que deux vecteurs ont la même forme.

```
A=np.asarray([[1,2],[3,4]])
A.reshape(-1)
u=np.asarray([[1, 2, 2, 1]])  # matrice ligne
v=np.asarray([[1],[2],[3],[4]]) # matrice colonne
u*v # résultat inatendu
u.reshape(-1)*v.reshape(-1) #on est sûr de la forme c'est le produit point a point
```

— **x** A** : élève le scalaire x à la puissance chacune des entrées de A (fonctionne aussi si A est un scalaire)

```
(-1/2)**np.arange(0,11) # les puissances de -1/2 de 0 à 10
```

1.4 Les variables

Les variables sont créées lors de leur première affectation, attention aux effets de bord.

```
u=np.ones((2,2))
v=u
v[0,0]=7
u    #  u a été modifié par l'action sur v car v n'est qu'une copie de pointeur
v=u.copy() # voici la bonne manière de créer une copie indépendante de u
v[0,0]=8
u
```

Les indices des tableaux commencent à 0 : **h[0]** est le premier élément du vecteur **h**. **A[2,1]** est l'élément situé à la ligne numérotée 2 (troisième ligne) colonne numérotée 1 (deuxième colonne) de la matrice **A**. On peut aussi accéder à une collections de valeurs en indiquant une collection d'indices :

```
h=np.arange(0,10);
h[1:3]
h[1:10].sum() # somme les éléments numéro 1 à 9 de h
h[:-1].sum() #somme tous les éléments de  h sauf le dernier
h[0::2].sum() # h[0]+h[2]+...+h[plus grand indice pair possible]
h[::-1] # renvoie le vecteur h à l'envers
h[2:5]=np.ones(3) # remplace certaines entrées de h par des 1
```

2 Réaliser une convolution et écouter un écho

Dans cette partie vous allez réaliser une convolution entre deux suites. Vous appliquerez cela à un signal sonore pour simuler un écho et écouterez le résultat. Certes, python possède des packages capables de faire cela (scipy.signal), mais, pour vous exercer vous en réaliserez une vous même.

2.1 Création d'une fonction

On définit une fonction en python de la manière suivante

```
def mafonction(a,b,c):
    tmp=a+b
    toto=c-b
    titi=tmp/toto
    return titi
```

Le résultat renvoyé par cette fonction est $(a+b)/(c-b)$. Les variables internes (tmp, toto et titi) seront oubliées après l'appel de cette fonction. Pour OASIS, nous définirons toutes les fonctions dans un seul fichier de travail pour ne pas multiplier les fichiers, mais un gros projet demanderait plus d'organisation.

2.2 Modélisation en vue de la convolution

L'opération de convolution entre deux suites u et h donne une suite, que l'on va noter v définie par

$$v_n = \sum_{m\in\mathbb{Z}} h_m u_{n-m}$$

Sur un ordinateur, et dans python en particulier, on ne peut avoir que des suites à support fini. Ce que nous signifions par "opération de convolution" doit tenir compte de cette limitation. Quitte à décaler les suites, on peut toujours supposer qu'une suite à support fini a son support du type $\{0,\dots,N-1\}$ pour un certain entier N. Si deux suites ont leur support dans $\{0,\dots,N-1\}$ et $\{0,\dots,M-1\}$ respectivement, alors leur produit de convolution a son support dans $\{0,\dots,N+M-2\}$. Ainsi la convolution entre deux vecteurs de taille N et M sera un vecteur de taille $N+M-1$. Nous ferons la convention que, si un vecteur python $\mathbf{h}$ [3] représente une suite h alors le premier élément du vecteur $\mathbf{h}$ (i.e. $\mathbf{h[0]}$) est la valeur de h_0 et le support de la suite h est supposé être dans $\{0,\dots,\mathbf{len}(\mathbf{h})-1\}$.

2.3 Valeur en un point

Avec ces conventions, écrire une fonction qui prend en entrée deux vecteurs $\mathbf{h}$ et $\mathbf{u}$ et un entier n et renvoie la valeur de leur convolution en n (i.e. $(h*u)_n$). S'il n'y avait pas de problèmes de bord cela s'écrirait

```
def valconv(h,u,n):
    a=len(h) # pour un tableau numpy len(v)==v.shape[0]
    b=len(u)
    u=u.reshape(-1) #s'assurer que u et v son de la meme forme
    h=h.reshape(-1) #
    idx=np.arange(0,a); # les indices pour h
    return (h[idx]*u[n-idx]).sum()
```

3. à partir de maintenant, on écrira en gras des variables python et en italique leur représentation mathématique

Après avoir bien compris l'expression dans "(...).sum()" proposer une manière de remédier au problème des bords. On remarquera que pour que **h[m]** (m sera un élément du vecteur **idx**) intervienne dans la somme sans poser de problème d'indice, il faut et il suffit d'avoir

$$0 \leqslant m \leqslant a-1 \text{ pour que } \mathbf{h[m]} \text{ existe}$$
$$0 \leqslant n-m \leqslant b-1 \text{ pour que } \mathbf{u[n\text{-}m]} \text{ existe}$$

En déduire la formule correcte pour le vecteur **idx** en utilisant `min(x,y)` et `max(x,y)` qui renvoient le minimum et le maximum entre x et y.

Testez votre fonction avec

```
valconv(np.ones(3),np.ones(2),0) # doit renvoyer 1
valconv(np.ones(3),np.ones(2),2) # doit renvoyer 2
```

2.4 Convolution complète

Modifiez votre fonction pour qu'elle renvoie le vecteur entier de la convolution (on supprime le paramètre n). Pour cela on utilisera une boucle

```
out = np.zeros(a+b-1)
for n in range(a+b-1): #La boucle va être parcourue avec n parcourant le vecteur 0,1,..
    idx=... # Formule trouvée précédemment
    out[n]=   (h[idx]*u[n-idx]).sum()
return out
```

Testez votre fonction avec de petites suites dont vous calculerez la convolution à la main pour vérifier.

2.5 Ecoute d'un écho

Charger un son dans python à l'aide de la fonction

```
[x,Fe]=sf.read('piano.wav')
```

Cela va charger une variable **x** qui contient un son. La valeur de Fe est la fréquence d'échantillonnage. La commande suivante permet d'écouter un son :

```
play(x,Fe)
```

Que donne la commande suivante ?

```
play(x,Fe/2)
```

On peut modéliser un écho par le fait que le son s'ajoute à lui même avec un certain retard, si u représente le signal sonore original, le signal reçu sera du type

$$v_n = u_n + 0.8u_{n-t_1}$$

Ceci signifie que le son rebondit sur un obstacle, perd 20% de son amplitude et arrive retardé de t_1. On constate que cette modélisation revient à réaliser la convolution du son u avec une réponse impulsionnelle h définie par $h_0 = 1,\ h_{t_1} = 0.8$ (tous les autres termes de h sont nuls).

Créer le vecteur **h** correspondant à un retard approximatif de 0,01 seconde (on rappelle que le son utilisé ici est échantillonné à Fe échantillons par seconde). Convoler, avec votre programme, ce vecteur **h** avec le signal sonore **y** (stockez le résultat dans une variable).

À combien estimez-vous le temps de calcul pris par votre programme ?

Utilisez la commande

```
z=scipy.convolve(h,y);
```

Cette fonction est une fonction python qui fait ce que fait votre programme de convolution. Que constatez-vous sur le temps de calcul ?

Ecoutez le son **z**. (play(z,Fe))

Augmentez le temps de l'écho à 0,2 seconde, quel impression cela donne-t-il ? En termes de distance parcourue par le son, que représentent 0,2 et 0,01 secondes, cela vous aide-t-il à interpréter l'effet de l'écho ?

3 Fréquence en Hz et réduite

Générer une onde de fréquence 0,01 et longue de 10000 échantillons ($n \mapsto e^{2i\pi\nu n}$, ν est la fréquence). Écoutez sa partie réelle(np.real(u)) par

```
play(real(u),44100) # Le son est joué à 44100 échantillons par seconde
play(real(u),22050)# le son est joué à 22050 échantillons par seconde
```

Quelles fréquences en Hz (1Hz=1/seconde) percevez-vous dans les deux cas ?

On dit que la fréquence en Hz est la fréquence réelle (ce qui nécessite de connaître la fréquence d'échantillonnage) alors que la fréquence 0,01 est la fréquence réduite (qui est toujours, pour un signal défini sur $\mathbb{Z}$, entre $-1/2$ et $1/2$).

4 Filtre rejecteur

Dans la suite on utilise la commande python lfilter (qui doit être définie par lfilter=scipy.signal.lfilter).

Après avoir consulté l'aide par help(lfilter), dire comment utiliser lfilter pour effectuer une convolution d'un signal contre un RIF (réponse impulsionnelle finie).

Dans la suite nous allons voir comment utiliser lfilter pour effacer une fréquence parasite dans un son. Pour cela nous créons une version polluée de la variable **y** :

```
n=np.arange(len(y)); #le temps discret que dure le signal y
f0=1261;
yb=y+0.1*cos(2*pi*f0/Fe*n); #on ajoute une onde parasite
```

Écouter le son **yb** à la fréquence **Fe**. Quelle est la fréquence en Hz et fréquence réduite de l'onde parasite ?

Donner la TZ de la RI du filtrage qu'effectue la fonction rejette1 ? Écouter le résultat de ce filtrage sur le signal **yb**. Cela vous paraît-il satisfaisant ?

Donner la TZ de la RI du filtrage qu'effectue la fonction rejette2 ? Écouter le résultat sur le signal **yb**.

Pour une valeur de **rho** proche de 1, placer les zéros et les pôles des TZ précédentes dans le plan complexe. Utiliser ces positions pour tracer (approximativement) le module des deux TFtD correspondantes (sur l'intervalle [-1/2,1/2[). Conclure quand à la qualité comparée des deux filtrages.

Pourquoi **rho** (paramètre de rejette2) doit-il toujours être strictement plus petit que 1 ?

En transmettant à la fonction rejette2 une fréquence **f0=1267** et un paramètre **rho** très très proche de 1, conclure au compromis nécessaire dans le choix de **rho**.