TRAVAUX PRATIQUES DE GENIE INFORMATIQUE

1 • BUT DE LA SEANCE.

Ces deuxième et troisième séances ont pour but de vous faire avancer dans la programmation avec Python. Vous y découvrirez les fonctions, les procédures et la manipulation d'objets. Dans ces exercices il est très important de bien contrôler où vous sauvegardez vos fichiers. N'oubliez pas, de plus, les règles de bonne programmation vous imposant de créer un dossier de version chaque fois que vous modifiez quelque chose d'important dans votre programme. Vous devez créer au moins un dossier de version par séances, mais vous pouvez en faire plus. Essayer d'aller le plus loin possible.

2 • PROGRAMMES, PROCEDURES ET FONCTIONS.

2.1 - Programme.

Avec Python, un programme est simplement un fichier texte contenant une liste d'instruction. Ce fichier est considéré comme un exécutable Python a partir du moment où il possède l'extension `.py`. L'IDE Spyder permet de créer et d'écrire des programmes dans la fenêtre de gauche. Il suffit d'aller dans le menu "File" et choisir "New file" (ou de cliquer sur l'icône en forme de feuille en haut à gauche) pour créer un nouveau programme.

-> • Travail : mettez-vous dans un nouveau répertoire de travail, et crée votre premier programme et donnée lui le nom de `MonPremierProgramme` (menu "File" puis "Save as" ou en cliquant sur l'icône en forme de disquette en haut à gauche).

Dans ce programme, nous allons réaliser un filtre passe-bas à réponse impulsionnelle exponentielle. Si x_n est l'entrée du filtre et y_n est la sortie du filtre, l'équation récursive de ce filtre est simplement : $y_n = \alpha x_n + (1-\alpha) y_{n-1}$ avec $\alpha \in [0,1]$ *(c'est peut- être l'occasion de réviser vos cours sur le filtrage).*

Pour lancer le programme, il suffit de cliquer sur le triangle vert dans la barre des menus (vous pouvez aussi le lancer à partir du terminal en tapant "`python3 MonPremierProgramme.py`".

Vous allez générer un signal d'entrée qui sera une somme de trois sinusoïdes de fréquences différentes et d'un bruit gaussien de variance égale 0,3. Par exemple :

```
>>import numpy as np
>> delta_t = 0.01
>> nombre_echantillons = 1000
>> temps = np.arange(0,nombre_echantillons)*delta_t
>> signal = 1.9*np.sin(5*temps)+ 1.5*np.sin(11*temps)+1.7*np.sin(2.3*temps)
>> signal = signal + np.sqrt(0.3)*np.random.randn(np.size(temps))
```

Vous mettrez ensuite en oeuvre le filtrage de `x` en créant une boucle itérative. Affichez les signaux de sortie, regardez l'influence du paramètre α sur le signal de sortie. Par exemple si $\alpha = 1$ que se passe-t-il ?

-> - Conseil : créez des variables pour le facteur du filtre α, les périodes de vos sinusoïdes, le nombre

de données de vos signaux numériques, etc..
De façon générale, même si le langage Python ne vous oblige pas à ça, créez les variables que vous utilisez en début de programme.
Essayez d'utiliser le "debugger », fixez des points d'arrêt.

2.2 - Fonction.

Les fonctions permettent d'isoler une partie d'un programme et ainsi de clarifier la lecture du programme principal. Dans le langage Python les fonctions (ainsi que les procédures) sont déclarées et définies dans le même fichier que le programme principal dans une section introduite par le mot-clé `def`. Le corps de la fonction étant défini par un bloc d'instruction indenté et le résultat renvoyé à l'aide du mot-clé `return`. Ce qui n'est pas le cas pour les procédures qui utilise les effets de bord et qui ne revoie pas de résultats, mais modifie les mêmes variables que le programme principal ou fait un affichage par exemple.

```
>> def nom_fonction(liste des paramètres):
>>    bloc d'instructions
>>    return(résultat)

>> def nom_procedure(liste des paramètres):
>>    global variables # pour pouvoir y accéder depuis le programme principal
>>    bloc d'instructions
```

Ainsi une fonction réalisant le filtrage ci-dessus devrait être définie comme suit :

```
>> def FiltreExponentiel(entree, alpha):
>>    """ ici, vous mettez votre calcul qui permet
>>    de définir la sortie à partir de l'entrée.
>>    Par exemple """
>>    NombreEchantillon = len(entree)
>>    return (alpha * numpy.ones((1,n))
```

Ce n'est bien sûr pas le bon code. Pour appeler cette fonction (ou procédure) dans votre programme principal il suffit d'écrire par exemple dans ce programme principal : `FiltreExponentiel(signal, 0.4)`
Exercice : écrire la "fonction" de filtrage à l'aide d'une fonction puis à la l'aide d'une procédure. Explorer les possibilités offertes par les fonctions et les procédures. Voyez entre autres que les variables des fonctions sont locales (par exemple `NombreEchantillon` n'existe pas dans le programme principal à moins qu'il ait été défini par ailleurs).

2.3 » Fonctions à nombre d'arguments variables.

Il est possible de créer des fonctions dont le nombre d'arguments est variable. Dans ce cas il faut utiliser les symboles spéciaux `*args` (arguments autres que les mots clés) et `**kwargs` (arguments de mot-clé) pour transmettre des listes d'argument.

```
>> # fonction
>> def myFun(*args):
>>   for a in args:
>>         print(a)

>> # Implémentation
>> myFun('Hello', 'Welcome', 'to', 'GeeksforGeeks')
```

Vous pouvez utiliser la fonction `len(args)` pour connaitre le nombre d'arguments réellement passés lors de l'appel à la fonction. Concernent la sortie, il est toujours possible de retourner une liste de valeurs (tableau).

Utilisez l'aide (`help()`) de Spyder (ou internet), modifiez votre fonction `FiltreExponentiel` de façon à ce que la valeur par défaut de 0.4 soit attribuée à alpha lorsque la fonction n'est appelée qu'avec un argument.

2.4 - Modification de votre fonction.

(si vous travaillez lentement, ne faites pas cette question).

Vous devez faire une modification de votre fonction de filtrage exponentiel. Ce filtrage est causal et provoque donc un déphasage. Considérez cette modification comme une révision majeure de votre programme, donc créez un nouveau dossier de version.

La nouvelle fonction que vous devez créer réalise un filtrage non-causal séparable (c'est-à-dire que l'algorithme est composé d'un filtrage causal et d'un filtrage non-causal). La récursion causale est identique à la récursion causale précédente à savoir :

$y_n^C = \alpha x_n + (1-\alpha) y_{n-1}^C$ (y_n^C est la partie causale du filtre),

$y_n^A = \alpha x_n + (1-\alpha) y_{n+1}^C$ (y_n^A est la partie anti-causale du filtre), avec $\alpha \in [0,1]$,

et enfin $y_n = \frac{1}{2-\alpha} y_n^C + y_n^A - \alpha x_n$ où y_n est la sortie du filtre.

Programmez proprement de façon à minimiser le nombre de calculs réalisés par la machine. Regardez la différence de comportement de votre ancien filtrage (causal) et de ce nouveau filtrage (non-causal).

2.5 - Fonctions de fonction.

Cette situation arrive lorsqu'on souhaite trouver les solutions d'une équation, ou des minima d'une fonction, etc.. Dans ce cas, il faut passer à ces fonctions, une autre fonction en argument (ce qui reviendrait en C à passer un pointeur de fonction). Sous Python, ce passage de pointeur est obtenu

facilement en utilisant le nom de la fonction (sans les parenthèses).
Par exemple, créez la fonction suivante :

$$f(x) = 0.8 - \left(e^{\frac{-(x-3)^2-5}{100}} - sin\left(\pi\left(\frac{x}{30}+1\right)\right)\right)$$

en la nommant (par exemple) `MaFonction`. Créez-la et tracez-la (vous pouvez utiliser les fonctions "lambda" pour une écrire plus concise `f=lambda x: expression`)
Vous chercherez le minimum de cette fonction entre -20 et +20 grâce à la fonction `resol.fmin()` (de la librairie optimize de scipy : `import scipy.optimize as resol`). Vous chercherez la valeur annulant cette fonction qui est la plus proche de 20 grâce à la fonction `resol.fsolve()`. Répétez cette opération pour chercher la valeur annulant cette fonction qui est la plus proche de 22.

```
>> import scipy.optimize as resol
>> resol.fsolve(Mafonction, 22)
>> # ou plutôt print("Mafonction = 0 pour x = ",resol.fsolve(Mafonction, 22))
```

Regardez les différentes options de ces fonctions.

3 • UN PEU PLUS LOIN AVEC L'AFFICHAGE.

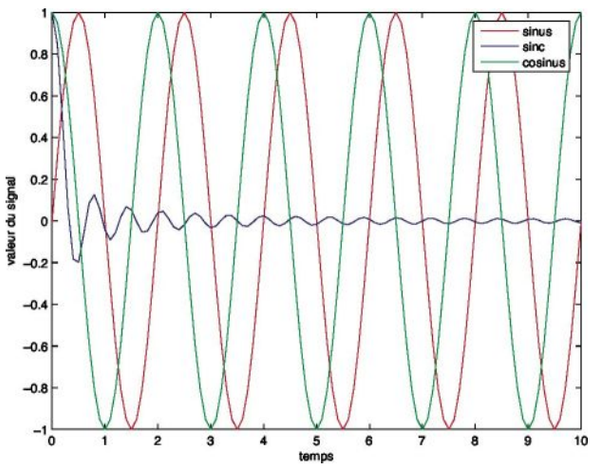
3.1 - Manipulation des axes.

La commande `pyplot.axis` (de la librairie pyplot) permet de contrôler les abscisses et ordonnées des axes que vous manipulez. Regardez l'aide de cette fonction. Créez une variable aléatoire `x` et affichez la sur un graphique qui soit compris entre o et 1.

La commande `figsize` permet de contrôler la taille des graphiques (en inch) dans une fenêtre. Regardez l'aide de cette fonction. Essayez de dessiner la variable `x` dans une autre figure qui n'occuperait que le quart de la fenêtre et positionnée en haut à gauche.

3.2 - Écriture d'un titre sur une figure.

```
>> import numpy as np
>> t = np.arange(0,10,0.1)
>> x = np.sin(np.pi*t) ; y=np.sinc(np.pi*t) ; z=np.cos(np.pi*t)
```



Affichez `x`, `y` et `z` sur une même figure (avec 3 couleurs différentes) avec `t` en abscisse. En utilisant la fonction `plt.title()` donnez un titre à votre figure. Nommez les axes verticaux et horizontaux grâce aux fonctions `plt.xlabel()` et `plt.ylabel()`. Utilisez la fonction `plt.legend()` pour donner le nom à chaque courbe comme illustrée ci-dessus.

3.3 - Sous-figures.

Utilisez la commande `plt.subplot` pour créer trois sous-figures sur la même figure sur lesquelles vous dessinerez `x` `y` et `z`.

3.4 - Le *handle.*

Le *handle* est un concept qui peut être utilisé pour tous les objets. Le *handle* contient la liste de toutes les propriétés d'un objet. Chaque fois que vous ne saurez pas manipuler un objet, il y a toujours un moyen de le faire avec le *handle.* Pour récupérer le *handle* d'un objet, il vous faut utiliser la fonction

`plt.getp()`. Par exemple pour récupérer le *handle* de la figure 1 il vous faut taper :

```
>> fig1 = plt.figure(1)
>> h = plt.getp(fig1) # ou h = fig1.properties()
```

Si vous affichez `h` vous verrez l'ensemble des variables que vous pouvez consulter ou modifier sur la figure. Par exemple, essayez.

```
>> plt.setp(fig1,visible='0') # ou fig1.set_visible('0')
```

suivit de :

```
>> plt.setp(fig1, visible='1') # ou fig1.set_visible('1')
```

Pour modifier la taille d'une figure utilisée la syntaxe suivante : `plt.figure(figsize=(10, 5))`
Pour afficher la grille : `plt.grid(True)`
Expérimentés quelques-unes des possibilités de manipulation des figures dont vous disposez.

3.5 - Visualisation de l'animation d'un graphique.

Dans de nombreux cas, on a besoin d'afficher graphiquement les résultats d'un calcul au fur et à mesure du calcul. Par exemple, essayez le code suivant :

```
>> x = np.arange(-3,3,0.1)
>> y = x
>> X, Y = np.meshgrid(x, y)
>> for a in range(30):
>>     z = np.sinc(np.sin(a*np.pi/60)*X)*np.sinc(np.sin(a*np.pi/60)*Y)
>>     fig1=plt.figure(1)
>>     ax = plt.axes(projection='3d')
>>     ax.plot_surface(X, Y, z, cmap='viridis')
>>     plt.show()
```

Il se peut que dans les nouvelles versions, l'usage de la fonction `plt.show()` qui oblige Python à tout afficher à chaque itération soit inutile.

4 • SAUVEGARDE DE VARIABLES, LECTURE DE FICHIERS.

4.1 - SAUVEGARDE DE VARIABLES.

Certains calculs peuvent avoir été très longs et on peut souhaiter sauvegarder certains résultats en vue d'une exploitation ultérieure. Pour sauvegarder des variables, il vous faut utiliser la fonction `dill.dump_session()` de la librairie `dill`. Regardez l'aide de cette fonction et sauvegardez vos

variables. Essayez d'ouvrir les fichiers sauvegardés.

4.2 - LECTURE DU FICHIER DE SAUVEGARDE.

La procédure inverse de la sauvegarde est la lecture du fichier sauvegardé. Pour cela on utilise la fonction `load_session`. Effacez la totalité de vos variables : `>> %reset -f`
vérifiez que votre environnement est vide `(whos)` puis rechargez vos variables.

5 UN PEU D'ALGEBRE.

5.1 - Un peu de calculs de moments en 2D.

Dans un premier temps, on va créer un nuage de points aléatoires orientés :

```
>> a=2 ; b=-3
>> x=np.random.randn(1,200)*10
>> y= a*x + b + np.random.randn(np.size(x))*5
```

Visualisez les points.

```
>> plt.figure(1) ; plt.plot(x,y,'.')
>> plt.axis('equal')
>> plt.show()
```

Créez un vecteur $X = \begin{bmatrix} x_1 & y_1 \\ \cdots & \cdots \\ x_n & y_n \end{bmatrix}$. calculez le barycentre de ces n points.

On rappelle que le barycentre n'est autre que le point dont les coordonnées x_0 y_0 sont les moyennes des coordonnées du nuage de point.
La matrice d'inertie du nuage de point est donnée par : $M = \frac{1}{n} X^T X$
Calculez les valeurs propres et les vecteurs propres de cette matrice .
Multipliez la matrice X^T par la matrice de vecteur propre pour obtenir le vecteur Y^T.
Visualisez le nuage de point du vecteur Y.

```
>> plt.figure(2) ; plt.plot(Y[:, 1],Y[:,2],'.')
>> plt.axis('equal')
>> plt.show()
```

On appelle et λ_1 et λ_2 les deux valeurs propres de la matrice M.
Tracez, sur cette même figure l'ellipse dont les allongements sont égaux à trois fois les racines carrées des valeurs propres de la matrice et dont le centre est le barycentre des nouveaux points. Pour vous aider, on rappelle qu'une ellipse de centre (x_0, y_0) et d'allongements α_x et α_y pour équation :

$\left(\frac{x-x_0}{\alpha_x}\right)^2 + \left(\frac{x-x_0}{\alpha_y}\right)^2 = 1$ ce qui s'écrire en utilisant un paramètre angulaire $\theta \in [-\pi, \pi]$

$x = \alpha_x cos\theta + x_0$, $y = \alpha_y cos\theta + y_0$

Pour tracer l'ellipse, créez une trentaine de points. Expliquez ce que vous constatez si vous le pouvez. Déduire de cet exercice une méthode pour tracer une ellipse ayant la même position dans le nuage des points de départ (avant transformation).

5.2 - Fonction.

Regroupez toute cette procédure dans une seule fonction dont le prototype serait :

`[barycentre, allongement] = AnalyseNuageDePoint(Nuage)`

Cette fonction prendrait en entrée un nuage de point 2D, afficherait sur une figure le nuage de points en bleu et l'ellipse englobante en rouge et renverrait le barycentre des points `(barycentre)` et les deux allongements de l'ellipse `(allongement)`.

5.3 - Rotation et translation.

Une matrice de rotation d'un angle θ dans le plan s'écrit : $\begin{bmatrix} cos\theta & sin\theta \\ -sin\theta & cos\theta \end{bmatrix}$

Cette fonction qui a comme entrée l'angle de rotation et comme sortie une matrice de rotation. Utilisez cette fonction pour créer une animation faisant tourner l'ensemble de vos points autour du point (0,0). Essayez de les faire tourner autour d'un autre point.

Créez un objet 3D.

```
>> # Import libraries
>> from mpl_toolkits import mplot3d
>> import numpy as np
>> import matplotlib.pyplot as plt

>> # Creating dataset
>> x = np.outer(np.linspace(-3, 3, 32), np.ones(32)) # 32 points
>> y = x.copy().T # transpose
>> z = (np.sin(x **2) + np.cos(y **2) )

>> # Creating figure
>> fig = plt.figure(figsize =(14, 9))
>> ax = plt.axes(projection ='3d')

>> # Creating plot
>> ax.plot_surface(x, y, z)

>> # show plot
>> plt.show()
```

Faites-le tourner selon le même principe en remarquant qu'en 3D un objet peut tourner autour de trois axes. La rotation résultant de cette rotation autour de 3 axes est obtenue en multipliant toutes les rotations entre elles :

$$Rot(\theta_x, \theta_y, \theta_z) = \begin{bmatrix} 1 & 0 & 0 \\ 0 & cos\theta_x & sin\theta_x \\ 0 & -sin\theta_x & cos\theta_x \end{bmatrix} \begin{bmatrix} cos\theta_y & 0 & sin\theta_y \\ 0 & 1 & 0 \\ -sin\theta_y & 0 & cos\theta_y \end{bmatrix} \begin{bmatrix} cos\theta_z & sin\theta_z & 0 \\ -sin\theta_z & cos\theta_z & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Utilisez les différentes options d'affichages des objets 3D. En utilisant le *handle*, faites tourner la figure sur elle-même.

Conseil : pour faire cette opération, vous aurez besoin de regrouper les points dans une seule matrice. Une telle procédure peut être obtenue en faisant :

```
>> k=1
>> for i in range (32): # 32 points
>>    for J in range (32):
>>          P[k][1]=X[i][J] ; P[k][2]=Y[i][J] ; P[k][J]=Z[i][J] ; k=k+1
```

Une procédure identique dans l'autre sens doit vous permettre de visualiser cette forme.