BLAISE PASCAL
PT 2020-2021

Fiche outil

Python pour le TIPE

L'objectif de cette fiche outil est de présenter certaines fonctions des modules `numpy` et `matplotlib` qui peuvent être utiles dans le cadre de la mise en forme finale de votre TIPE. Ces bibliothèques s'importent comme d'habitude :

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

Le parti pris de cette fiche est d'utiliser au maximum la bibliothèque `numpy`, qui permet beaucoup de souplesse, et donc d'abandonner complètement l'utilisation des listes.

Table des matières

I - Listes et tableaux numpy : le jeu des sept différences

La structure de données utilisée par `numpy` est appelée « tableau » ou « array ». Un peu comme une liste de listes, un tableau peut être à plusieurs dimensions, par exemple pour coder une matrice. Dans (presque) toute cette fiche, les tableaux seront unidimensionnels : à vous d'explorer la documentation de `numpy` si vous avez besoin de davantage.

▷ **Convertir une liste en tableau :** une liste `L` peut être convertie en tableau par `T = np.array(L)`, et réciproquement `L = list(T)`. C'est d'ailleurs le moyen classique de construire un tableau « à la main ».

▷ **Accéder aux éléments du tableau :** exactement comme pour une liste avec `T[i]`, et si le tableau contient deux dimensions `T[i][j]` ou `T[i,j]`.

▷ **Ajouter un élément :** la méthode `L.append(elt)` ne fonctionne qu'avec les listes, pour ajouter un élément `elt` en dernière position d'un tableau il faut utiliser `T = np.append(T,elt)`.
Attention ! Contrairement à la méthode des listes, la fonction `np.append(T,elt)` définit un **nouveau** tableau mais **ne modifie pas directement** le tableau `T`, d'où la nécessité de la réaffectation.

▷ **Opérations sur les éléments d'un tableau :** c'est là que réside tout l'intérêt de `numpy` ! Appliquer une fonction à un tableau revient à appliquer cette fonction à chaque élément du tableau ... et quasiment tout devient possible sans utiliser la moindre boucle.

```
T1 = np.array([1,2,3,4,5])
T2 = np.array([5,4,3,2,1])

A = T1 + T2        # les deux tableaux doivent être de même taille
B = 3 * T1 * T2**2
C = 5 * np.cos(2*T1 + np.pi/4) * np.exp(- T1 / 10)
```

▷ **Quelques tableaux bien pratiques :**
→ `np.zeros(N)` et `np.ones(N)` créent des tableaux contenant respectivement N fois la valeur 0 ou 1 ;
→ `np.zeros_like(T)` et `np.ones_like(T)` créent des tableaux de même forme que `T` ne contenant que 0 ou 1 ;
→ `np.linspace(start,stop,N)` créée un tableau de N valeurs équiréparties entre `start` et `stop`. Par exemple, `np.linspace(0,1,5)` créée le tableau `[0,0.25,0.5,0.75,1]`.

II - Tracer des jolies courbes

▷ **Fermer toutes les figures d'un coup :** `plt.close('all')` est bien pratique quand on avance par tâtonnement et que cela devient un peu trop le bazar ☺

Pour tracer une figure, il faut évidemment disposer de deux tableaux `x` et `y` de même taille qui seront l'abscisse et l'ordonnée.

II.A - Syntaxe de base

▷ **La plus rudimentaire des figures :**

```
plt.figure()    # créée une nouvelle figure
plt.plot(x,y)   # trace dans cette figure y en fonction de x
```

Ce n'est pas très joli, mais heureusement les fonctions `figure` et `plot` acceptent de nombreux arguments optionnels qui permettent de tout personnaliser. Tous les arguments optionnels se passent après les arguments obligatoires, avec une syntaxe de type « clé = valeur » :

```
plt.plot(x,y,color='red')  # mais de quelle couleur est cette courbe ?
```

Plusieurs arguments optionnels peuvent être donnés les uns à la suite des autres, l'ordre étant sans importance. Notez bien qu'il en existe trop pour être exhaustif : **explorez la documentation** si vous voulez aller plus loin que cette fiche, c'est la meilleure façon d'apprendre.

▷ **Couleur de la courbe :** argument optionnel du type « `color='red'` ». La fonction accepte des raccourcis pour certaines couleurs, par exemple `color='r'`. La courbe est bleue si rien n'est précisé.

▷ **Style de ligne :** argument optionnel du type « `ls='-'` » (`ls` pour line style). Outre le trait continu `'-'`, les lignes peuvent être en pointillés `'--'`, alternées `'-.'`, etc. Il est possible de ne pas tracer de ligne avec `ls='None'` ... mais alors il est indispensable de faire apparaître les points sous peine de ne rien voir du tout.

▷ **Marquer les points :** argument optionnel du type « `marker='o'` ». `'o'` donne des points, mais on peut aussi obtenir des croix `'+'` ou `'x'`, des carrés `'s'` (`s` pour square), des losanges `'D'` (`D` pour diamond), etc.

▷ **Pour aller beaucoup plus vite,** il est généralement possible de fusionner les trois arguments optionnels précédents en une unique chaîne de caractères donnée comme un argument obligatoire : `plt.plot(x,y,'g--D')` représente la courbe en vert, en trait pointillé, et avec des losanges sur chaque point. Il peut arriver que ces raccourcis ne fonctionnent pas ou aient un comportement innattendu, auquel cas il faut revenir à la syntaxe « clé = valeur ».

▷ **Légender les axes :** Les fonctions `plt.xlabel('axe␣des␣x')` et `plt.ylabel('axe␣des␣y')` permettent de faire apparaître un nom pour chaque axe. Il est possible d'utiliser des formules mathématiques en LaTeX entre dollars, par exemple `plt.ylabel('$\cos(\omega␣t)$')`.

▷ **Légender les courbes :** Lorsque plusieurs courbes apparaissent sur une même figure, il peut être utile de faire apparaître une légende en nommant chaque courbe via l'argument optionnel `label='nom␣de␣la␣courbe'` puis en utilisant la fonction `plt.legend()`. La fonction peut prendre un argument optionnel du type `loc='upper␣left'` qui indique à quel endroit de la figure la légende doit être placée.

▷ **Valeurs limites des axes :** Les fonctions `plt.xlim(xmin,xmax)` et `plt.ylim(ymin,ymax)` permettent d'imposer les valeurs extrêmes des axes des abscisses et des ordonnées.

▷ **Un exemple :** le code ci-dessous permet de produire la figure 1.

```
# Valeurs expérimentales :
X_exp = np.array([0, 1, 2, 3, 4, 5])
Y_exp = np.array([0, 0.7, 4.5, 9.3, 15.5, 24])  # x^2 et du bruit

# Valeurs théoriques :
X_th = np.linspace(0,5,100)
Y_th = X_th**2

# Tracé :
plt.figure()
plt.plot(X_exp, Y_exp, 'o', label='Expérience') # pas besoin de pré
    ciser marker='o' grâce à la syntaxe compacte.
plt.plot(X_th, Y_th, 'r', label='Théorie')   # pas besoin de préciser
    color='red' grâce à la syntaxe compacte.
plt.xlabel('$x$')
```

```
plt.ylabel('$y = x^2$ (enfin, normalement)')
plt.ylim(-10,26)
plt.legend(loc='lower right')
```

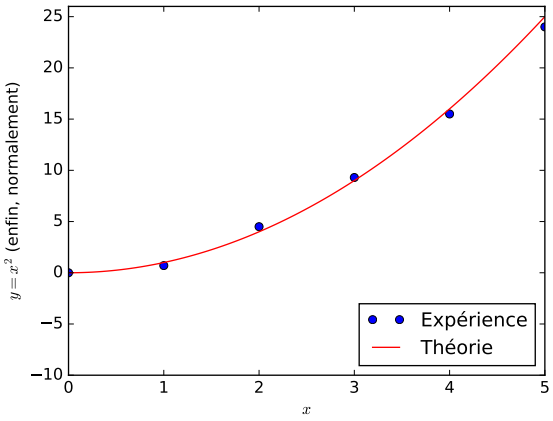

Figure 1 – Un exemple de figure simple.

II.B - D'autres petites choses

▷ **Incertitudes :** La fonction `errorbar` permet de tracer des barres d'incertitude autour des points expérimentaux, données sous forme d'arguments optionnels `xerr` et `yerr`. En donnant un seul des deux arguments, on représente la barre d'incertitude pour une variable seulement. Cette fonction accepte une bonne partie des arguments optionnels décoratifs de `plot`. Le code ci-dessous est un extrait qui pourrait améliorer la figure 1 en donnant des incertitudes sur les points expérimentaux.

```
X_exp = np.array([0, 1, 2, 3, 4, 5])
Y_exp = np.array([0, 0.7, 4.5, 9.3, 15.5, 24])  # x^2 et du bruit

DeltaX = .05 * np.ones_like(X_exp) # incertitude constante de 0.05
DeltaY = .1 * Y_exp      # incertitude relative de 10%

plt.figure()
plt.errorbar(X_exp, Y_exp, xerr=DeltaX, yerr=DeltaY)
```

▷ **Échelle logarithmique :** Les fonctions `semilogx`, `semilogy` et `loglog` s'utilisent exactement comme la fonction `plot`, mais représentent l'un, l'autre ou les deux axes en échelle logarithmique.

▷ **Histogramme :** La fonction `plt.hist(T)` permet de tracer de tracer l'histogramme des valeurs contenues dans le tableau `T`. L'argument optionnel **range** permet d'imposer l'intervalle sur lequel l'histogramme est tracé, et l'argument optionnel `bins` permet de choisir le nombre de sous-intervalles utilisés. Par exemple, si vos notes de DS sont rangées dans le tableau `notes`, leur histogramme s'obtient avec la ligne de code `plt.hist(notes, range=(0,20), bins=20)`.

▷ **Tracer une ligne verticale ou horizontale :** Pour indiquer par exemple un seuil, une valeur limite, un axe, etc, les fonctions `plt.axvline(x=2)` et `plt.axhline(y=2)` permettent de tracer une ligne qui occupe toute la largeur de la figure, ici respectivement en $x = 2$ et $y = 2$. Ces fonctions acceptent une bonne partie des arguments optionnels décoratifs de `plot`.
Attention ! `x` et `y` sont des arguments optionnels, ce qui impose la syntaxe « clé = valeur » : `plt.axvline()` trace la ligne en $x = 0$.

▷ **Écrire du texte sur une figure :** La fonction `plt.text(x,y,'texte')` permet d'écrire une chaîne de caractères au point de la figure de coordonnées (x, y). De manière précise, les coordonnées sont celles du coin inférieur gauche de la zone de texte. Pour que les coordonnées désignent la position du centre, il faut utiliser les arguments optionnels `ha='center',va='center'` (pour horizontal et vertical alignment).

▷ **Un exemple :** le code ci-dessous permet de tracer le diagramme de Bode en gain d'un filtre du premier ordre, représenté figure 2 ... et de présenter quelques décos supplémentaires !

```
# Les fréquences :
fc = 2e3 # fréquence de coupure 2 kHz
```

```
f = np.logspace(1,5,500) # 500 pts logarithmiquement espacés entre 10^1 et 10^5 Hz ... attention il faut indiquer les puissances

# Calcul du gain
H = 1/(1 + 1j*f/fc)      # utilisation des complexes avec 1j
GdB = 20 * np.log10(np.abs(H))  # np.log est népérien, np.ln n'existe pas, np.abs donne le module

plt.figure()
plt.semilogx(f,GdB,'r')

# Repérer la fréquence de coupure
plt.axvline(x=fc,'b--')
plt.text(fc,-41,'$f_c$',ha='center',va='top',color='blue')

plt.axhline(y=-3,'b--')
plt.text(9,-3,'-3␣dB',ha='right',va='center',color='blue')

# Les deux lignes suivantes permettent d'ajouter une grille pour faciliter la lecture
# Gris foncé épais sur les graduations majeures, gris clair fin sur les mineures
# Le nombre décimal indique le niveau de gris.
plt.grid(which='major', ls='-', color='.5', linewidth=.5)
plt.grid(which='minor', ls='-', color='.75', linewidth=.25)

# Les axes :
plt.axhline(y=0,color='black')  # axe horizontal
plt.xlabel('$f$␣(Hz)')
plt.ylabel('$G$␣(dB)')
plt.ylim(-40,5)
```

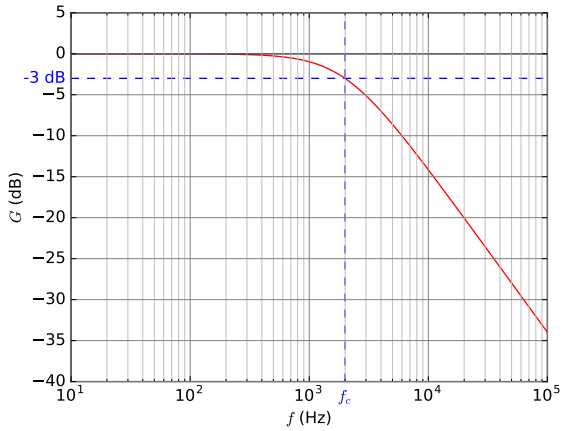

Figure 2 – Exemple de tracé d'un diagramme de Bode.

II.C - Sauvegarder une figure

Les figures 1 et 2 sont jolies, mais on peut regretter qu'elles soient écrites un peu trop petit, notamment les axes : c'est parce qu'elles ont été sauvegardées « à la main », puis redimensionnées « à la main » car elles étaient bien trop grosses par rapport à la page. Ce paragraphe explique comment y remédier.

▷ **Modifier la taille de la figure :** Comme toutes les autres, la fonction `figure` accepte un certain nombre d'arguments optionnels. Celui qui donne les dimensions de la figure est `figsize`, à renseigner sous forme d'un doublet `(largeur,hauteur)` qui indique les dimensions de la figure ... en pouces. Par exemple, `plt.figure(figsize=(6.4,4.8))` redonne les dimensions par défaut.

▷ **Modifier la taille de la police d'écriture :** Toutes les fonctions permettant l'affichage de chaînes de caractère (`xlabel, legend, text`, etc.) acceptent un argument optionnel `fontsize` qui donne la taille de la police d'écriture.

Pour modifier la taille des nombres écrits sur les axes, on utilise `plt.xticks(fontsize=8)` ou `plt.yticks(fontsize=8)`, qui les met ici en taille 8. La taille par défaut est 10 ou 12 selon les fonctions. Il est possible de changer globalement la taille par défaut de toutes les zones de texte juste après l'import de `matplotlib` : ici, la taille est mise à 8.

```
import matplotlib.pyplot as plt
plt.rcParams.update({'font.size':8})
```

▷ **Enregistrer la figure :** Il est possible d'enregistrer la dernière figure ouverte avec la fonction `plt.savefig(ma-jolie-figure.pdf)`. La figure est alors sauvegardée dans le même dossier que le fichier Python. De nombreux formats de fichier sont possibles : `jpg`, `png`, `pdf`, etc. Pour la lisbilité ultérieure de la figure, il est recommandé de privilégier un format vectoriel, `pdf` étant probablement le plus simple à manipuler. Il est fréquent qu'une grande zone blanche entoure la figure, ce qui n'est pas toujours pratique : on peut la réduire avec l'argument optionnel `bbox_inches='tight'`.

▷ **Un exemple :**

```
X_exp = np.array([0, 1, 2, 3, 4, 5])
Y_exp = np.array([0, 0.7, 4.5, 9.3, 15.5, 24])

X_th = np.linspace(0,5,100)
Y_th = X_th**2

plt.figure(figsize=(2,3))  # une figure plus haute que large
plt.plot(X_exp, Y_exp, 'o', label='Expérience')
plt.plot(X_th, Y_th, 'r', label='Théorie')
plt.xlabel('$x$', fontsize=24)   # on écrit très gros !
plt.ylabel('$y = x^2$ (enfin, normalement)')
plt.ylim(-10,26)
plt.yticks(fontsize=6)      # on écrit tout petit !
plt.legend(loc='lower right',fontsize=8)  # on écrit petit, mais moins

plt.savefig('exple-3.pdf',bbox_inches='tight')
```

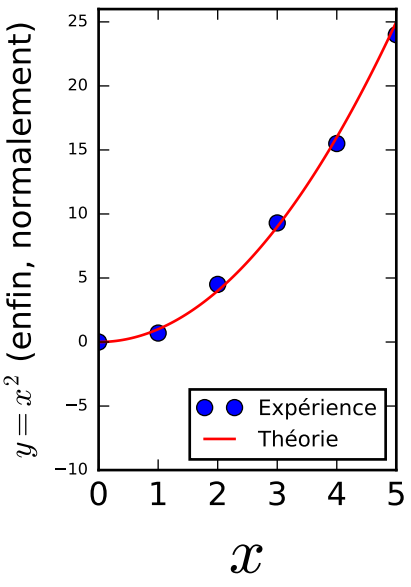

Figure 3 – **Dernier exemple.** Contrairement aux autres figures, aucun redimensionnement manuel n'a eu lieu ici.

III - Régression linéaire

La régression linéaire est une méthode très courante d'exploitation de mesures en sciences, qu'il est possible de réaliser avec Python grâce à la fonction `polyfit` de la bibliothèque `numpy`.

On suppose disposer de deux grandeurs expérimentales (ou calculées à partir de grandeurs expérimentales) `Xexp` et `Yexp`. La régression linéaire s'obtient avec la fonction `np.polyfit(Xexp,Yexp,1)`. Le dernier argument désigne l'ordre du polynôme par lequel est faite la régression, toujours égal à 1 pour une régression linéaire. Cette fonction renvoie un tableau à deux éléments qui désignent les coefficients de la régression.

Un exemple : le code ci-dessous permet d'obtenir la figure 4 (qu'il conviendrait ensuite de légender correctement, réduire la taille de police d'écriture, décorer, etc.).

```
Xexp = np.array([0, 2, 4, 6, 8, 10])
Yexp = np.array([0.5, 7.9, 11, 17.5, 26, 31.8])    # 3x+1 avec un bruit

p = np.polyfit(Xexp,Yexp,1) # p est un tableau

Ymod = p[0] * Xexp + p[1] # p[0] est la pente, p[1] l'ordonnée à l'
    origine

plt.figure(figsize=(3,2))
plt.plot(Xexp,Yexp,'o')      # points expérimentaux
plt.plot(Xexp,Ymod,'r')      # droite modèle

plt.savefig('exple-regression.pdf',bbox_inches='tight')
```

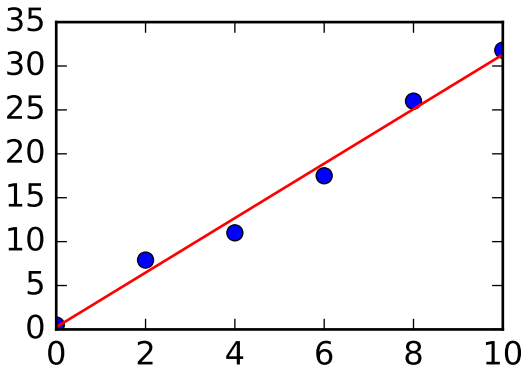

Figure 4 – Exemple de régression linéaire.

IV - Importer des données à partir d'un fichier

La grande majorité des capteurs du lycée ne peuvent pas directement être interfacés avec Python, il faut passer par un logiciel d'acquisition intermédiaire. Tous ces logiciels permettent normalement d'exporter les données dans des fichiers texte, au format `txt`, `csv`, etc. Exploiter ces données dans un script Python exige d'importer le fichier et de transformer les informations intéressantes en tableaux.

Attention ! Avant toute tentative de lecture de fichier dans Python, **la première chose à faire** est d'ouvrir le fichier dans un éditeur de texte (bloc-note Windows ou équivalent) pour regarder comment les données y sont organisées.

- **Convertir en tableau les données d'un fichier :** L'import de fichier et la conversion en un tableau se fait grâce à la fonction `np.loadtxt('le-fichier.txt')`. Si le fichier ne contient que des nombres, elle renvoie un tableau à deux dimensions contenant les données ... et un message d'erreur sinon, d'où les manipulations suivantes.
  Attention ! Si le fichier à importer n'est pas dans le même dossier que le fichier Python, il faut donner le chemin d'accès complet (absolu ou relatif).
- **Supprimer les premières lignes :** Les fichiers obtenus par exportation d'un logiciel d'acquisition contiennent souvent des premières lignes informatives : nom de la colonne, date de l'acquisition, etc. Pour ne pas en tenir compte lors de l'import, indiquer à la fonction `loadtxt` l'argument optionnel `skiprows=2`, ici par exemple pour sauter deux lignes. Pour connaître le nombre de lignes à sauter ... il faut ouvrir le fichier.

- **Indiquer le séparateur de colonnes :** Il arrive que la fonction `loadtxt` ne reconnaisse pas automatiquement le symbole séparateur de colonnes utilisé dans le fichier : il faut alors l'indiquer par l'argument optionnel `delimiter=','` s'il s'agit d'une virgule, sans oublier les guillemets de la chaîne de caractère. Pour connaître le délimiteur ... il faut ouvrir le fichier.
- **Extraire une colonne du tableau :** Il faut ensuite extraire du tableau renvoyé par `loadtxt` les grandeurs intéressantes, qui sont généralement des colonnes : la deuxième colonne d'un tableau `T` s'obtient avec la syntaxe `T[:,1]` ... eh oui, Python compte à partir de 0. On obtient ainsi des classiques tableaux unidimensionnels, avec lesquels on peut faire des tracés, des régressions linéaires, etc.
- **Un exemple :** le code ci-dessous permet de récupérer trois tableaux correspondant aux trois colonnes du fichier `fichier-donnees.csv` reproduit ci-contre.

```
1 donnees = np.loadtxt("fichier-donnees.csv",
      skiprows=3, delimiter=',')

3 t = donnees[:,0]
4 x = donnees[:,1]
5 y = donnees[:,2]
```

```
Des donnees cools„
t,x,y
s,cm,cm
1,2,1
2,3,2
3,5,1
4,6,2
5,8,1
6,9,2
```