Outils numériques

Numpy, Scipy et Matplotlib en 10 créneaux de cours

Contexte

Le but de ce cours est d'apprendre à utiliser efficacement Numpy/Scipy et Matplotlib pour écrire des programmes de calcul scientifique et visualiser les données obtenues. Vous pourrez trouver l'intégralité de la documentation de Scipy et Numpy ici! http://docs.scipy.org/doc/numpy/genindex.html en particulier il faudra s'intéresser aux explications données sur cette page: http://docs.scipy.org/doc/numpy/reference/arrays.ndarray.html. Concernant Matplotlib il faut regarder ici https://matplotlib.org/.

De nombreuses notions propres à Numpy/Scipy sont abordées dans ce TD, n'hésitez pas à solliciter votre encadrant si vous ne savez pas répondre de manière satisfaisante à une question et ne finissez pas ce TD sans avoir les idées claires sur ce qu'on entend par "array slicing" ou "fancy indexing" par exemple.

Introduction rapide à la bibliothèque numpy

La bibliothèque numpy permet d'effectuer des calculs numériques, et de manipuler des tableaux à N dimensions (bien souvent on se contentera de N=1, 2 ou 3).
Pour pouvoir l'utiliser on doit d'abord l'importer en plaçant en haut du programme: *import numpy as np*
Toutes les fonctions issues de cette bibliothèque seront précédées du préfixe *np.* pour signifier d'où elles proviennent.
Pour le calcul numérique, sont disponibles :

- Variables prédéfinies: par exemple les valeurs approchées de π et e via les commandes *np.pi* et *np.e*
- Fonctions prédéfinies :
- *np.exp* pour l'exponentielle
- *np.log* pour ln
- *np.cos* pour cos
- *np.sin* pour sin
- *np.tan* pour tan
- *np.sqrt* pour la fonction racine carrée
- *np.abs* pour la fonction valeur absolue
- *np.floor* pour la fonction partie entière
- *np.arctan* ou *np.arctan2* (préférable) pour arctan
- et des milliers d'autres, Google est votre ami

Pour les tableaux Numpy, on se contentera généralement d'utiliser les tableaux de une à trois dimensions, mais il n'y a pas de limite en dehors de la mémoire maximale de votre ordinateur.

- Tableau à une dimension : (tableau 1D): c'est une succession de valeurs indexées à partir de l'indice 0. On peut s'en servir pour stocker l'équivalent de l'objet mathématique nommé matrice-ligne par exemple, ou faire des opérations numériques sur des données préalablement stockées sous forme de liste et converties en tableau numpy. Il n'y a pas de notion de ligne ou de colonne ici.
- Tableau à deux dimensions : (tableau 2D): c'est un tableau d'éléments au sens classique, avec des lignes et des colonnes. On peut

s'en servir pour stocker l'équivalent de l'objet mathématique nommé matrice mais ce n'est pas son but premier.
- Tableau à trois dimensions : (tableau 3D): c'est un empilement de tableaux 2D, on peut par exemple s'en servir pour stocker les différentes composantes de couleur d'une image.

Remarque fondamentale:

Les fonctions prédéfinies sous Python listées plus haut ne sont pas l'équivalent des fonctions numériques que l'on connait en mathématiques. Le paramètre d'entrée n'est pas forcément un flottant, ce peut être un tableau. Dans ce cas la fonction numérique s'applique à chaque coefficient du tableau.

Création de tableaux

Le tableau à une dimension se déclare en listant ses éléments (on parlera de ses coefficients). Par exemple la commande *np.array([1, -4, 9, 0, 31, -56])* crée un tableau 1D à partir de la liste *[1,-4,9,0,31,-56]*.

On peut faire de même avec les tableaux à N dimensions en partant de listes de listes de listes... de listes.

Par exemple la commande *np.array([[2, 3], [4, 5]])* créé un tableau 2 dimensions avec 2 lignes et 2 colonnes.

Pour un tableau à 1 dimension on accède à ses éléments de la même manière que pour une liste, par exemple *A[3]*.

Pour un tableau à plus d'une dimension on met les coordonnées de l'élément auquel on souhaite accéder à la suite en les séparant par des virgules, par exemple *A[3, 7, 1].*

Attention : le premier coefficient est repéré (indexé) par 0. Si un tableau 1D possède 6 coefficients, comme celui du premier exemple, ses éléments sont numérotés de 0 à 5.

- Pour obtenir le nombre d'éléments d'un tableau nommé A: A.size
- Pour obtenir le nombre de dimensions, ainsi que leurs tailles respectives, d'un tableau nommé A: A.shape

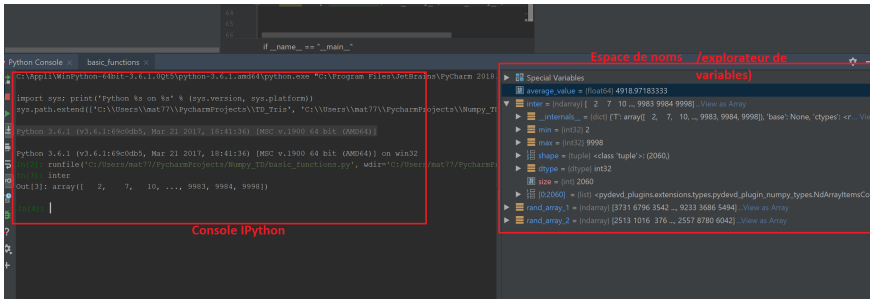
Testez :

```
A = np.array([1, 0, 87, 23, 0 , 6])
print(A)
print(A[1])
print(A[0])
print(A[-1])
print(A.size)
print(A.shape)
print(A[6])
```

```
B = np.array([[1, 0], [87, 23], [0 ,6]])
print(B)
print(B[0])
print(B[0, 1])
print(B[-1, 0])
print(B[:, 1])
print(B.size)
print(B.shape)
```

Remarques:

1. Pour l'appel du dernier coefficient d'un tableau t de taille n, on peut faire : *t[-1]* (au lieu de *t[n-1]*) De même pour l'avant dernier coefficient, faire : *t[-2]*. Etc...
2. Créer un tableau à plusieurs dimensions à partir de liste de liste... de liste peut être syntaxiquement lourd. Il est plus aisé de créer un tableau 1D avec une liste simple et le redimensionner par la suite. Par exemple pour créer le tableau 2D nommé B ci-dessus on aurait pu faire: *B = np.array([1, 0, 87, 23, 0 ,6]).reshape((3,2))* et ainsi éviter d'avoir à manipuler les différents [et] imbriqués.
3. L'explorateur de variables intégré à Pycharm permet de visualiser rapidement le contenu d'un tableau Numpy, n'hésitez pas à l'utiliser (cf ci-dessous). Pour l'activer il suffit de faire un click droit dans le code et sélectionner "Run file in Python Console". Suivant votre version de Pycharm il peut être nécessaire de désactiver la console quand vous n'en avez pas besoin, décocher alors l'option *Run in python console* dans le menu *Run -> Edit Configurations*.



Quelques commandes utiles

- *np.zeros(n)*: renvoie un tableau 1D de taille n ne contenant que des zéros.
- *np.zeros((n, m))*: renvoie un tableau 2D de taille (n, m) ne contenant que des zéros.
- *np.ones(*arguments ici*)*: fonctionne comme zeros mais avec des uns.
- *np.linspace(a, b, N)*: renvoie un tableau 1D de taille N , les N coefficients étant équirépartis entre a et b inclus.
- *np.arange(n)*: renvoie le tableau 1D ayant pour coefficients les n nombres entiers de 0 inclus à n exclu.
- *np.arange(a, b)*: renvoie le tableau 1D ayant pour coefficients les nombres entiers de a inclus à b exclu.
- *np.arange(a, b, p)*: renvoie le tableau 1D ayant pour coefficients les nombres entiers de a inclus à b exclu, avec un pas égal à p.
- *np.sum(A)*: renvoie la somme des coefficients de A.
- *np.prod(A)*: renvoie le produit des coefficients de A.
- *np.min*(A): renvoie le plus petit des coefficients de A. (Et *np.max(A)* le plus grand)
- *np.mean(A)*: renvoie la moyenne des coefficients de A.
- Pour avoir la documentations sur toute autre commande: https://numpy.org/doc/stable/

Opérateurs arithmétiques

Les opérations +, ∗, −, /, ∗∗ se font coefficient par coefficient Par exemple, si L est le tableau [0 8 9 −3] et A le tableau [5 −4 1 2] , mathématiquement, seuls L + A et L − A ont un sens. On ne peut pas multiplier deux matrices-lignes entre elles, ni élever une matrice-ligne au carré par exemple. Par contre, avec Numpy, ça a un sens car il a été conçu pour manipuler des tableaux, l'analogie avec les matrices étant secondaire. Évidemment les deux tableaux doivent avoir la même taille ! On peut tester :

```
L = np.array([0, 8, 9, 3])
A = np.array([5, 4, 1, 2])
ratio_LA = L / A
A_carre = A**2
prod_AL = A * L
```

La multiplication, au sens matriciel, se fait avec l'opérateur @, les opérandes doivent avoir des dimensions compatibles.

```
A = np.arange(16).reshape((4, 4))
X = np.array([5, 4, 1, 2])
print(A)
print(X)
B = A @ X
print(B)
```

Note: on peut transposer facilement un tableau en le postfixant par *.T*

```
A = np.arange(16).reshape((4, 4))
X = np.array([5, 4, 1, 2])
print(A.T)
print(X.T)
```

- Comment expliquer le résultat obtenu pour *X.T* ?

Lectures complémentaires obligatoires

Cette introduction rapide aux tableaux Numpy n'est en rien suffisante pour commencer à les utiliser mais vous permet d'avoir

les notions minimales pour pouvoir comprendre l'excellent cours d'introduction à Scipy et Numpy disponible ici: https://scipy-lectures.org/

Avant de commencer les exercices vous devez absolument avoir lu, et compris, le chapitre 1.4.1: https://scipy-lectures.org/intro/numpy

/array_object.html#id2. Vous pouvez commencer directement au chapitre 1.4.1.2 "Creating arrays". Ne pas faire le chapitre *1.4.1.4. Basic visualization* car on ne vous a pas fait installer les outils nécessaires en début d'année.

Partie 1: pratique de numpy

Exercice 1: approche naïve du calcul scientifique en Python

Le but de cet exercice va être de vous montrer quel est le réflexe fondamental à avoir quand on programme en Python+Numpy/Scipy (ou en Python tout seul aussi).

À défaut d'avoir des données supposées issues de campagnes de mesures nous allons travailler sur un tableau de données aléatoires:

1. Lancez Pycharm.
2. Récupérez le fichier basic_func.py et sauvegardez le dans votre répertoire Pycharm que vous avez dédié à ce TD.
3. Que fait ce script ?
4. Que peut-on déduire du résultat ? (ne pas hésiter à exécuter le code plusieurs fois de suite)

Indications pour répondre aux questions:
Notions vues dans cet exercice: calcul vectoriel.

Fonctions Numpy: *mean, randint*

Éléments de correction:

En général on constatera au moins un ordre de grandeur d'écart entre le temps d'exécution d'une fonction écrite purement en Python et son strict équivalent évalué via Numpy/Scipy. Évidemment ce n'est pas toujours vrai, si la fonction Numpy que vous utilisez est elle même écrite en Python il n'y aura aucun gain à part si vous avez fait une erreur dans votre algorithme. Par contre le plus souvent les fonctions Numpy ne sont pas écrites en Python, elles font appel à des programmes écrits dans des langages beaucoup plus rapides, tels que C ou Fortran, mais aussi plus pénibles à écrire que Python. Le rôle de Numpy/Scipy est de servir de glue entre vos données dans votre programme Python, qui est facile à écrire mais est lent à s'exécuter, et les fonctions plus rapides en C ou en Fortran qui vont manipuler ces données et fournir un résultat. **En général si vous commencez à écrire une boucle en Python pour effectuer un calcul il faudra toujours vous demander s'il n'existe pas déjà une fonction Numpy ou Scipy qui fait déjà ce que vous cherchez à réaliser, et la plupart du temps la réponse sera oui.**

Pour finir on insistera lourdement sur le fait qu'il n'y a pas une manière magique de calculer une moyenne dans Numpy et une manière maladroite que vous auriez programmée dans cet exercice (sauf si vous avez écrit des bêtises). Les complexités algorithmiques de votre fonction et de celle de Numpy sont strictement identiques, l'algorithme est le même, le problème vient bien des opérations individuelles qui sont nettement plus lentes dans un des deux cas.

Exercice 2: Généralités sur les tableaux numpy

On s'attachera à trouver une réponse qui n'implique pas d'écrire explicitement une boucle en Python.

Dans cet exercice nous allons manipuler un tableau de données aléatoires (à copier-coller dans le fichier Python dédié à ce TP):

```
foo = np.random.randint(0, 10, (np.random.randint(50, 60), np.random.randint(42, 69)))
```

Remarque:

Quand on parlera de "ligne N du tableau X" on se référera à la N*ième* ligne du tableau, pas à X[N]. Par exemple par "première ligne" on entend X[0], pas X[1], de même par "première colonne" on entend X[:,0] et pas X[:,1].

Les réponses attendues se font en une ligne de code chacune.

- Que fait la ligne servant à créer *foo* ?
- Comment peut-on connaître le nombre de lignes du tableau de données qui vient d'être créé ? Le nombre de colonnes ? Le nombre total d'éléments ?
- Quel est le type de données contenu dans foo ? Comment convertir un tableau d'un type de données à un autre ?
- Quel est le sens de ce que renvoie la commande suivante ? *foo>8*
- Quelles sont les coordonnées, dans le tableau, des points dont la valeur est inférieure strictement à 5 ? (fonction *where*)
- Multipliez ces points par 2.
- Que font les lignes suivantes ? Il peut être intéressant de se renseigner sur la syntaxe dite "slicing" qu'on pourrait traduire par "découpage en tranches". Voir le paragraphe "Basic slicing" ici: http://docs.scipy.org/doc/numpy-1.11.0/reference/arrays.indexing.html
  - bar = foo[:, 0:13:2]

- bar_copy = foo[:, 0:13:2].copy()
- bar_float = bar.astype(float)
- baz = foo[4:14,(1,6,11,26)]

- Vérifiez que *foo[5,6]*, *bar[5,3]*, *baz[1,1]* et *bar_float[5,3]* contiennent la même valeur
- Modifiez la valeur de *foo[5,6]* en la passant à 42. Qu'observez vous ? Pourquoi ? Si vous ne comprenez pas ce que vous venez d'observer sollicitez votre encadrant de TD, ne passez pas à la question suivante !

Indications pour répondre aux questions:

Notions vues dans cet exercice: types de données, array slicing, fancy indexing, références et copies.

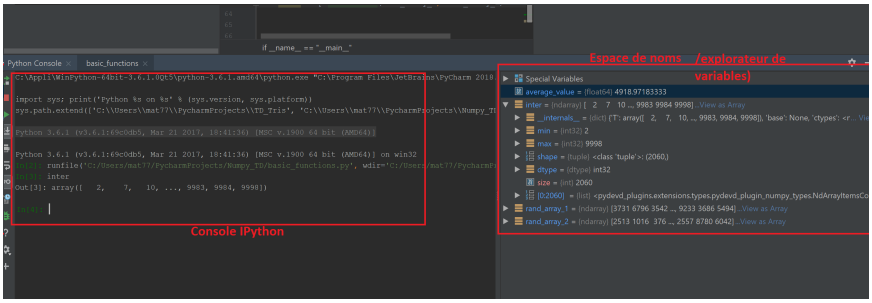
Variables d'instance: *shape*, *size* et *dtype* des objets de type ndarray

Méthodes : *astype()* des objets de type ndarray

Fonctions Numpy: where().

Rappel (pour ceux qui lisent trop vite l'introduction):

L'explorateur de variables intégré à Pycharm permet de visualiser rapidement le contenu d'un tableau Numpy, n'hésitez pas à l'utiliser (cf ci-dessous). Pour l'activer il suffit de faire un click droit dans le code et sélectionner "Run file in Python Console". Suivant votre version de Pycharm il peut être nécessaire de désactiver la console quand vous n'en avez pas besoin, décocher alors l'option *Run in python console* dans le menu *Run -> Edit Configurations*.



Exercice 3: manipulation de tableaux

On s'attachera à trouver une réponse qui n'implique pas d'écrire explicitement une boucle en Python.

On utilisera comme point de départ des longueurs aléatoires des petits côtés d'un ensemble de triangles rectangles:

```
short_sides = np.random.randint(1, 10, (200,2))
```

Les réponses attendues se font en une ligne de code chacune.

- Calculer l'hypoténuse de chaque triangle en utilisant le théorème de Pythagore.
- Créer un tableau *triangles* rassemblant *short_sides* et les hypoténuses calculées ci-dessus (privilégier column_stack plutôt que atleast_2d/concatenate).
- Choisir un des deux angles non droit de chaque triangle et le calculer via l'arc cosinus, l'arc sinus et l'arc tangente.
- Les valeurs trouvées sont-elles toutes égales ? Pourquoi ?
- Quelques unes sont-elles malgré tout égales ?
- Peut-on tout de même savoir si les valeurs trouvées sont toutes à peu près égales ?
- Quel est le sens de "à peu près" dans la fonction que vous venez d'utiliser pour la question précédente ?

Indications pour répondre aux questions:

Notions vues dans cet exercice: opérations sur les tableaux, concaténation, comparaison, axes (axis en anglais).

Fonctions Numpy: *atleast_2d, concatenate, column_stack, all, any, allclose, arccos, arcsin, arctan2.*

Exercice 4: algèbre linéaire

Les tableaux Numpy peuvent être utilisés comme des matrices et servir à résoudre des problèmes d'algèbre linéaire. La plupart des fonctions sont dans le module *linalg* de Numpy. Vous pouvez soit les préfixer par *np.linalg*, soit faire *import numpy.linalg as npla* (par

exemple) et les préfixer par *npla*.

Soit le système linéaire suivant:

```
8x + 2 y + z + t + 5w = 42
x + 9 y + z + 3 t +  2 w = 12
3x + 5 y + 42 z + 28 t + 5 w = 28
x + 9 y + z − 50 t + 12 w = 90
6 x + 5 y + 3 z + 12 t − 38 w = 32
```

- Le mettre sous forme matricielle A X = B avec A tableau 2D 5*5 et B tableau 1D 5.
- Quel est le rang de A ? (rank en anglais)
- Quelles sont ses valeurs propres ? Donner une base de vecteurs propres (eigenvalues, eigenvectors en anglais)
- À partir des résultats ci-dessus construire les matrices P et D, respectivement matrice de passage et forme diagonale de A.(*diag* pour faire une matrice diagonale à partir d'un array 1D.)
- Vérifier que A est proche de P D P^{-1} (*inv* pour inverser une matrice, *allclose* pour la comparaison)
- Quelle est la différence fondamentale entre les calculs effectués ici et ceux que vous faites en TD de maths sur table ?
- Résoudre le système grâce à la fonction *solve* de numpy.
- Vérifier que la solution obtenue est acceptable en comparant l'erreur absolue et relative entre le produit matriciel de A par X et la valeur de B désirée.

Conseil: il est plus aisé de saisir A comme un array 1D de 25 éléments puis de faire un reshape 5*5.

Note: quel que soit le langage utilisé ne **JAMAIS** résoudre numériquement un système linéaire en calculant A^{-1} puis en le multipliant par B. Le résultat est plus long à obtenir, demande plus de mémoire et est de moins bonne qualité que l'utilisation directe de *solve*.

Partie 2: Représentation graphique

On utilise habituellement la bibliothèque Matplotlib pour effectuer des représentations graphiques de données en Python.

Voici un exemple montrant comment représenter la fonction *sin(x)* entre 0 et 10.

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

X = np.linspace(0, 10, 100) # 100 points entre 0 et 10
plt.plot(X, np.sin(X))
plt.show()
```

Exercice 5: présenter correctement les données, courbes multiples

- Le graphique obtenu avec le code ci-dessus est notoirement incomplet, il n'y a pas de légende sur les axes, pas de titre, etc. Le compléter en vous inspirant du tutoriel https://matplotlib.org/2.0.2/users/pyplot_tutorial.html
- Modifier le tableau X pour représenter sin(X) sur deux périodes de cette fonction avec un pas de 10^{-4}.
- Rajouter sur le même dessin la représentation graphique de cos(X)
- Rajouter un cartouche avec une légende de couleurs permettant de différencier les courbes. Voir ici à **partir de la ligne "or (pyplot style)"** https://matplotlib.org/stable/tutorials/introductory/usage.html#the-object-oriented-interface-and-the-pyplot-interface

Exercice 6: échelle logarithmique, "subplot"

On dispose du lot de données suivant obtenu par l'observation d'un phénomène non précisé (utilisez votre imagination pour mettre des unités sur chaque jeu de données !).

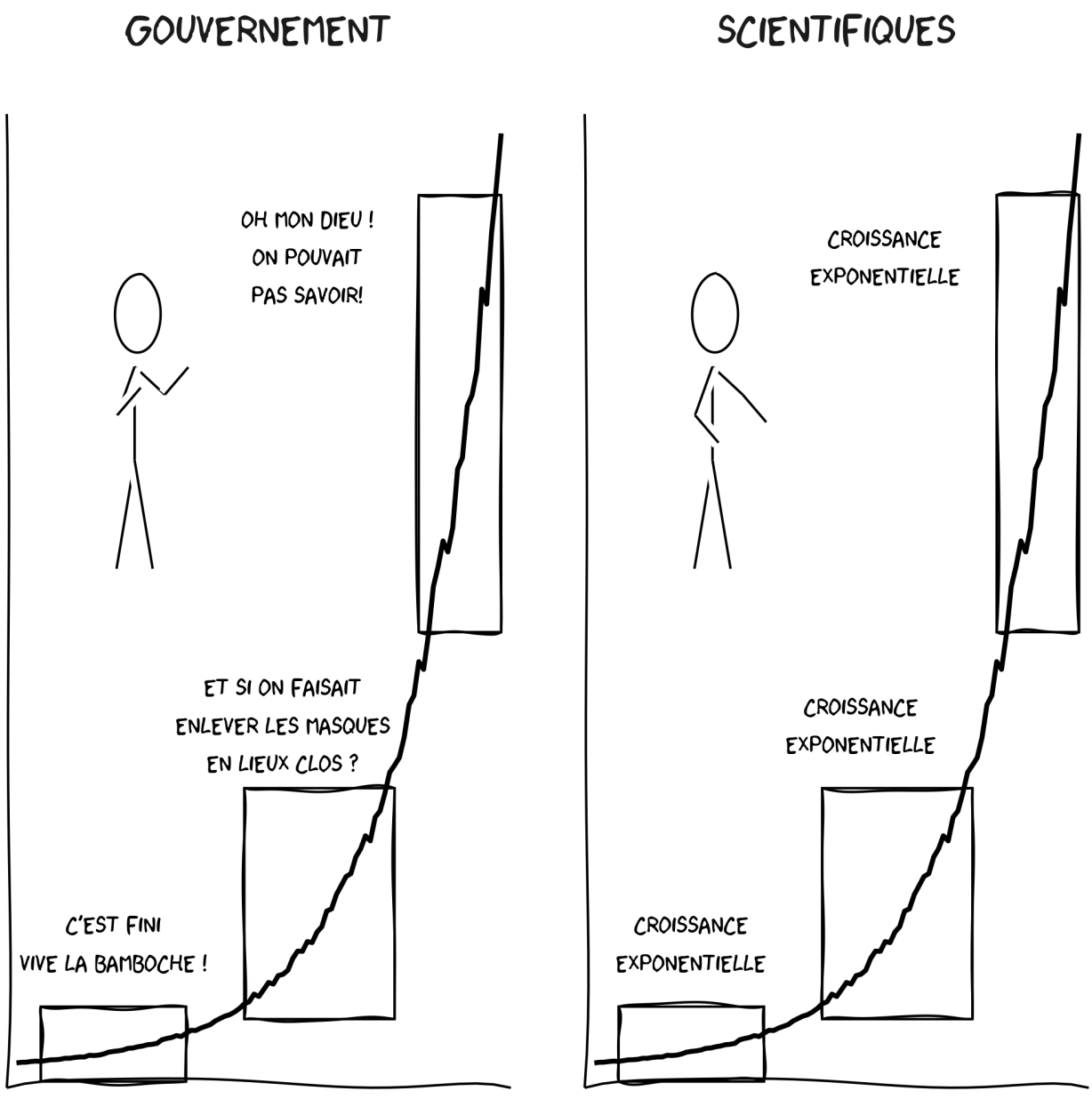
Points de mesure:

[1.0, 1.02, 1.05, 1.07, 1.1, 1.12, 1.15, 1.18, 1.2, 1.23, 1.26, 1.29, 1.32, 1.35, 1.38, 1.42, 1.45, 1.48, 1.52, 1.56, 1.59, 1.63, 1.67, 1.71, 1.75, 1.79, 1.83, 1.87, 1.92, 1.96, 2.01, 2.06, 2.1, 2.15, 2.21, 2.26, 2.31, 2.36, 2.42, 2.48, 2.54, 2.6, 2.66, 2.72, 2.78, 2.85, 2.92, 2.98, 3.05, 3.13, 3.2, 3.27, 3.35, 3.43, 3.51, 3.59, 3.68, 3.76, 3.85, 3.94, 4.04, 4.13, 4.23, 4.33, 4.43, 4.53, 4.64, 4.75, 4.86, 4.98, 5.09, 5.21, 5.34, 5.46, 5.59, 5.72, 5.86, 5.99, 6.14, 6.28, 6.43, 6.58, 6.73, 6.89, 7.05, 7.22, 7.39, 7.56, 7.74, 7.92, 8.11, 8.3, 8.5, 8.7, 8.9, 9.11, 9.33, 9.55, 9.77, 10.0]

Phénomène observé:

[2.51, 2.75, 3.71, 2.28, 2.13, 1.92, 4.28, 2.31, 3.31, 1.77, 4.32, 2.25, 5.5, 2.31, 2.83, 3.72, 2.63, 6.28, 6.08, 6.91, 6.89, 3.22, 4.29, 5.87, 4.7, 4.49, 5.24, 6.84, 8.54, 6.83, 8.94, 11.2, 7.66, 12.65, 7.52, 14.32, 13.53, 10.69, 5.93, 15.08, 18.23, 20.04, 12.13, 18.32, 21.33, 15.16, 12.49, 10.88, 22.25, 15.39, 20.57, 14.61, 30.77, 19.6, 33.46, 26.79, 41.86, 53.54, 40.62, 48.61, 64.58, 33.77, 81.54, 62.5, 79.17, 94.5, 133.09, 83.2, 83.37, 182.57, 186.02, 196.24, 123.59, 178.63, 325.9, 213.13, 446.09, 599.6, 363.89, 505.19, 909.76, 596.79, 1063.83, 1335.95, 1521.91, 787.45, 1856.59, 2884.75, 2633.02, 1564.78, 3285.46, 4193.35, 4748.44, 4035.98, 5373.65, 6705.32, 15623.23, 19650.54, 23973.92, 27795.41]

- Représenter ces données comme vu à l'exercice précédent (ne pas oublier les légendes, titre, etc).
- Émettre une hypothèse sur la dynamique du phénomène.
- Dans une autre figure représenter ces données en échelle semi-logarithmique en Y en vous inspirant de cet exemple: https://matplotlib.org/stable/gallery/scales/log_test.html
- Conclure.
- Reprendre cet exercice en plaçant les deux courbes dans la même fenêtre mais réparties en deux "subplot" empilés verticalement et en enlevant les graduations numériques sur l'axe des abscisses du graphe supérieur. Voir ici https://matplotlib.org/stable/gallery/subplots_axes_and_figures/subplots_demo.html.



Exercice 7: ajustement de données

On souhaite établir un lien entre la consommation d'une voiture et sa vitesse, pour ce faire on a répété cinq fois un trajet à différentes

vitesses. On dispose de ces données sous la forme d'une liste Python donnée ci-dessous:

cons = [1.03, 1.01, 1.03, 1.05, 1.03, 1.21, 1.25, 1.22, 1.24, 1.24, 1.68, 1.66, 1.69, 1.69, 1.62, 2.19, 2.3, 2.2, 2.36, 2.12, 3.14, 3.13, 2.94, 3.25, 3.19, 3.86, 3.95, 4.09, 4.11, 4.28, 5.01, 5.01, 4.95, 5.01, 5.17, 6.91, 6.99, 6.31, 6.14, 6.17, 8.71, 8.5, 8.96, 7.75, 8.18, 10.79, 9.68, 9.93, 10.39, 10.44, 11.04, 11.49, 11.38, 11.44, 12.0, 13.41, 15.08, 13.69, 14.04, 15.07, 15.51, 15.43, 15.58, 15.54, 17.46]

Les vitesses sont comprises entre 5 et 125 km/h, par pas de 10 kmh.

Les cinq premières valeurs de *cons* représentent les cinq consommations mesurées à 5km/h, les cinq suivantes les cinq consommations à 15 km/h, etc.

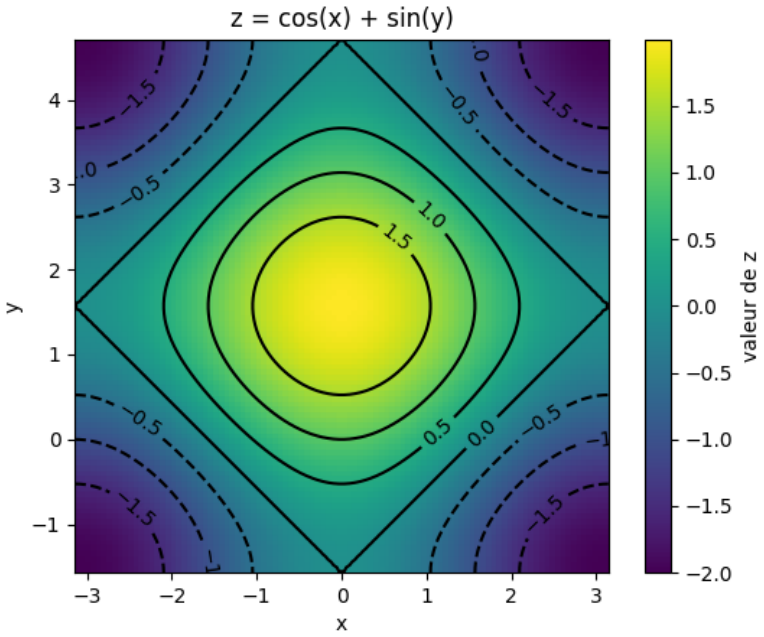
- Créer un tableau Numpy des vitesses allant de 5 à 125 (inclus) par pas de 10.
- Transformer *cons* en un array Numpy de 5 colonnes et autant de lignes qu'il y a de vitesses différentes.
- Créer un tableau Numpy nommé *cons_moy* qui contiendra la consommation moyenne pour chaque vitesse (fonction *mean*, jouer sur le paramètre *axis*).
- Tracer *cons_moy* en fonction de la vitesse avec un rond rouge sur chaque point mesuré, sans relier les points entre eux (ne pas oublier les légendes, etc).
- Grâce à la fonction *polyfit* déterminer le polynôme d'ajustement de degré 2 qui lie la consommation moyenne à la vitesse.
- Superposer ce polynôme aux valeurs moyennes tracées précédemment.

Exercice 8: représentation de surfaces en cartes de couleurs, contours

Le but de l'exercice est de représenter par une carte de couleurs $z(x, y) = cos(x) + sin(y)$ pour $x \in [-\pi, \pi]$ et $y \in [-\frac{\pi}{2}, \frac{3\pi}{2}]$

- Utiliser *np.meshgrid()* pour générer tous les couples (x, y) possibles avec un pas / nombre de points de votre choix.
- Passer ces couples à une fonction chargée de calculer Z
- Représenter le résultat avec *plt.imshow()* de matplotlib
- Bien spécifier l'étendue en X et en Y à *imshow* (mot clé extent)
- Ajouter une légende des couleurs: *plt.colorbar()*
- Superposer des contours en noir *plt.contour()*
- Ajouter un label sur ces contours: *plt.clabel()*

Il faut chercher à obtenir un résultat identique à celui-ci:

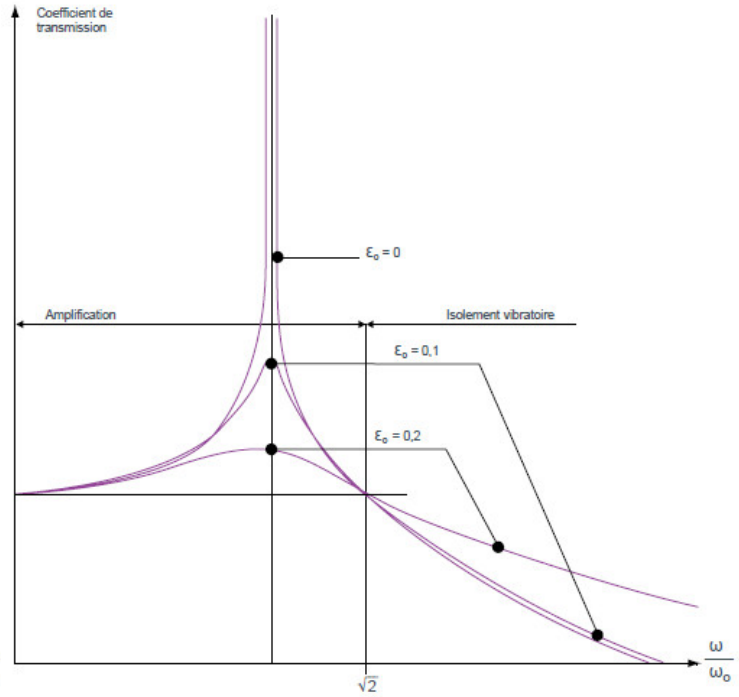


Exercice 9: flèches, annotations, formules mathématiques, etc

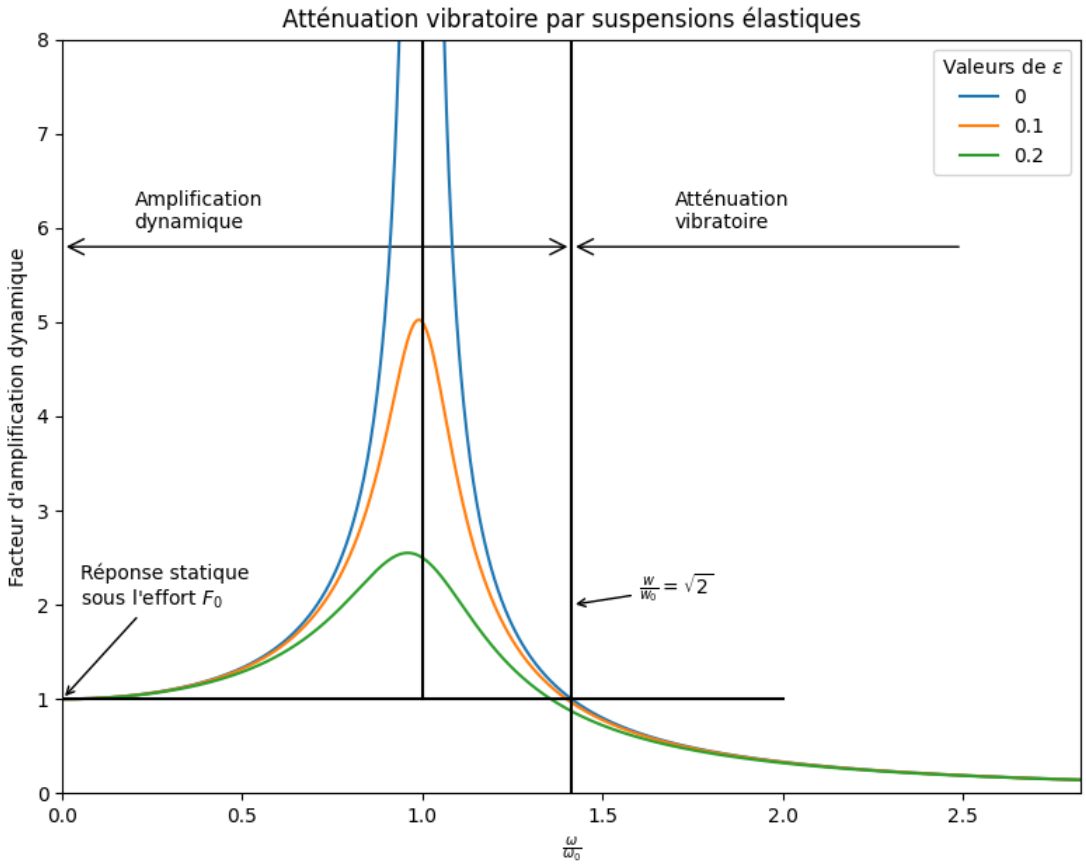
On souhaite, dans la mesure du possible, reproduire au mieux cette illustration issue d'une documentation du constructeur de solutions d'amortissements en caoutchouc Paulstra. La courbe tracée suit l'équation:

$$D(\beta, \epsilon) = \frac{1}{\sqrt{(1-\beta^2)^2 + 4 * \epsilon^2 * \beta^2}}$$

Où $\beta = \frac{w}{w0}$ et *D* représente le coefficient de transmission, ou facteur d'amplification dynamique (ça n'a pas d'importance pour la suite de l'exercice).



- Grâce à vos connaissances acquises tentez de vous rapprocher, le plus possible, de la représentation Matplotlib ci-dessous:



Indications:

- Pour écrire des formules voir ici: https://matplotlib.org/stable/tutorials/text/mathtext.html
- Pour les flèches qui pointent sur un point précis, avec éventuellement un texte associé: https://matplotlib.org/stable/tutorials/introductory/pyplot.html#annotating-text
- Pour placer du texte seul: https://matplotlib.org/stable/gallery/pyplots/pyplot_text.html#sphx-glr-gallery-pyplots-pyplot-text-py
- Pour les flèches doubles qui délimitent une zone, vous inspirer de l'exemple ci-dessous:

```
import matplotlib.pyplot as plt
import matplotlib.patches as patches
plt.figure()
ax = plt.gca()
p1 = patches.FancyArrowPatch((0.1, 0.1), (0.9, 0.9),
                             arrowstyle='<->', mutation_scale=20)
ax.add_patch(p1)
plt.show()
```

Partie 3: Exploitation de quelques millions de mesures

Préliminaire: charger rapidement des lots de données importants

Le fichier coquilles.zip contient sous forme compressée les positions de 13400 coquilles Saint Jacques enregistrées à 196 époques différentes, soit au total 2626400 enregistrements. Pensez à le dézipper une fois téléchargé (click droit, extraire sous Windows par exemple), vous devriez obtenir un fichier nommé coquilles.txt d'environ 80 Mo à placer dans le répertoire PyCharm où vous faites ce TD.

Les enregistrements sont faits selon le format suivant:

- colonne 1: identifiant coquille (int)
- colonne 2: époque d'enregistrement (int)
- colonne 3: latitude de la coquille (double)

- colonne 4: longitude de la coquille (double)
- colonne 5: stade d'évolution (int), vaut 1 ou 5.

La fonction *loadtxt* de *numpy* permet de charger simplement les fichiers texte, compressés ou non et d'allouer un tableau numpy contenant les données lues.

- Cherchez comment fonctionne *loadtxt*, vous en servir pour placer les données du fichier dans un array Numpy.
- Combien de temps faut-il pour charger le fichier fourni ? Cela vous semble-t-il acceptable ? Inspirez-vous de l'exemple suivant pour mesurer le temps passé dans une fonction:

```
import time
t_debut = time.time_ns()
print("Coucou")
t_fin = time.time_ns()
print(t_fin - t_debut)
```

- Sauvez le tableau chargé par *loadtxt* grâce à la fonction *save* de numpy (**pas** *savetxt* !). Combien de temps faut-il alors pour recharger ce tableau grâce à la fonction *load* ?

Par la suite on utilisera *load* en début de script pour charger les données préalablement converties au format interne numpy via *loadtxt* et *save*, **et on ne les convertit qu'une fois !!!** Cela permettra un gain de temps appréciable lors de la phase de mise au point du programme.

Migration de coquilles

Le but est d'effectuer toutes les opérations suivantes sans faire explicitement de boucle en Python:

- Tracer la trajectoire de la 42e coquille.
- Trouver les coquilles qui ont été en stage 5 avant le 60e enregistrement.
- Trouver le point de départ de chaque coquille.
- Pour chaque coquille, trouver le point où elle est passée en stage 5 (on pourra utiliser le fait que les coquilles passent directement du stage 1 au stage 5, ou le fait que toutes les coquilles sont simulées sur le même nombre de pas de temps).
- Trier les coquilles en fonction de leur date de passage en stage 5.
- Tracer le point de départ et le point d'arrivée de chaque coquille sous la forme d'un nuage de points.
- Bonus stage: animer le déplacement (on s'autorisera une boucle for pour grandement simplifier l'écriture, demandez de l'aide à votre encadrant !).

Modifié le: lundi 6 décembre 2021, 08:24

◄ Forum des nouvelles

Aller à...

basic_func.py ►

Connecté sous le nom « Tanguy ROUDAUT » (Déconnexion)
FIPA_outils_num

Français (fr)
English (en)
Français (fr)

Résumé de conservation de données
Obtenir l'app mobile