

# Une introduction à Python

Christophe Winisdoerffer

Version du 3 décembre 2019



# Introduction

## I Introduction

### I - 1 Objectifs

L'objectif du module est de proposer une introduction à l'utilisation des outils numériques dans le cadre de la formation "Sciences de la matière". Il s'agit donc d'écrire des programmes (des "scripts") permettant d'une part de résoudre de manière efficace des problèmes simples de physique ou de chimie, et d'autre part de visualiser les résultats obtenus (graphiques, ...). Un accent particulier est porté sur le caractère "pérenne" des codes ainsi écrits : cela impose certaines contraintes (sauvegarde sous la forme de fichiers, présence de commentaires, ...) mais permet également de les réutiliser très facilement (pour ajouter une légende à une figure, modifier la résolution d'une image, ...).

### I - 2 Langage de programmation

Le langage de programmation choisi est Python<sup>1</sup>, pour diverses raisons parmi lesquelles on peut citer le caractère libre et gratuit (contrairement à des langages propriétaires tels que MATLAB par exemple), la simplicité d'utilisation ou encore l'existence de librairies, qui contiennent "à peu près tout ce dont on a besoin" (et dont il sera fait par conséquent un usage intensif). Les différents concepts introduits durant ce module pourront être transposés le cas échéant à d'autres langages de programmation (Fortran, C, C++...) si l'utilisation de ces derniers s'avère décisive, notamment pour des aspects d'efficacité.

S'agissant d'une introduction à l'utilisation d'un tel outil numérique, il a été délibérément décidé de ne pas présenter la "programmation orientée objet" que permet le langage, ni d'aborder l'interfaçage graphique (Tkinter, GTK+, Qt...).

## II Python

### II - 1 Version

Les versions les plus notables de Python datent de 1991 (version 0.9, la toute première), 2006 (version 2.5), 2008 (versions 2.6 et 3.0), 2010 (version 2.7) et 2015 (version 3.5). Les versions 2.x étant appelées à être obsolètes, le choix naturel<sup>2</sup> s'est porté sur Python 3 (*i.e.* une version 3.x).

La commande pour lancer Python dépend de l'installation du système. Pour des raisons historiques, la commande `python` lance en général la version 2, et il faut taper `python3` pour lancer la version 3. L'option `--version` permet de savoir quelle est la version précise utilisée, cette information apparaissant également dans les lignes précédant la première invite de commande `>>>` en mode interactif.

1. [https://www.lemonde.fr/pixels/article/2018/07/25/je-n-imagineais-pas-que-python-connaitrait-un-tel-succes\\_5335917\\_4408996.html](https://www.lemonde.fr/pixels/article/2018/07/25/je-n-imagineais-pas-que-python-connaitrait-un-tel-succes_5335917_4408996.html)

2. Ce choix peut être discuté car certaines situations imposent d'utiliser Python 2, la plus commune étant qu'un code a été développé pour Python 2 et que Python 3 n'assurant pas la compatibilité descendante, il faudrait réécrire "tout" le programme. C'est notamment le cas pour la syntaxe de `print`. Avec un peu de soin, on peut cependant s'assurer, dans la plupart des cas, qu'un code fonctionne indifféremment avec Python 2 ou Python 3. À noter cependant, *Python 2.7 will reach the end of its life on January 1st, 2020. [...] Python 2.7 won't be maintained after that date.*

## II - 2 Utilisation

### II - 2.1 Mode interactif

On peut utiliser Python en mode interactif, un peu comme une calculatrice : il suffit de taper dans un terminal `python` (ou `python3`) pour disposer de l'invite de commande `>>>` à partir de laquelle l'utilisateur peut taper n'importe quelle commande Python. L'inconvénient de ce mode est que l'on perd l'ensemble des commandes tapées lorsque l'on quitte cette interface (avec la commande `quit()`, avec la combinaison de touches `Ctrl-d` ou en tuant le terminal).

On peut également lancer Python en mode interactif à l'aide de la commande `ipython` (ou `ipython3`) : les fonctionnalités sont plus développées, avec notamment la possibilité de conserver l'ensemble des commandes exécutées sous la forme d'un fichier “brut” (avec la commande `%save`) ou d'un “notebook” (avec la commande `%notebook`). Si elles ont le mérite d'exister, il est cependant très fortement déconseillé (au moins dans le cadre de ce cours) d'utiliser ces fonctionnalités pour différentes raisons, la première d'entre elles étant qu'en général, aucun commentaire ou descriptif n'accompagne la succession de commandes lorsqu'elles sont tapées en mode interactif. Sans ces informations<sup>3</sup>, il est illusoire d'espérer pouvoir réutiliser de tels scripts au-delà d'un futur (très) immédiat...

### II - 2.2 Mode non-interactif

L'ensemble des commandes est écrit et sauvegardé dans un fichier (par exemple sous le nom `monfichier.py`). Ce fichier est ensuite exécuté depuis un terminal<sup>4</sup> à l'aide de la commande `python3 monfichier.py`.

Ce mode est également accessible en exécutant la commande `python -c` suivie des instructions Python (*cf man python*), mais son utilisation est très marginale et réservée à des instructions très simples.

---

3. À l'évidence, les “notebooks” souffrent moins de ce défaut rédhibitoire, pourvu qu'ils soient développés dans un environnement propice à la rédaction de tels commentaires (`Jupyter` par exemple) ; cependant, l'expérience montre que la structure même d'un tel document ne facilite pas les étapes de correction du code.

4. On peut également exécuter un fichier depuis un mode interactif, en tapant, après l'invite de commande Python `>>>`, la commande `execfile(monfichier.py)` (ou, avec `ipython`, la commande `%run monfichier.py`). Il est cependant déconseillé de procéder ainsi, car les résultats peuvent par exemple être modifiés si l'utilisateur a défini des variables ou importé des librairies préalablement à l'exécution du fichier `monfichier.py` ; en outre, si l'utilisateur est amené à modifier le fichier tout en conservant la session interactive Python, un soin particulier doit être apporté pour que les modifications soient effectivement prises en compte.

# Prérequis

À ce stade du cours, le lecteur est sensé avoir déjà bénéficié d'une introduction à Python, et de connaître les bases de ce langage de programmation. Dans le cas contraire, il est invité à consulter l'un des très nombreux cours qu'il trouvera sans peine sur Internet. L'objectif de ce chapitre est donc de rappeler (et éventuellement préciser) quelques éléments, ainsi que de présenter les "bons réflexes" concernant l'usage "avancé" de ce langage.

## I Généralités

### I - 1 Indentation

En Python, on utilise les indentations pour structurer un code. Il est recommandé<sup>1</sup> de construire chaque indentation à l'aide de 4 espaces (même si dans ce cours, ce ne sont que 3 espaces qui sont utilisés...). À l'inverse, l'usage des tabulations est déconseillé, et le mélange de tabulations et d'espaces est à proscrire. La raison tient au fait que le caractère "Tabulation" (correspondant à la touche Tab ou à la combinaison de touches Ctrl-I) correspond à un caractère ASCII dont l'interprétation n'est pas clairement spécifiée mais paramétrable : en fonction de l'éditeur utilisé (vi, emacs, ...) et de ses préférences, ce caractère peut être affiché comme 4 espaces, ou 8, ou autre... Par conséquent, avec un certain éditeur, les différentes instructions d'un script peuvent apparaître comme correctement alignées/indentées, mais ne plus l'être sous un autre éditeur, ou si d'aventure l'utilisateur modifie les préférences de son éditeur. De façon corollaire, le lecteur qui préfère les environnements intégrés (tels que IDLE, Spyder) aux éditeurs "de base" est invité à tenir compte de leurs comportements par défaut (et de leur éventuelle personnalisation) vis-à-vis de cette problématique.

### I - 2 Structure d'un script

Comme annoncé dans l'introduction, tous les programmes seront sauvegardés dans des fichiers (avec l'extension .py). Pour les raisons décrites ci-après, tous ces programmes présenteront **TOUJOURS** la structure suivante :

```
1 #!/usr/bin/env python3
2 # -*- coding: utf-8 -*-
3 """
4     Descriptif du fichier
5 """
6
7
8 # Importation des librairies
9
10 # Definition des fonctions
11
12 # Programme principal
13 if __name__ == "__main__":
14     pass
```

La ligne 1, appelée **shebang**, est celle qui permet au système d'exploitation de "savoir" quel est l'interpréteur qu'il doit utiliser pour exécuter le fichier lorsque celui-ci est exécuté comme une commande (par ./monfichier.py ou en double-cliquant dessus) pourvu que les droits du fichier le permettent. Il est absolument nécessaire que les caractères **#!** soient les deux premiers du fichier.

La ligne 2 est celle qui permet de gérer l'utilisation éventuelle (mais très fortement déconseillée) d'accents dans

1. La PEP 8 (<https://www.python.org/dev/peps/pep-0008>) propose un ensemble de règles stylistiques pour l'écriture de scripts Python. À chacun de s'y conformer de façon plus ou moins stricte...

le fichier. Plus précisément, historiquement<sup>2</sup>, un script Python devait être composé à l'aide des 128 caractères ASCII. Cette ligne étend l'ensemble des caractères qui peuvent figurer dans le fichier à ceux (plusieurs dizaines de milliers...) du standard Unicode (le `u` de `utf-8`), avec certaines règles d'encodage (le `tf-8...`). D'autres standards<sup>3</sup> sont envisageables. Il n'en reste pas moins que seuls les caractères ASCII sont autorisés pour les noms d'"identificateurs" (variables, fonctions, modules...), et qu'on ne saurait trop insister sur les risques<sup>4</sup> auxquels on s'expose à ne pas s'en satisfaire...

La ligne 5, appelée `docstring`, correspond au descriptif du contenu du fichier ; elle est délimitée de part et d'autre par des triple-guillemets `"""`. Apparaissent ensuite les importations de librairies, les définitions des fonctions, et enfin, au sein d'une "boucle conditionnelle" (le `if __name__ == "__main__":` de la ligne 15), le "programme principal<sup>5</sup>". De cette manière, ce n'est que lorsque l'on exécute le fichier par la commande `python3 monfichier.py` (ou les équivalents en mode interactif) que Python "passe" dans la boucle `if __name__ == "__main__":`. Cela permet d'envisager d'utiliser le fichier `monfichier.py` comme une librairie additionnelle, et ainsi de pouvoir notamment réutiliser les fonctions qui y sont définies, sans qu'aucune instruction ne soit exécutée. Un exemple explicite de cette fonctionnalité est donné dans le chapitre "Librairies".

## I - 3 Identificateurs et règles de nommage

En programmation informatique, on appelle "identificateur" un ensemble de caractères qui désigne une "donnée" d'un programme : variable, fonction, module, ... Pour tous les identificateurs, seuls les caractères ASCII sont autorisés<sup>6</sup> ; en particulier, l'usage des accents dans ce contexte est interdit ! Le nom des identificateurs doit toujours commencer par une lettre<sup>7</sup>, et jamais par un chiffre. Sont interdits comme noms d'identificateurs les mots-clés du langage, dont la liste est reproduite ci-dessous :

<code>False</code>	<code>None</code>	<code>True</code>	<code>and</code>	<code>as</code>	<code>assert</code>	<code>break</code>	<code>class</code>	<code>continue</code>	<code>def</code>	<code>del</code>
<code>elif</code>	<code>else</code>	<code>except</code>	<code>finally</code>	<code>for</code>	<code>from</code>	<code>global</code>	<code>if</code>	<code>import</code>	<code>in</code>	<code>is</code>
<code>lambda</code>	<code>nonlocal</code>	<code>not</code>	<code>or</code>	<code>pass</code>	<code>raise</code>	<code>return</code>	<code>try</code>	<code>while</code>	<code>with</code>	<code>yield</code>

Le caractère `#` est également réservé pour les commentaires et ne peut donc pas apparaître dans le nom d'un identificateur, tout comme le caractère de coupure de ligne "backslash" `\`. Enfin, une évidence peut-être, qu'il vaut mieux cependant rappeler : Python est un langage sensible à la casse, et différencie par conséquent les majuscules des minuscules...

## II Variables

### II - 1 Types

Les types de base les plus communs, et supposés connus, sont,

- pour les types numériques<sup>8</sup> : les entiers relatifs (`int` ou `long`), les réels (`float`), les complexes (`complex` avec la lettre `j` ou `J` pour désigner le nombre `i` tel que  $i^2 = -1$ ) ;
- pour les types séquentiels : les chaînes de caractères<sup>9</sup> (`str`), les listes (`list`), les tuples (`tuple`).

2. Avec Python 3, le codage par défaut est `utf-8`, et cette ligne n'est donc là que pour assurer la compatibilité avec les versions antérieures de Python. On peut s'en convaincre en exécutant la commande `python3 -c 'import sys; print(sys.getdefaultencoding())'`.

3. La liste est là : <https://docs.python.org/3/library/codecs.html#standard-encodings>.

4. cf par exemple la note concernant la différence entre Python 2 et Python 3 concernant ce que recouvre le type `str` et les conséquences que cela peut avoir pour les caractères non-ASCII.

5. Pour des raisons syntaxiques, cette partie du programme doit contenir au moins une instruction (qui sera, en général, en lien avec les fonctions définies dans la première partie du script). À défaut, on peut utiliser l'instruction `pass`, qui remplit ce rôle : ne rien faire d'autre que satisfaire une exigence d'ordre syntaxique.

6. Et même plus précisément, seuls certains caractères ASCII...

7. Éventuellement par `_` mais on s'abstiendra d'utiliser cette possibilité car elle est conventionnellement réservée à des situations très particulières. C'est ainsi que `__name__` correspond au nom d'une variable qui est automatiquement définie par Python, et dont la valeur est fixée à `__main__` si cela correspond à l'exécution d'un script en tant que programme principal, ou au nom du module si cela correspond à une importation du fichier en tant que module externe.

8. C'est un abus de langage que de parler de  $\mathbb{Z}, \mathbb{R}, \mathbb{C}$  car seuls les nombres représentables sur un nombre fini de bits peuvent l'être en réalité, ce qui impose des limites minimale / maximale et des considérations de précision.

9. De fait, ce qui est désigné sous le type `str` diffère entre Python 2 (il s'agit d'une séquence de bytes) et Python 3 (il s'agit d'une réelle chaîne de caractères Unicode). Tant que l'on se restreint aux caractères ASCII, cela n'a aucune conséquence mais c'est une

C'est au moment de l'affectation d'une variable que son type est défini (on parle de "typage dynamique") ; contrairement à d'autres langages, il n'est donc pas nécessaire (en général) de définir le type des différentes variables que l'on compte utiliser.

Concernant les types numériques, les opérations mathématiques communes sont accessibles avec les symboles traditionnels  $+, -, *, /$  (qui correspond à la division entière pour Python 2 si numérateur et dénominateur sont des entiers !),  $//, \%$  (division entière et reste),  $**$  (élévation à la puissance).

Pour les types séquentiels, on accède à n'importe lequel des éléments à l'aide des crochets `[]`, le premier élément étant repéré par l'indice 0. Pour les listes et les tuples, chacun des éléments peut être de n'importe quel type (et on peut donc facilement imaginer avoir une liste de listes, ou un tuple mélangeant des chaînes de caractères et des nombres par exemple). Le tableau suivant présente différentes syntaxes d'initialisation.

	<code>type(x)</code>		<code>type(x)</code>		<code>type(x)</code>
<code>x = 1</code>	<code>int</code>	<code>x = 'possible'</code>	<code>str</code>	<code>x = [0, 1, 2]</code>	<code>list</code>
<code>x = 1.</code>	<code>float</code>	<code>x = "c'est possible"</code>	<code>str</code>	<code>x = (0, 1, 2)</code>	<code>tuple</code>
<code>x = 1+1j</code>	<code>complex</code>			<code>x = 0, 1, 2</code>	<code>tuple</code>
<code>x = complex(1,1)</code>	<code>complex</code>				

Il existe bien entendu d'autres types (booléen, dictionnaire, ...), dont on fera usage à l'occasion.

## II - 2 Objets et méthodes

Un exemple étant plus parlant qu'un long discours, considérons un problème d'électromagnétisme pour lequel on est amené à considérer différents champs scalaires ( $\rho$  ou  $V$ ) ou vectoriels ( $\mathbf{E}, \mathbf{B}$ , ...). Dès que ces champs sont définis, il est licite de leur associer d'autres champs, tels que le gradient (pour les champs scalaires), la divergence ou le rotationnel (pour les champs vectoriels). On peut bien entendu facilement imaginer écrire les fonctions correspondantes. Mais on peut également envisager de "cacher" ces opérations dans la définition même de ces champs : dès lors que la variable `V` est identifiée comme un champ scalaire, `V.grad()` désigne le gradient correspondant (tout comme `E.div()` désigne la divergence du champ vectoriel `E`) ; en quelque sorte, l'objet "champ scalaire" (resp. "champ vectoriel") arrive avec la méthode `.grad` (resp. `.div`) permettant d'en déterminer le gradient (resp. la divergence). En d'autres termes, une variable, dès que son type est défini, bénéficie d'un certain environnement grâce auquel certaines propriétés ou opérations sont directement accessibles par l'appel aux méthodes correspondantes (par la commande générique `lavariable.lamethode()`). Même s'il a été décidé de ne pas mettre l'accent dans ce cours sur ces aspects relevant de la "programmation orientée objet", il est impossible de ne pas l'évoquer, car en Python, tout est objet<sup>10</sup> (variables, fonctions, modules, ...) ; par conséquent, il sera très fréquent de disposer, pour les types les plus usuels, d'une méthode équivalente à l'appel d'une fonction.

## II - 3 Un point "technique" : caractère mutable ou immutable

À ce stade de la présentation, il est important de faire une digression technique. Considérons ainsi l'instruction suivante : `pi = 3.14`. Dans cette expression, `pi` désigne le nom d'une variable, et `3.14` la valeur qui lui est affectée. D'un point de vue informatique, c'est une opération relativement complexe qui doit être effectuée, car l'ordinateur doit associer au mot "pi" une adresse dans la mémoire, adresse "dans" laquelle il doit écrire non seulement la valeur "3.14", mais également le fait qu'il s'agit d'un réel (un `float`), et qu'à ce réel est attaché différentes propriétés et fonctionnalités (les aspects "objet" du type `float`). Le mot "pi" est donc la référence "humainement lisible" qui pointe sur cet objet. La question est maintenant de savoir ce à quoi correspond l'instruction suivante : `Pi = pi` (on rappelle que Python est sensible à la casse...) : Python crée-t'il un nouvel

---

toute autre affaire sinon. Les plus curieux pourront s'en convaincre en créant tout d'abord un fichier contenant le mot "chinois" en chinois par `python -c "with open('chinois.txt', 'wb') as f: f.write(b'\xe4\xb8\xad\x9b\xbd')"`. Ce fichier contient deux caractères (`cf` le résultat de la commande `cat chinois.txt`, pourvu que le terminal supporte l'encodage utf-8 ce qui devrait être le cas pour une version de Linux à jour...) mais 6 bytes (`cf ls -l chinois.txt`) ; il suffit alors de relire le contenu de ce fichier à l'aide de la commande `python -c "with open('chinois.txt') as f: x = f.read(); print(type(x), len(x))"` et de comparer ce que retourne la même commande avec Python 3. L'utilisation de caractères non-ASCII, en particulier avec des écritures et lectures de fichiers, peut ainsi rapidement tourner au cauchemar...

10. Remarque pour les étudiants les plus avertis : le fait d'attacher une méthode au type d'un objet (on parle d'encapsulation) facilite la portabilité d'un programme. Il suffit pour s'en convaincre de considérer le laplacien, dont le sens diffère selon le caractère scalaire ou vectoriel du champ. La POO et l'utilisation de méthodes encapsulées dans les différents objets permettent de rendre le processus "transparent", sans avoir à écrire différentes fonctions (telles que `laplacien.scal` et `laplacien.vect`) ou faire des tests sur les types des variables.

“objet informatique”, avec un autre emplacement dans la mémoire dans laquelle il recopie (en particulier) la valeur 3.14, ou seulement une sorte d’alias, le mot “Pi” désignant uniquement une autre référence de l’objet créé avec l’instruction `pi = 3.14`? La question est loin d’être anodine car c’est elle qui va conditionner la valeur de la variable `pi` à la fin de la séquence `pi = 3.14 ; Pi = pi ; Pi = 3.14159`: la valeur de `pi` est-elle toujours 3.14 ou désormais 3.14159? La réponse dépend du caractère “mutable” ou “immutable” de l’objet. En l’occurrence, les variables de type numérique sont immutables, et dans notre exemple, `pi` vaut toujours 3.14. Et si “Pi” référençait bien le même objet que “`pi`” après l’instruction `Pi = pi`, c’est devenu la référence d’un nouvel objet<sup>11</sup> (avec un autre emplacement mémoire) lorsqu’on lui a affecté une nouvelle valeur. Tout comme les variables de type numérique, les chaînes de caractères ainsi que les tuples sont immutables; par conséquent, après création de ces objets, leurs éléments ne peuvent plus être modifiés. À l’inverse, les listes (ainsi que les dictionnaires) sont mutables. *Mutatis mutandis*, aucun lecteur ne devrait plus être surpris par le résultat de la séquence `a = [0, 1] ; b = a ; b[0] = -1 ; print(a)`. Si tel n’est pas le comportement souhaité, il faudra veiller à utiliser les fonctions `copy` ou `deepcopy` du module `copy` lors de l’affectation des variables mutables, ou recourir, le cas échéant, à des notations propres à certains types (par exemple le slicing `b = a[:]` pour que `b` et `a` ne réfèrent pas le même objet).

## III Éléments de programmation

### III - 1 Boucle conditionnelle

La syntaxe d’une boucle conditionnelle est la suivante :

```

1 if condition_1:
2     ... # do something
3 elif condition_2:
4     ... # do something_else
5 else:
6     ... # do default

```

Il peut y avoir aucune ou plusieurs sections `elif`, et la section `else` est optionnelle. Les conditions à satisfaire sont de type booléen; elles sont construites à l’aide d’opérateurs de comparaison (`<`, `>`, `<=`, `>=`, `==`, `!=`) ou de tests élémentaires (`is`, `in`, `isinstance`, `isnumeric`...), que l’on peut combiner à l’aide des opérateurs `or`, `and` ou `not` (avec les règles de priorité usuelles). Pour toute variable de type numérique, 0 est considéré comme `False`; pour toute variable de type séquentiel, une séquence vide est considérée comme `False`. `None` est également considéré comme `False`.

À noter la possibilité parfois pratique de la syntaxe “sur une ligne” du type `x = (x1 if cond else x2)`, les parenthèses étant optionnelles (mais recommandées pour faciliter la lecture).

### III - 2 Boucle itérative

La syntaxe d’une boucle itérative est la suivante :

```

1 for element in sequence:
2     ... # do something

```

Il est important de réaliser qu’en Python, c’est directement sur les éléments de la séquence que s’effectue l’itération, et qu’il n’y a pas (forcément) besoin par conséquent de recourir à des nombres (qui seraient, par exemple, les indices des différents éléments de la séquence). L’exemple ci-dessous illustre le propos.

```

1 import string
2 minuscules = string.ascii_lowercase
3 # la bonne façon de faire
4 for lettre in minuscules:
5     print(lettre)
# la mauvaise façon de faire
# for i in range(len(minuscules)):
#     print(minuscules[i])

```

11. Les plus suspicieux des lecteurs pourront s’en convaincre en utilisant la fonction `id(var)` qui renvoie l’identificateur (unique) de l’objet `var`. À noter que l’expression `a is b` teste si l’objet `a` désigne le même objet que `b`, alors que l’expression `a == b` teste si les valeurs des variables `a` et `b` sont égales.

L'instruction `break` permet de sortir de la (plus petite) boucle `for` dans laquelle elle apparaît. L'instruction `continue` permet de passer à l'itération suivante.

À ce stade de la présentation, et bien que cette pratique ne soit pas recommandée en général, il faut également mentionner la possibilité de construire une boucle itérative à l'aide de l'instruction `while` (combinée à (au moins) une instruction `break`). Une des difficultés de ce type de boucle est de s'assurer de l'exécution de l'instruction `break` au bout d'un nombre fini d'itérations.

### III - 3 Fonctions

#### III - 3.1 Syntaxe

Une fonction se déclare à l'aide du mot-clé `def`. Le nom de la fonction est suivi par des parenthèses () entre lesquelles figurent la liste des arguments (éventuellement vide). Même si la norme ne l'impose pas, une fonction se termine par l'instruction `return` suivie de l'ensemble des résultats qu'elle renvoie vers le programme appelant, ensemble éventuellement vide auquel cas la fonction retourne la valeur `None`. Elle contient bien entendu une docstring<sup>12</sup> correspondant au descriptif de son utilité, et de ses “effets de bord” éventuels (*cf* ci-dessous).

#### III - 3.2 Variables locales et globales

On rappelle que les variables définies au sein d'une fonction ont une “portée locale”, *i.e.* sont automatiquement détruites dès lors que l'on quitte la fonction ; il est donc parfaitement licite d'utiliser le même nom pour une variable dans deux fonctions différentes. À l'inverse, et même si la norme ne l'impose pas, il est **ABSOLUMENT INTERDIT** de faire usage, au sein d'une fonction, de variables qui ne lui auraient pas été passées comme arguments. Ne pas respecter cette règle<sup>13</sup>, et c'est courir à la catastrophe de façon certaine, à plus ou moins longue échéance.

#### III - 3.3 Passage d'arguments

En Python, le passage des arguments se fait par référence (et non pas par valeur). Cela à des conséquences extrêmement importantes, qu'il faut avoir à l'esprit. Tout dépend du caractère mutable ou immuable des arguments passés à la fonction. Pour éclairer le propos, considérons la fonction `func(x)` au sein de laquelle apparaît l'instruction `x *= 2`. Si `x` désigne une variable immuable (et dont la valeur ne peut donc pas être modifiée après création), par exemple un nombre, alors Python crée au moment de l'exécution de cette instruction un nouvel objet ; ce nouvel objet est référencé, *au sein de la fonction* par le nom “`x`” et a comme valeur le double attendu, mais il s'agit bien d'un autre objet que celui qui correspond à l'argument passé à la fonction. Dès la sortie de la fonction, cet objet est détruit (car il a été créé au sein de la fonction), et la valeur originelle de l'objet désigné par “`x`” dans le programme appelant n'est pas modifiée (car l'objet est immuable!). Si tel n'est pas l'objectif recherché, il faut que la fonction retourne la valeur calculée et que celle-ci soit affectée à `x` dans le programme appelant par l'instruction `x = func(x)`. À l'inverse, toujours avec la même fonction, si `x` désigne une variable mutable, par exemple la liste `[0, 1]`, alors Python modifie la valeur de l'objet référencé au moment de l'exécution de l'instruction `x *= 2` (sans avoir à créer de nouvel objet puisque celui-ci est mutable). À la sortie de la fonction, cet objet n'est pas détruit (car il n'a pas été créé dans la fonction mais au niveau du programme appelant), et la modification de la valeur est donc conservée, bien que nulle part ne figure une instruction du type `x = func(x)` ! C'est ce qu'on appelle un “effet de bord”. Si tel n'est pas l'objectif recherché, il ne faut pas travailler dans la fonction sur `x` mais sur une copie de `x`. Soulignons la nécessité de bien documenter ce type d'effets de bord (dans la docstring de la fonction) car bien souvent, ils sont à l'origine de bugs difficiles à détecter (et donc à corriger). Le script ci-dessous illustre ce type de “désagrément”.

---

12. Comme tout en Python, une fonction est un objet et possède donc certaines propriétés (par exemple un identificateur accessible par la fonction `id(func)` si `func` désigne le nom de la fonction) ou fonctionnalités. Ainsi, on peut accéder à la docstring de la fonction par `print(func.__doc__)`

13. En vérité, Python n'impose en rien cette règle : au sein d'une fonction, le programme a accès à l'ensemble des données disponibles dans l'espace de nommage (*i.e.* à l'ensemble des objets qui ont été créés depuis le lancement du script et pas encore détruits), et le mot-clé `global` permet de ne pas détruire une variable à la sortie d'une fonction. Mais on ne saurait souligner suffisamment la dangerosité extrême de vouloir profiter de ces possibilités.

```

1 #!/usr/bin/env python3
2 # -*- coding: utf-8 -*-
3
4 """
5 Illustration des effets de bord
6 """
7
8 # Importation des librairies
9
10 # Definition des fonctions
11 def f1(x):
12     """ Docstring à rediger """
13     x *= 2
14     print("Valeur de x au sein de la fonction: {}".format(x))
15     return
16
17 # Programme principal
18 if __name__ == "__main__":
19     # Definition de quelques paramètres
20     x_num = 1
21     x_lst = [1]
22     # Appel de fonctions
23     # --> x_num est immutable
24     print("Valeur de x_num dans le main avant appel: {}".format(x_num))
25     res = f1(x_num)
26     print("Valeur de x_num dans le main après appel: {}".format(x_num))
27     # --> x_lst est mutable
28     print("Valeur de x_lst dans le main avant appel: {}".format(x_lst))
29     res = f1(x_lst)
30     print("Valeur de x_lst dans le main après appel: {}".format(x_lst))
31     print("Valeur de x_lst dans le main avant appel: {}".format(x_lst))
32     res = f1(x_lst[:])
33     print("Valeur de x_lst dans le main après appel: {}".format(x_lst))

```

### III - 3.4 Arguments et arguments optionnels

Avec Python, on dispose de la possibilité extrêmement utile de pouvoir définir des fonctions dont un (ou plusieurs) arguments sont optionnels. À ce stade, le lecteur est supposé être familier avec les arguments “traditionnels”, qualifiés de positionnels et qui correspondent à une fonction du type `def func(x,y)` que l’on appelle à l’aide de l’instruction `res = func(a,b)`. Python permet de définir une fonction admettant un (ou plusieurs) argument(s) optionnel(s) au(x)quel(s) on accède *via* le(s) nom(s) qui lui (leur) est (sont) attribué(s). La valeur par défaut à considérer pour chaque argument optionnel est précisé dans la définition de la fonction. Ainsi, on peut définir une fonction `def func(x,y,z=0)` que l’on peut tout aussi bien appeler à l’aide de l’instruction `res = func(a,b)` (auquel cas `z`, au sein de la fonction, prend la valeur définie par défaut) que par `res = func(a,b,z=c)` (avec bien entendu `z` qui vaut `c` dans ce cas). La seule obligation est de placer les arguments positionnels avant les arguments nommés. Le script ci-dessous illustre quelques unes de ces possibilités.

```

1 #!/usr/bin/env python3
2 # -*- coding: utf-8 -*-
3
4 """
5 Définitions basiques de fonctions
6 """
7
8 # Importation des librairies
9
10 # Definition des fonctions
11 def f1(x,y):
12     """ 2 arguments obligatoires """

```

```

13     res = x+y
14     return res
15
16 def f2(x,y,p=1,q=0):
17     """ 2 arguments obligatoires, 2 optionnels """
18     res = (x+y)*(p+q)
19     return res
20
21 # Programme principal
22 if __name__ == "__main__":
23     # Definition de quelques parametres
24     a = 1
25     b = 2
26     c = 3
27     d = 4
28     # Appel de fonctions
29     # Basique
30     res = f1(a,b)          ; print('call f1 #1: {}'.format(res))
31     # Sans les arguments optionnels
32     res = f2(a,b)          ; print('call f2 #1: {}'.format(res))
33     # Avec les arguments optionnels ; preferable a f2(a,b,c,d)
34     res = f2(a,b,p=c,q=d) ; print('call f2 #2: {}'.format(res))
35     # Avec uniquement certains arguments optionnels
36     res = f2(a,b,q=0.5)   ; print('call f2 #3: {}'.format(res))

```

Reste à dévoiler un piège courant concernant l'utilisation d'arguments optionnels, et plus précisément la définition des valeurs par défaut à l'aide d'objets mutables : c'est lors de la définition de la fonction que ces valeurs sont évaluées et non lors de l'appel de la fonction. Cela peut induire des comportements contraires à ce qui est souhaité si l'on perd de vue le caractère mutable de certains types de variables. La règle consiste à ne pas initialiser des arguments optionnels à l'aide d'objet mutable mais à utiliser le mot-clé `None` pour tester l'éventuelle présence de cet argument lors de l'appel de la fonction. Cela est illustré sur le script ci-dessous.

```

1 #!/usr/bin/env python3
2 # -*- coding: utf-8 -*-
3
4 """
5 Illustration du piege concernant l'initialisation d'arguments optionnels a l'aide
6 d'un objet mutable
7 """
8
9 # Importation des librairies
10
11 # Definition des fonctions
12 def f1(x,y,lst=[]):
13     """ La mauvaise facon de faire: la valeur par defaut est evallee lors de la
14     definition de la fonction (et non pas son appel), et comme elle se refere a
15     un objet mutable, elle est 'conservee en memoire' """
16     lst.append(x+y+len(lst))
17     return lst
18
19 def f2(x,y,lst=None):
20     """ La bonne facon de faire: definir au sein de la fonction la valeur
21     correspondant a un objet mutable, qui sera donc perdu a la sortie de la
22     fonction """
23     if lst is None:
24         lst = []
25     lst.append(x+y+len(lst))
26     return lst
27
28 # Programme principal
29 if __name__ == "__main__":
30     # Definition de quelques parametres
31     a = 1

```

```

32     b = 2
33 # Appel de fonctions
34 res = f1(a,b)          ; print('call f1 #1: {}'.format(res))
35 res = f1(a,b)          ; print('call f1 #2: {}'.format(res))
36 res = f1(a,b,lst=res) ; print('call f1 #3: {}'.format(res))
37 res = f2(a,b)          ; print('call f2 #1: {}'.format(res))
38 res = f2(a,b)          ; print('call f2 #2: {}'.format(res))
39 res = f2(a,b,lst=res) ; print('call f2 #3: {}'.format(res))

```

Pour la plupart des applications envisagées au niveau de ce cours, cette présentation des arguments positionnels (obligatoires) et nommés (optionnels) suffit. Néanmoins, compte tenu du fait que l'un des objectifs est de convaincre le lecteur de l'intérêt d'exploiter les fonctionnalités offertes dans les librairies tierces, il est nécessaire de présenter une autre possibilité offerte par Python, qui apparaît dans les documentations de très nombreuses fonctions. Il s'agit de la possibilité de passer un ensemble d'arguments optionnels non nommés à l'aide de la syntaxe `def func(x,y,*args)` ainsi qu'un ensemble d'arguments optionnels nommés si l'on définit (traditionnellement) la fonction par `def func(x,y,**kwargs)` (le `kw` faisant référence au fait que les arguments sont associés à des mots-clés –keywords–). Le cas échéant, cela permet d'enrichir les fonctionnalités de la fonction. Le script ci-dessous illustre quelques unes de ces possibilités.

```

1 #!/usr/bin/env python3
2 # -*- coding: utf-8 -*-
3
4 """
5 Definitions avancees de fonctions
6 """
7
8 # Importation des librairies
9
10 # Definition des fonctions
11 def f1(x,y,*args):
12     """ 2 arg. obligatoires, n'importe quel nombre d'arg. optionnels """
13     if args:
14         res = (x+y)*len(args)
15     else:
16         res = x+y
17     return res
18
19 def f2(x,y,**kwargs):
20     """ 2 arg. obligatoires, n'importe quel nombre d'arg. optionnels nommes """
21     res = (x+y)
22     if kwargs:
23         facteur = 0
24         for key in kwargs.keys():
25             facteur += kwargs[key]
26         res *= facteur
27     return res
28
29 # Programme principal
30 if __name__ == "__main__":
31     # Definition de quelques parametres
32     a = 1
33     b = 2
34     c = 3
35     d = 4
36     # Appel de fonctions
37     res = f1(a,b)          ; print('call f1 #1: {}'.format(res))
38     res = f1(a,b,c,d,a+b) ; print('call f1 #2: {}'.format(res))
39     res = f2(a,b)          ; print('call f2 #1: {}'.format(res))
40     res = f2(a,b,p=c,q=d) ; print('call f2 #2: {}'.format(res))
41     res = f2(a,b,q=0.5)   ; print('call f2 #3: {}'.format(res))

```

## IV I/O

### IV - 1 Interaction avec l'utilisateur

Au cours de l'exécution d'un programme, l'utilisateur peut accéder à l'ensemble des "données" manipulées<sup>14</sup> par le script et les afficher sur la sortie standard (l'écran) à l'aide de la commande `print`. Il est donc extrêmement facile d'accéder non seulement à la valeur d'une variable, mais également à son type ou à toute autre de ses caractéristiques (en particulier à l'aide des méthodes attachées à cet objet), et c'est "l'outil-roi" dont on usera et abusera lors de la phase de debugage du programme. En Python 3<sup>15</sup>, `print` correspond à une fonction, et comme toute fonction, ce qui doit être affiché est passé comme argument<sup>16</sup> de la fonction ; on écrira ainsi la commande `print("Hello World")` (pour afficher la chaîne de caractères "Hello World"), `print(type(x), x)` (pour afficher le type et la valeur de la variable `x`), `print(func.__doc__)` (pour afficher la docstring de la fonction `func`).

Il est également possible que le programme réceptionne au cours de son exécution une donnée fournie (au clavier) par l'utilisateur, à l'aide de la fonction `raw_input` mais il ne sera pas fait usage de cette fonctionnalité dans ce cours. Enfin, comme toute autre commande Linux, la commande `python monscript.py` peut être suivie d'autres arguments, ce qui permet de passer des données au script au lancement de son exécution. La récupération des arguments (considérés comme des chaînes de caractères, avec un (ou plusieurs) espace(s) comme séparateur) se fait via l'utilisation du module `sys`, et plus précisément de la variable `sys.argv` (avec `sys.argv[0]` qui contient le nom du script, ici `monscript.py`).

### IV - 2 Mise en forme

Par défaut, l'utilisation de la commande `print` laisse Python mettre en forme la (ou les) donnée(s) affichée(s). Il est souvent judicieux, pour faciliter la lecture, de soigner la mise en forme. Pour cela, on utilise la méthode `format` disponible pour les chaînes de caractères ; l'idée est de construire une chaîne de caractères (entre guillemets) dans laquelle figure(nt) une (ou plusieurs) paire(s) d'accolades `{}` au sein desquelles seront "recopiées" les valeurs des arguments de la méthode `format`. Le formatage désiré (tout comme éventuellement l'ordre dans lequel doivent être traités les arguments de la méthode `format`) est précisé au sein de chaque paire d'accolades. Plutôt qu'un long discours, les exemples ci-dessous recouvrent les situations les plus communes<sup>17</sup>.

Argument de la fonction <code>print</code>	Résultat
<code>"{} {}".format('a', 3.14)</code>	a 3.14
<code>"{1} {0}".format('a', 3.14)</code>	3.14 a
<code>"{:&lt;10}".format('blabla')</code>	blabla justifié à gauche sur (au moins) 10 caractères
<code>"{:&gt;10}".format('blabla')</code>	blabla justifié à droite sur (au moins) 10 caractères
<code>"{:d}".format(1)</code>	1 integer
<code>"{:3d}".format(1)</code>	1 ... sur (au moins) 3 caractères
<code>"{:03d}".format(1)</code>	001 ... en complétant par des zéros si nécessaire
<code>"{:f}".format(3.14)</code>	3.140000 fixed-point number, 6 chiffres après la virgule (par défaut)
<code>"{:.3f}".format(3.14)</code>	3.140 ... 3 chiffres après la virgule
<code>"{:10.3f}".format(3.14)</code>	3.140 ... sur (au moins) 10 caractères
<code>"{:e}".format(3.14)</code>	3.140000e+00 notation scientifique, 6 chiffres après la virgule (par défaut)
<code>"{:.3e}".format(3.14)</code>	3.140e+00 ... 3 chiffres après la virgule
<code>"{:10.3e}".format(3.14)</code>	3.140e+00 ... sur (au moins) 10 caractères

14. À plus proprement parler, l'utilisateur a accès à l'ensemble des données disponibles dans "l'espace de nommage" au moment où apparaît la commande `print`.

15. En Python 2, `print` est une instruction, et ne retourne donc rien ; les syntaxes autorisées sont `print("Hello World")` (comme en Python 3), mais également `print "Hello World"`.

16. On notera au passage que cette fonction admet un nombre quelconque d'arguments non-nommés, cf la possibilité offerte par `*args`.

17. Pour plus de détails, se référer à la documentation officielle <https://docs.python.org/3/library/string.html>.

## IV - 3 Fichiers

L'interaction de Python avec un fichier se passe en deux étapes : la première est la création d'un objet de type "fichier", la seconde est l'utilisation de méthodes associées à ce type d'objet pour manipuler le contenu du fichier. Pour créer l'objet de type "fichier", on utilise la fonction `open()` avec un argument obligatoire, le nom du fichier (sous la forme d'une chaîne de caractères), et un argument optionnel mais presque toujours précisé, qui correspond à "l'activité" envisagée ('r' pour uniquement lire le fichier, 'w' pour écrire ce fichier (quitte à écraser le contenu précédent le cas échéant), 'a' pour écrire à la fin du fichier, '+' pour lire et écrire, ...). Par défaut, il s'agit d'un fichier "texte", humainement lisible, mais on peut envisager d'interagir avec un fichier binaire (en associant la lettre `b` à l'activité envisagée). Pour diverses raisons, cette opération sera associée au mot-clé `with` tel qu'illustré ci-dessous. La manipulation des données repose alors sur l'utilisation des méthodes correspondantes `write`, `read`, `readline`, `readlines`. Le script suivant illustre les différentes possibilités.

```

1 #!/usr/bin/env python3
2 # -*- coding: utf-8 -*-
3
4 """
5 Manipulation de fichiers
6 """
7
8 # Importation des librairies
9
10 # Definition des fonctions
11 def create_file(fname):
12     """ Creation du fichier fname """
13     with open(fname, 'w') as f: # Rq: leaving the loop automatically close the file
14         f.write('Ligne 1\n')    # 8 caracteres, le \n comptant pour 1
15         f.write('Ligne 2')
16         f.write(" ... qui n'est pas finie!\n") # importance du \n !!!
17         f.write('Ligne 3\n')
18         f.write(f.name)        # une des proprietes de l'objet f, son nom...
19     return
20
21 def read_file(fname):
22     """ Diverses facons de lire le fichier fname """
23     # 1ere facon: lit la totalite du fichier
24     #           retourne une unique chaine de caracteres
25     with open(fname, 'r') as f: # Rq: leaving the loop automatically close the file
26         txt = f.read()
27         print("Premiere facon", type(txt), len(txt))
28     # 2eme facon: lit la totalite du fichier
29     #           retourne une liste de chaines correspondant a chaque ligne
30     with open(fname, 'r') as f: # Rq: leaving the loop automatically close the file
31         txt = f.readlines()
32         print("Seconde facon", type(txt), len(txt))
33     # 3eme facon: lit le fichier ligne a ligne et retourne la chaine correspondante
34     #           un peu penible pour detecter la derniere ligne...
35     with open(fname, 'r') as f: # Rq: leaving the loop automatically close the file
36         txt = f.readline()
37         print("Troisieme facon", type(txt), len(txt))
38         txt = f.readline()
39         print("Troisieme facon", type(txt), len(txt))
40     # 4eme facon: lit le fichier ligne a ligne et retourne la chaine correspondante
41     #           --> la BONNE methode pour lire le fichier jusqu'au bout sans tout stocker!
42     with open(fname, 'r') as f: # Rq: leaving the loop automatically close the file
43         for line in f:
44             print("Quatrieme facon", type(line), len(line))
45     #
46     return
47
48
49 # Programme principal
50 if __name__ == "__main__":

```

```

51 # Definition du nom du fichier
52 fname = 'toto.txt'
53 # Creation du fichier
54 create_file(fname)
55 # Lecture
56 read_file(fname)

```

## V Pour conclure : quelques règles très générales de programmation

L'erreur étant humaine, écrire un programme ne va pas sans trouver (et corriger) les différentes erreurs qui peuvent apparaître ; celles-ci peuvent être classées en trois catégories : les erreurs de syntaxe, qui empêchent le code de s'exécuter et qui sont en général faciles à corriger (il "suffit" de lire et comprendre le message d'erreur qui s'affiche) ; les erreurs de "logique", qui correspondent au fait que le code est syntaxiquement correct (et s'exécute donc) mais le résultat n'est pas celui attendu ; cela tient au fait que le code fait exactement ce que vous lui avez dit de faire, mais qui n'est pas ce que vous espériez qu'il fasse ! Cette phase de debugage, parfois longue et difficile<sup>18</sup>, nécessite de se convaincre, à chaque étape du code, que le comportement est celui attendu (type des variables, valeurs, ...) jusqu'à identifier "là où ça bugue", et pour cela, rien de mieux qu'afficher toutes ces informations au cours de l'exécution (avec la commande `print!`). Le troisième type d'erreurs, dites "Run-time error" correspondent à des erreurs qui peuvent apparaître dans des circonstances particulières, et la gestion de ces "exceptions" ne sera (essentiellement) pas abordée dans ce cours, la complexité des programmes envisagés ne l'imposant pas.

Que ce soit pour faciliter ce travail de debugage ou pour permettre une utilisation postérieure des différents scripts, on n'insistera jamais assez sur la nécessité absolue de commenter son code, à l'aide de docstrings (entre triples ou simples guillemets ") ou de commentaires "bruts", initiés par le caractère `#`. Il est évident qu'il faut également favoriser la lisibilité du code et, même si le langage le permet, éviter par exemple d'imbriquer des boucles dans une même instruction voire tout écrire sur une seule et même ligne... Aux aphorismes<sup>19</sup> de la PEP 20 "Le Zen du Python", on pourra préférer, bien qu'anachronique, l'art poétique de Boileau.

*Réant donc que d'écrire, apprenez à penser.*

...

*Hâtez-vous lentement, et, sans perdre courage,  
Vingt fois sur le métier remettez votre ouvrage :  
Polissez-le sans cesse et le repolissez ;  
Ajoutez quelquefois, et souvent effacez.*

18. Citation de Brian Kernighan à méditer : *Debugging is twice as hard as writing the code in the first place. Therefore, if you write the code as cleverly as possible, you are, by definition, not smart enough to debug it.*

19. <https://www.python.org/dev/peps/pep-0020/>



# Librairies : généralités

## I Introduction

En plus de la “librairie standard”, celle qui définit les types “standards”, les méthodes associées, certaines fonctions et qui est automatiquement “chargée” lorsqu’on lance Python, il est possible d’utiliser des fonctionnalités supplémentaires présentes dans d’autres librairies<sup>1</sup>. La richesse de ces librairies constitue l’un des avantages évident de Python sur d’autres langages de programmation. Parmi celles-ci, on peut citer :

- la librairie `sys` : “provides access to some variables used or maintained by the interpreter and to functions that interact strongly with the interpreter” ; permet en particulier de récupérer les éventuels arguments de la commande `python3 monfichier.py arg1 arg2 ...` ;
- la librairie `os` : “provides a portable way of using operating system dependent functionality” ; permet en particulier d’exécuter des commandes système depuis un script Python ;
- la librairie `time` : “provides various time-related functions” ; permet (entre autres) de manipuler aisément ce qui correspond à des aspects temporels (date, heure, durée, ...);
- la librairie `numpy` : “is the fundamental package for scientific computing with Python” ; rend en particulier sans (grand) intérêt l’utilisation de la librairie `math` ;
- la librairie `scipy` : “provides many user-friendly and efficient numerical routines such as routines for numerical integration and optimization” ;
- la librairie `matplotlib` : “is a plotting library for the Python programming language and its numerical mathematics extension NumPy” ;
- ...

## II Utilisation

Pour utiliser une librairie, il faut dans un premier temps “l’importer” avec la commande `import lalibrairie`. On dispose d’une aide en ligne avec la commande `help(lalibrairie)` (qui ressemble beaucoup à un `man` de Linux), et d’un listing du contenu avec la commande `dir(lalibrairie)`. La fonction<sup>2</sup> `lafonction` de la librairie `lalibrairie` peut alors être invoquée avec la commande `lalibrairie.lafonction(...)`. Python offre la possibilité de modifier le nom de la librairie à l’aide du mot clé `as`, ce qui permet d’utiliser des alias pour simplifier la syntaxe. Le listing ci-dessous illustre ces différentes commandes.

```
1 #!/usr/bin/env python3
2 # -*- coding: utf-8 -*-
3
4 """
5 Illustration des importations de librairies
6 """
7
8 # Importation des librairies
9 import sys                      # importation de la lib. sys
10 import numpy as np               # importation de la lib. numpy avec l'alias np
11 import matplotlib.pyplot as plt  # importation de la sous-lib. matplotlib.pyplot
12                                         # avec l'alias plt
13
14 # Definition des fonctions
15
16 # Programme principal
```

1. Techniquelement, ces librairies, également appelées *packages*, sont une façon d’organiser et hiérarchiser des “objets” plus élémentaires, appelés *modules*, qui sont des scripts contenant des définitions, des algorithmes, des interfaces avec d’autres librairies (éventuellement implémentées dans un autre langage que Python!)... En pratique, l’importation de librairies ou de modules est essentiellement identique, et l’on ne fera pas de distinction entre les deux au niveau de ce cours.

2. En fait, n’importe quel “objet” de la librairie, mais pour la clareté de l’exposé, on se restreint ici à parler de “fonction”

```

17 if __name__ == "__main__":
18     # Affiche les répertoires visites dans lesquels doivent se trouver les lib.
19     # que l'on cherche à importer
20     print(sys.path)
21     # Affiche le contenu de la lib. numpy, désormais accessible sous l'alias np
22     print(dir(np))
23     # Affiche l'aide sur la fonction show de la lib. plt (= help(plt.show()))
24     print(plt.show.func_doc)

```

On peut importer<sup>3</sup> un fichier local (pour utiliser par exemple les fonctions qui y sont définies) avec la commande `import monfichier`, pourvu que ce fichier s'appelle `monfichier.py` (l'extension `.py` est cruciale) et qu'il se trouve soit dans le répertoire local où est exécuté le script, soit dans le “PYTHONPATH” (que l'on peut modifier à souhait soit au niveau de l'environnement système avec `export PYTHONPATH=...`, soit au sein du script à l'aide de la variable `sys.path`). L'utilité de la boucle conditionnelle `if __name__ == "__main__":` apparaît alors : lors de l'importation d'un tel fichier comme librairie additionnelle, aucune des instructions figurant dans cette boucle n'est exécutée ; cela permet d'envisager d'utiliser dans un autre script les fonctions définies dans `monfichier.py` de façon “transparente”, sans qu'aucune instruction qui n'apparaîtrait pas dans le nouveau script ne soit exécutée. Les deux scripts ci-dessous illustrent cette possibilité, et l'on comparera les résultats de l'exécution du second script lorsque la boucle `for i in range(10000):` présente dans `monfichier.py` est ou n'est pas sujette à la condition `if __name__ == "__main__":...`

```

1 #!/usr/bin/env python3
2 # -*- coding: utf-8 -*-
3
4 """
5 Descriptif du fichier, qui s'appelle monfichier.py
6 """
7
8 # Importation des librairies
9
10 # Definition des fonctions
11 def my_favorite_func(n):
12     """ Definition d'une fonction, qui s'écrit, admettons, sur 10000 lignes,
13     et qui a nécessite un effort considérable de debugage. Pas question donc
14     de recommencer ce travail, ni même de faire un copier-coller... Imaginons
15     en outre que l'exécution de cette fonction prenne bcp de temps [ce n'est
16     pas le cas ici !] """
17     n += 1
18     return n
19
20
21 # Programme principal
22 if __name__ == "__main__":
23     # Appels de la fonction, par exemple pour faire des tests de vérification
24     # Ne sont exécutés que si __name__ == "__main__" !!!
25     for i in range(10000):
26         res = my_favorite_func(i)
27         print(res)

```

```

1 #!/usr/bin/env python3
2 # -*- coding: utf-8 -*-
3
4 """
5 Descriptif du fichier
6 """
7
8 # Importation des librairies
9 import monfichier # importation d'un fichier local
10
11 # Definition des fonctions
12

```

3. Le lecteur attentif reconnaîtra ici l'importation à proprement parler d'un module et non d'une librairie.

```

13 # Programme principal
14 if __name__ == "__main__":
15     # Accès aux fonctions définies dans la "librairie" monfichier
16     res = monfichier.my_favorite_func(41)
17     print("La réponse est {}".format(res))

```

*Remarque :* il est également possible, bien que très fortement déconseillé, d'importer un ou plusieurs contenus d'une librairie avec la commande `from lalibrairie import qqch` (ou `from lalibrairie import *` pour importer l'ensemble de la librairie). On accède alors directement à la fonction à l'aide du mot clé `qqch` (ou des mots clés présents dans la librairie) ; cependant, cette façon de procéder est à l'origine de bugs difficiles à détecter car rien ne “protège” ce(s) mot(s) d'une réaffectation ultérieure malencontreuse ; une telle réaffectation est beaucoup moins naturelle à écrire si le(s) mot(s) clé(s) reste(nt) “attaché(s)” à la librairie d'origine.

## III Exercices

### III - 1 Exercice 1

#### III - 1.1 Énoncé

Créer  $N$  fichiers (avec  $N$  passé comme argument optionnel de la commande, par défaut  $N = 5$ ) dont le nom sera construit en concaténant le nom de l'utilisateur (accessible avec la librairie `getpass`), le nom de la machine (accessible avec la librairie `socket`), le jour de création sous la forme YYMMJJ supposé identique pour tous les fichiers (accessible avec la librairie `time`) et le numéro  $N$  du fichier. Tous les champs doivent être séparés par un “underscore” `_`; le numéro  $N$  doit être écrit sur 5 caractères, avec le nombre nécessaire de 0 précédant la valeur de  $N$ . Tous les fichiers portent la même extension `txt` et doivent être placés dans le répertoire `./TEST`, à créer (éventuellement) avec la librairie `os`. Le résultat attendu est illustré ci-dessous.

```

[cwinisdo@leo38]:~$ python createfiles.py
[cwinisdo@leo38]:~$ ls -lrt TEST
total 0
-rw-rw-r-- 1 cwinisdo cwinisdo 0 sept. 12 14:32 cwinisdo_leo38_180912_00004.txt
-rw-rw-r-- 1 cwinisdo cwinisdo 0 sept. 12 14:32 cwinisdo_leo38_180912_00003.txt
-rw-rw-r-- 1 cwinisdo cwinisdo 0 sept. 12 14:32 cwinisdo_leo38_180912_00002.txt
-rw-rw-r-- 1 cwinisdo cwinisdo 0 sept. 12 14:32 cwinisdo_leo38_180912_00001.txt
-rw-rw-r-- 1 cwinisdo cwinisdo 0 sept. 12 14:32 cwinisdo_leo38_180912_00000.txt
[cwinisdo@leo38]:~$ python createfiles.py 2
[cwinisdo@leo38]:~$ ls -lrt TEST
total 0
-rw-rw-r-- 1 cwinisdo cwinisdo 0 sept. 12 14:32 cwinisdo_leo38_180912_00004.txt
-rw-rw-r-- 1 cwinisdo cwinisdo 0 sept. 12 14:32 cwinisdo_leo38_180912_00003.txt
-rw-rw-r-- 1 cwinisdo cwinisdo 0 sept. 12 14:32 cwinisdo_leo38_180912_00002.txt
-rw-rw-r-- 1 cwinisdo cwinisdo 0 sept. 12 14:33 cwinisdo_leo38_180912_00001.txt
-rw-rw-r-- 1 cwinisdo cwinisdo 0 sept. 12 14:33 cwinisdo_leo38_180912_00000.txt
[cwinisdo@leo38]:~$

```

#### III - 1.2 Corrigé

```

1 #!/usr/bin/env python3
2 # -*- coding: utf-8 -*-
3
4 """
5 Descriptif du fichier
6 """
7
8 # Importation des librairies
9

```

```

10 # Definition des fonctions
11
12 # Programme principal
13 if __name__ == "__main__":
14     pass

```

### III - 2 Exercice 2

#### III - 2.1 Énoncé

Créer une liste contenant le nom de 4 étudiants (Christophe, David, Ruben, Stéphane). À l'aide de la librairie `random`, choisir pour chacun d'eux un entier (compris entre 3 et 5) correspondant au nombre de notes qui leur ont été attribuées au cours du semestre, chacune d'elle étant tirée au hasard entre 0 et 20. Effectuer la moyenne pour chaque étudiant et imprimer le résultat à l'écran. Le résultat attendu est illustré ci-dessous, pour peu que le générateur de nombres aléatoires soit initialisé avec la `seed = 1726980199`, et que, pour chaque étudiant, les notes soient attribuées avant que le nombre de notes de l'étudiant suivant ne soit déterminé. Résultat qui met en évidence l'intérêt de suivre avec assiduité le module... ☺

```
[cwinisdo@leo38]:~$ python notes.py
Resultat de Christophe : moyenne = 13.80
Resultat de David       : moyenne = 12.80
Resultat de Ruben       : moyenne =  9.50 (*)
Resultat de Stephane   : moyenne = 12.25
[cwinisdo@leo38]:~$
```

#### III - 2.2 Corrigé

```

1 #!/usr/bin/env python3
2 # -*- coding: utf-8 -*-
3
4 """
5 Descriptif du fichier
6 """
7
8 # Importation des librairies
9
10 # Definition des fonctions
11
12 # Programme principal
13 if __name__ == "__main__":
14     pass

```

### III - 3 Exercice 3

#### III - 3.1 Énoncé

À l'aide des librairies `importlib` (dont la fonction `importlib.import_module` permet d'importer un module) et `inspect` (qui permet entre autres d'analyser le contenu d'une librairie), déterminer pour les 3 librairies suivantes, `numpy`, `matplotlib` et `scipy`, le nombre de caractères utilisés pour écrire les documentations (`docstring`) de l'ensemble des fonctions présentes dans chacune d'entre elles. Le résultat attendu est présenté ci-dessous :

```
[cwinisdo@leo38]:~$ python documentation.py
Longueur de la documentation de la librairie numpy (version 1.15.2): 470241 caracteres
Longueur de la documentation de la librairie matplotlib (version 1.5.1): 10087 caracteres
Longueur de la documentation de la librairie scipy (version 0.17.0): 477562 caracteres
[cwinisdo@leo38]:~$
```

### III - 3.2 Corrigé

```

1 #!/usr/bin/env python3
2 # -*- coding: utf-8 -*-
3
4 """
5 Descriptif du fichier
6 """
7
8 # Importation des librairies
9
10 # Definition des fonctions
11
12 # Programme principal
13 if __name__ == "__main__":
14     pass

```

### III - 4 Exercice 4

#### III - 4.1 Énoncé

À l'aide des librairies `sys` (qui permet notamment de récupérer les arguments passés à la commande `python`),  `getopt` (qui permet notamment de “parser” [= décrypter] une chaîne de caractères de type “options d'une commande”), et de `math` (qui donne accès à des manipulations mathématiques élémentaires), déterminer l'angle  $\theta_{\text{eq}}$  caractérisant la position d'équilibre stable et la pulsation  $\omega_0$  des petites oscillations autour de  $\theta \simeq \theta_{\text{eq}}$  dans le problème du “cerceau tournant” (*cf* exercice 1 du TD 2 de mécanique analytique). Le programme doit pouvoir être utilisé sans arguments (*i.e.* sous la forme `python monscript.py`, avec des valeurs par défaut de  $g$ ,  $r$  et  $\omega$ ), en passant les valeurs des paramètres dans la commande (soit sous la forme “short version” `python monscript.py -g 9.81 -r 1.0 -w 2.0`, soit sous la forme “long version” `python monscript.py --gravity 9.81 --radius 1.0 --omega 2.0`), ou enfin en passant comme argument le nom d'un fichier dans lequel sont définis ces paramètres (sous la forme `python monscript.py -f myfile.txt` ou la version longue avec le mot-clé `--inputfile`). Le résultat attendu est présenté ci-dessous :

```

[cwinisdo@leo38]:~$ python cercle_tournant.py
Les parametres du probleme sont : g = 9.81, r = 1.0, omega = 2.0 [MKSA]
La position d'équilibre stable est : theta_eq = 3.14159265359
La pulsation des petites oscillations autour de cet equilbre est : w0 = 2.41039415864 (/s)
[cwinisdo@leo38]:~$ python cercle_tournant.py --radius 1.0 -w 2.0
Les parametres du probleme sont : g = 9.81, r = 1.0, omega = 2.0 [MKSA]
La position d'équilibre stable est : theta_eq = 3.14159265359
La pulsation des petites oscillations autour de cet equilbre est : w0 = 2.41039415864 (/s)
[cwinisdo@leo38]:~$ python cercle_tournant.py -f cercle_tournant.txt
Les parametres du probleme sont : g = 9.81, r = 1.0, omega = 4.0 [MKSA]
La position d'équilibre stable est : theta_eq = 2.2308066409
La pulsation des petites oscillations autour de cet equilbre est : w0 = 3.15994363083 (/s)
[cwinisdo@leo38]:~$

```

#### III - 4.2 Corrigé

```

1 #!/usr/bin/env python3
2 # -*- coding: utf-8 -*-
3
4 """
5 Descriptif du fichier
6 """
7
8 # Importation des librairies

```

```

9 # Definition des fonctions
10
11
12 # Programme principal
13 if __name__ == "__main__":
14     pass

```

## IV Miscellanées

### IV - 1 La librairie timeit

Le temps d'exécution d'un script est, sans surprise, un aspect important à considérer lorsque l'on envisage d'adopter une approche numérique pour résoudre un problème. Compte-tenu du caractère "multi-tâches" des ordinateurs actuels, il n'est pas si facile d'obtenir ces informations : lorsque l'on exécute à deux instants différents le même programme, il se peut que certaines ressources soient utilisées pour d'autres opérations que celles relatives au script d'intérêt, par exemple pour des aspects relevant du système d'exploitation, à la communication avec des périphériques (écran, clé USB, ...), à l'utilisation d'autres programmes (Firefox, ...). Une des possibilités d'obtenir des informations relativement pertinentes consiste à s'appuyer sur la librairie `timeit`, "which provides a simple way to time small bits of Python code. [...] It avoids a number of common traps for measuring execution times.". Le script ci-dessous illustre son utilisation.

```

1 #!/usr/bin/env python3
2 # -*- coding: utf-8 -*-
3
4 """
5 Mesures de performances temporelles
6 """
7
8 # Importation des librairies
9 import timeit
10 import random
11
12 # Definition des fonctions
13 def generate_onelist():
14     """ Genere une liste de 50 entiers choisis aleatoirement entre 0 et 100 """
15     random_list = [random.randint(0,100) for i in range(50)]
16     return random_list
17
18 # Programme principal
19 if __name__ == "__main__":
20     # Illustration de la granulometrie du resultat
21     for i in range(10):
22         beg = timeit.default_timer()
23         random_list = generate_onelist()
24         end = timeit.default_timer()
25         print("Elapsed wall clock time (s) for iteration {} : {}".format(i, end-beg))
26     # Utilisation de la methode timeit de la librairie timeit
27     print("Utilisation de la methode timeit.timeit (s) : {}"\n
28          .format(timeit.timeit(generate_onelist)))

```

Les résultats obtenus sur un ordinateur de bureau sont les suivants :

```
[cwinisdo@leo38]:~$ time python timing.py
Elapsed wall clock time (s) for iteration 0 : 5.50746917725e-05
Elapsed wall clock time (s) for iteration 1 : 7.10487365723e-05
Elapsed wall clock time (s) for iteration 2 : 6.103515625e-05
Elapsed wall clock time (s) for iteration 3 : 5.81741333008e-05
Elapsed wall clock time (s) for iteration 4 : 6.00814819336e-05
Elapsed wall clock time (s) for iteration 5 : 5.3882598877e-05
```

```
Elapsed wall clock time (s) for iteration 6 : 6.00814819336e-05
Elapsed wall clock time (s) for iteration 7 : 5.41210174561e-05
Elapsed wall clock time (s) for iteration 8 : 5.60283660889e-05
Elapsed wall clock time (s) for iteration 9 : 5.50746917725e-05
Utilisation de la methode timeit.timeit (s) : 34.6899759769
```

```
real 0m34.709s
user 0m34.660s
sys 0m0.048s
[cwinisdo@leo38]:~$
```

On notera les fluctuations des durées à chaque itération, pour une moyenne proche de  $5.8 \cdot 10^{-5}$  s. La méthode `timeit.timeit`, qui effectue (par défaut)  $10^6$  fois l'appel de la fonction, donne un résultat moyen proche de  $3.5 \cdot 10^{-5}$  s. La différence peut s'expliquer par la gestion interne des ressources nécessaires à l'appel et l'exécution des instructions présentes dans la fonction. En outre, l'appel de la fonction `timeit.default_timer` est elle-même (marginalement) chronophage. Enfin, la comparaison avec les résultats retournés par la commande Linux `time` montrent que l'exécution du script complet s'est effectué en 34.709 s, dont la quasi-totalité correspond au temps nécessaire au CPU pour “effectuer le calcul” et seuls  $\sim 0.1\%$  de cette durée étant “perdus” pour d'autres aspects.



# Numpy

## I De la nécessité d'utiliser numpy

Dans le domaine du calcul scientifique, il est très commun d'être amené à manipuler les “tableaux” (vecteurs, matrices, ...) de taille conséquente. La librairie `numpy`, en créant un nouveau type de variable *mutable* appelé “`numpy array`”, permet de manipuler ces objets de façon extrêmement efficace, comme le prouve l'exécution<sup>1</sup> du script reproduit ci-dessous.

```
1 #!/usr/bin/env python3
2 # -*- coding: utf-8 -*-
3
4 """
5 Illustration des performances (timing) de numpy
6 """
7
8 # Importation des librairies
9 import timeit
10 import numpy as np
11
12 # Definition des fonctions
13 def init alist raw(N):
14     """ Definition d'une liste de taille N avec range """
15     lst = range(N)
16     return lst
17
18 def init nparr(N):
19     """ Definition d'un numpy array de taille N avec np.arange """
20     arr = np.arange(N)
21     return arr
22
23 def init alist wnp(N):
24     """ Definition d'une liste de taille N avec np.arange puis conversion """
25     lst = np.arange(N)
26     lst = lst.tolist()
27     return lst
28
29 def mult nparr(arr, scalar):
30     """ Multiplication par un scalaire d'un numpy array (for timing comparaison) """
31     arr = scalar * arr
32     return
33
34 def mult alist(lst, scalar):
35     """ Multiplication par un scalaire d'une liste (for timing comparaison) """
36     for i, val in enumerate(lst):
37         lst[i] = scalar * val
38     return
39
40 def wrapper(func, *args, **kwargs):
41     """ Decorator trick to time a function with arguments """
42     def wrapped():
43         return func(*args, **kwargs)
44     return wrapped
```

1. Mesurer le temps d'exécution d'un script (ou d'une partie de celui-ci) n'est pas aussi facile qu'on pourrait le penser (dépendance vis-à-vis du système d'exploitation, distinction entre “CPU time” et “wall-clock time”...). La solution proposée dans le script repose sur l'utilisation de la librairie `timeit`, qui effectue, pour estimer le temps moyen d'exécution,  $10^6$  fois le fragment de code passé en argument. Des solutions à base de `time.time` (ou `time.clock`) sont également envisageables.

```

45
46 # Programme principal
47 if __name__ == "__main__":
48     # Definition des valeurs par defaut
49     Nsize = 1000
50     scalar = 1.
51     # Definition des array/list
52     arr = np.arange(Nsize)
53     lst = arr.tolist()
54     # Timing d'importation
55     print("Timing de l'importation de numpy (microsec.): {0}\n"\
56           .format(timeit.timeit("import numpy as np")))
57     # Timings d'initialisation
58     print("Initialisation d'une liste/np array de taille {0} (microsec.):"\\
59           .format(Nsize))
60     wrapped = wrapper(init alist raw, Nsize)
61     print(" --> dans le cas d'une liste : {0}".format(timeit.timeit(wrapped)))
62     wrapped = wrapper(init nparr, Nsize)
63     print(" --> dans le cas d'un numpy array : {0}".format(timeit.timeit(wrapped)))
64     wrapped = wrapper(init alist wnp, Nsize)
65     print(" --> dans le cas d'un np array converti finalement en liste: {0}"\
66           .format(timeit.timeit(wrapped)))
67     # Timings de multiplication
68     print("\nMultiplication d'une liste/np array de taille {0} par le scalaire {1} "\
69           (microsec.):{0}.format(Nsize, scalar))
70     wrapped = wrapper(mult alist, lst, scalar)
71     print(" --> dans le cas d'une liste : {0}".format(timeit.timeit(wrapped)))
72     wrapped = wrapper(mult nparr, arr, scalar)
73     print(" --> dans le cas d'un numpy array : {0}".format(timeit.timeit(wrapped)))

```

Les résultats obtenus sur un ordinateur de bureau sont les suivants :

```

Multiplication d'une liste/np array de taille 1000 par le scalaire 1.0      (microsec.):
--> dans le cas d'une liste :      57.2727451324
--> dans le cas d'un numpy array : 1.71499800682

```

Ces différences s'expliquent par la façon dont les objets sont stockés en mémoire (et donc la manière dont on y accède), ainsi que par l'implémentation des opérations spécifiques aux numpy arrays. Cet exemple illustre l'intérêt (ou plutôt la nécessité) d'abandonner des objets de type "liste" (ou assimilés) au bénéfice des numpy arrays dès que l'on doit travailler sur de grands ensembles de données numériques. L'exemple ci-après illustre quant à lui "l'obligation" (d'un point de vue performance) d'utiliser les fonctionnalités offertes par la librairie numpy et d'éviter à tout prix les boucles explicites sur les indices parcourant les différentes dimensions<sup>2</sup> d'un numpy array.

```

1 #!/usr/bin/env python3
2 # -*- coding: utf-8 -*-
3
4 """
5 Illustration des performances (timing) de numpy, et en particulier du caractere
6 'C-ordered' des numpy arrays
7 """
8
9 # Importation des librairies
10 import timeit
11 import numpy as np
12
13 def amax_byhand(arr):
14     """ Compute 'manually' the maximum of the array arr of dimension 3 """

```

2. En plus d'illustrer l'optimisation en terme de vectorisation des opérations portant sur un numpy array, cet exemple démontre le caractère "orienté ligne" du stockage (par défaut) d'un tel tableau [les plus suspicieux des lecteurs pourront utiliser la méthode `.strides` attachée à tout numpy array pour lever leurs derniers doutes...]. C'est un aspect qui ne peut être ignoré, en particulier si l'on est amené à utiliser d'autres langages de programmation, tels que C/C++, Mathematica (orientés ligne, *row-major order*), ou Fortran, Matlab (orientés colonne, *column-major order*).

```

15     val = None
16     if arr.ndim == 3:
17         for k in range(arr.shape[2]):
18             for j in range(arr.shape[1]):
19                 for i in range(arr.shape[0]):
20                     if arr[i,j,k] > val: val = arr[i,j,k]
21     return
22
23 def amax_sequential(arr, order):
24     """ Compute the maximum of the array arr by looping sequentially over the
25     different dimensions. The argument order switches between ascending and
26     decreasing order of the indexes. As an example, for arr[i,j,k], if order > 0,
27     the maximum over the i-index at fixed j,k is first computed, then the maximum
28     over the j-index at fixed k, and finally the maximum over the k-index.
29     Alternatively, if order < 0, the maximum over the k-index at fixed i,j is first
30     computed, then the maximum over the j-index at fixed i, and finally the maximum
31     over the i-index. If order = 0, the native numpy implementation is used """
32     if order > 0:
33         while arr.ndim > 0:
34             arr = np.amax(arr, axis=0)
35     elif order < 0:
36         while arr.ndim > 0:
37             arr = np.amax(arr, axis=-1)
38     else:
39         arr = np.amax(arr)
40     return
41
42 def wrapper(func, *args, **kwargs):
43     """ Decorator trick to time a function with arguments """
44     def wrapped():
45         return func(*args, **kwargs)
46     return wrapped
47
48 # Programme principal
49 if __name__ == "__main__":
50     # Definition des valeurs par defaut
51     ndim    = 3
52     Nsize   = 1000
53     Ntimeit = 10
54     if True:
55         # Definition du numpy array
56         default  = np.random.normal(size=Nsize**ndim).reshape(ndim*(Nsize,))
57         Cordered = np.array(default, order='C')
58         Fordered = np.array(default, order='F')
59         print("Timing using a {} array".format(ndim*(Nsize,)))
60         for arr, msg in zip((default, Cordered, Fordered),
61                             ('default ordered', 'C-ordered', 'F-ordered')):
62             if arr.ndim == 3:
63                 wrapped = wrapper(amax_byhand, arr)
64                 print("Manual computation of the max of a {} array : {} (s)".\
65                       format(msg, timeit.timeit(wrapped, number=Ntimeit)/Ntimeit))
66                 print("Numpy amax computing with a {} array".format(msg))
67                 for order, msg in zip((1,-1,0),('increasing order','decreasing order',\
68                                       'full numpy implementation')):
69                     wrapped = wrapper(amax_sequential, arr, order)
70                     print(" timing using {:<25} : {} (s)".\
71                           format(msg, timeit.timeit(wrapped, number=Ntimeit)/Ntimeit))
72

```

Les résultats obtenus sur un ordinateur de bureau sont les suivants, et se passent de tout commentaire :

Timing using a (1000, 1000, 1000) array

Manual computation of the max of a default ordered array : 274.487861872 (s)

Numpy amax computing with a default ordered array

timing using increasing order : 1.14200429916 (s)

```

timing using decreasing order      : 0.526754808426 (s)
timing using full numpy implementation : 0.450998401642 (s)
Manual computation of the max of a C-ordered array : 265.048481941 (s)
Numpy amax computing with a C-ordered array
    timing using increasing order      : 1.32190718651 (s)
    timing using decreasing order      : 0.531886410713 (s)
    timing using full numpy implementation : 0.440027809143 (s)
Manual computation of the max of a F-ordered array : 146.721508026 (s)
Numpy amax computing with a F-ordered array
    timing using increasing order      : 0.510158514977 (s)
    timing using decreasing order      : 1.15292780399 (s)
    timing using full numpy implementation : 0.45540459156 (s)

```

## II Numpy arrays : commandes élémentaires

Dans toute la suite du document, la librairie `numpy` est importée au préalable à l'aide de la commande `import numpy as np`.

### II - 1 Caractérisation de numpy arrays

Si `x` désigne un numpy array, les méthodes suivantes permettent d'obtenir différentes caractéristiques de ce tableau.

<code>x.dtype</code>	type des éléments du tableau
<code>x.size</code>	nombre d'éléments du tableau
<code>x.shape</code>	dimension(s) du tableau [tuple, avec <code>x.ndim = len(x.shape)</code> ]
<code>x.ndim</code>	rang du tableau (vecteur, matrice, “tableau cubique”, ...)

### II - 2 Création de numpy arrays

La fonction `np.array` permet de convertir “beaucoup” de choses en numpy array.

	<code>x.dtype</code>	<code>x.size</code>	<code>x.shape</code>
<code>x = np.array([0,1,2,3])</code>	<code>int64</code>	4	(4,)
<code>x = np.array([0.,1,2])</code>	<code>float64</code>	3	(3,)
<code>x = np.array([[0,1,2],[1j,1+1j,2+1j]])</code>	<code>complex128</code>	6	(2, 3)
<code>x = np.array([[0],[1,2]])</code>	<code>object</code>	2	(2,)
<code>x = np.array([0,[1,2]])</code>			

Ces exemples mettent en évidence la conversion d'une liste en numpy array. Cette conversion s'accompagne (éventuellement) d'une “homogénéisation” du type de chacun des éléments. Si cette étape est impossible, la conversion échoue (*cf* dernier exemple).

Il existe également de nombreuses fonctions intrinsèques de créations de numpy arrays. La liste suivante est loin d'être exhaustive ; on se reportera à la documentation officielle pour explorer les possibilités offertes en particulier par les arguments optionnels.

<code>np.arange</code>	equivalent to the Python built-in range function for integer arguments ; when used with a float endpoint, the endpoint may or may not be included due to round-off error ( $\neq \text{np.linspace}$ )
<code>np.linspace</code>	evenly spaced numbers with careful handling of endpoints
<code>np.logspace</code>	return numbers spaced evenly on a log scale
<code>np.meshgrid</code>	return coordinate matrices from coordinate vectors ; very useful to evaluate functions on a grid
<code>np.mgrid</code>	grid-shaped arrays of evenly spaced numbers in N-dimensions
<code>np.zeros</code>	return a new array of given shape and type, filled with zeros
<code>np.ones</code>	return a new array of given shape and type, filled with ones

*Remarque :* `np.meshgrid` et `np.mgrid` permettent de faire essentiellement la même chose, mais les objets retournés par ces deux fonctions et l'ordre des indices diffèrent.

Il est possible de “concaténer” plusieurs numpy arrays en un seul en les “juxtaposant” pourvu que les dimensions des tableaux initiaux soient compatibles à l'aide des fonctions `np.hstack`, `np.vstack` et plus généralement

`np.concatenate`; ainsi, à partir des tableaux `x` (de dimension  $(dim1, dim2)$ ) et `y` (de dimension  $(dim1, dim3)$ ), `np.hstack((x,y))` retourne un tableau de dimension  $(dim1, dim2 + dim3)$  correspondant à la “juxtaposition horizontale” de `x` et `y`. Les opérations inverses sont également possibles à l'aide de `np.hsplit` ou `np.vstack`). Enfin, la sous-librairie `np.random` permet de créer des numpy arrays dont les éléments peuvent être tirés aléatoirement selon différentes lois (uniforme, normale, ..., cf documentation officielle).

## II - 3 Modification de format

Lorsque le tableau est créé, ses dimensions peuvent être modifiées *a posteriori* à l'aide de la fonction `np.reshape` (ou de la méthode équivalente du même nom). La fonction `np.ravel` (ou la méthode équivalente `flatten`) crée, à partir d'un tableau, un vecteur par “concaténation” de toutes les dimensions du tableau original; cette fonction/méthode permet d'optimiser certaines opérations portant sur des tableaux. La fonction `np.transpose` (ou la méthode équivalente `T`) permet de permuter les dimensions du tableau, et donc en particulier de déterminer la transposée d'une matrice (à noter que dans ce cas, la méthode `T` effectue l'opération “in-place”, i.e. la commande `x.T` écrase le contenu initial de la variable, contrairement à la fonction `np.transpose(x)`, qui retourne une variable supplémentaire sans affecter le contenu de `x`).

## II - 4 Accès aux éléments d'un numpy array

Soit `x` un numpy array de dimensions  $(dim1, dim2, dim3)$ , dont les éléments sont notés  $x_{ijk}$ , où  $i \in \{0, dim1 - 1\}$ ,  $j \in \{0, dim2 - 1\}$ ,  $k \in \{0, dim3 - 1\}$ . De façon “visuelle”, ce tableau possède  $dim1$  lignes<sup>3</sup>,  $dim2$  colonnes, et  $dim3$  “profondeurs”. Les commandes suivantes retournent ainsi :

		type	alternative
<code>x[0,0,0]</code>	$x_{000}$	de type correspondant à <code>x.dtype</code>	
<code>x[:,0,0]</code>	$x_{i00} \forall i \in \{0, dim1 - 1\}$	numpy array de shape $(dim1, )$	
<code>x[0,:,0]</code>	$x_{0j0} \forall j \in \{0, dim2 - 1\}$	numpy array de shape $(dim2, )$	
<code>x[0,0,:]</code>	$x_{00k} \forall k \in \{0, dim3 - 1\}$	numpy array de shape $(dim3, )$	
<code>x[0,:,:]</code>	$x_{0jk} \forall j, \forall k$	numpy array de shape $(dim2, dim3)$	<code>x[0]</code>
<code>x[:,0,:]</code>	$x_{i0k} \forall i, \forall k$	numpy array de shape $(dim1, dim3)$	<code>x[:,0]</code>

*Remarque* : il est facile de se convaincre que si `x[0]` désigne le même numpy array (de shape  $(dim2, dim3)$ ) que `x[0,:,:]`, alors `x[0][0]` correspond à `x[0,0,:]` et est un numpy array de shape  $(dim3, )$  pour lequel on peut accéder par exemple au premier élément par `x[0][0][0]`. Sauf justification précise (par exemple si le format du tableau est inconnu *a priori*), il est déconseillé d'utiliser ce type de notations.

Python offre également la possibilité de compter “à rebours” :  $-1$  désigne le dernier indice (i.e.  $dim1 - 1, dim2 - 1$  ou  $dim3 - 1$ ),  $-2$  le pénultième, etc. Ainsi `x[-1,-2,-3]` permet d'accéder au même élément que celui que retourne les commandes successives `dims = x.shape ; x[dims[0]-1,dims[1]-2,dims[2]-3]`.

Python offre enfin la possibilité d'effectuer du “slicing” sur chacun des indices sous la forme d'un “triplet” `ibeg:ienend:step`. S'ils ne sont pas spécifiés, les différents champs prennent les valeurs par défaut à savoir `0` pour `ibeg`, `-1` pour `ienend` et `1` pour `step`. Ainsi, en supposant que l'indice concerné est `i` :

		type
<code>x[1:,0,0]</code>	$x_{i00} \forall i \in \{1, dim1 - 1\}$	numpy array de shape $(dim1 - 1, )$
<code>x[1:5,0,0]</code>	$x_{i00} \forall i \in \{1, 2, 3, 4\}$	numpy array de shape $(4, )$
<code>x[1:5:2,0,0]</code>	$x_{i00} \forall i \in \{1, 3\}$	numpy array de shape $(2, )$
<code>x[1::2,0,0]</code>	$x_{i00} \forall i \in \{1, 3, 5, 7, 9, \dots\}$	...
<code>x[0:1,:,:]</code>	$x_{0jk} \forall j, \forall k$	numpy array de shape $(1, dim2, dim3)$ (!!)

On notera que l'indice `ibeg` est *inclus* alors que l'indice `ienend` est *exclus*. Cela permet d'accéder de façon très efficace à une “sous-partie” du tableau. Les esprits tortueux pourront vérifier qu'il est licite d'utiliser un pas `step` négatif.

*Remarque* : une manière visuelle de se souvenir de la façon dont fonctionne le slicing consiste à considérer les indices comme pointant *entre* les éléments du tableau. Ainsi, pour le numpy array `np.linspace(0,0.5,6)`, cela correspond au schéma suivant :

+---+---+---+---+---+

3. Dans le cas d'un tableau de rang 1, le vecteur ne doit pas être vu “verticalement” mais “horizontalement”, comme l'atteste le fait que c'est la fonction `np.hstack` (et non pas `np.vstack`) qui permet de “concaténer” deux tels vecteurs.

	0.	0.1	0.2	0.3	0.4	0.5	
0	1	2	3	4	5	6	
-6	-5	-4	-3	-2	-1		

### III Opérations sur les numpy arrays

L'intérêt fondamental de la librairie `numpy` est de fournir des fonctions mathématiques extrêmement efficaces pour traiter les numpy arrays, soit élément par élément, soit sur le tableau (ou les tableaux) en entier. Il est donc crucial (en particulier si la taille des tableaux devient conséquente) d'utiliser ces fonctions optimisées, associées éventuellement à des “slicings”, et de ne surtout pas expliciter des boucles sur les indices.

Un concept important pour les opérations portant sur les numpy arrays est l'aspect “conformant” (*broadcastable* en anglais). Pour illustrer le propos, considérons deux matrices  $M$  (de dimension  $m \times n$ ) et  $P$  (de dimension  $p \times q$ ). L'addition  $M + P$  est naturelle si  $m = p$  et  $n = q$ . Les tableaux sont alors dits conformants<sup>4</sup>. Mais on peut décider d'étendre la signification du signe “+” et donner un sens par exemple à l'opération d'addition entre une matrice et un scalaire : très naturellement, cela correspond à ajouter le scalaire à tous les éléments de la matrice. On peut prolonger la démarche et considérer l'addition entre une matrice et un vecteur : celle-ci sera entendue comme l'addition (terme à terme) du vecteur ligne à chaque ligne de la matrice<sup>5</sup>, et ne sera donc possible que si le format de la matrice ( $dim1, dim2$ ) est compatible avec celui du vecteur ( $dim2, 1$ ). Si ce n'est pas le cas, la matrice et le vecteur ne sont pas conformants et l'opération est impossible. Et de poursuivre ainsi l'extension de la signification du signe “+”... On peut imaginer faire de même avec d'autres opérations, par exemple pour la multiplication (le signe “\*” désignant autre chose que le “produit externe” lorsqu'il ne s'agit plus d'effectuer la multiplication par un scalaire  $\in \mathbb{K}$ ), le calcul de puissance... Ces définitions conventionnelles, qui obligent initialement à beaucoup de circonspection dans le sens à attribuer à certaines notations (par exemple que signifie  $M/N$  lorsque  $N$  n'est pas une matrice, ou, si elle l'est, pas inversible ?), peuvent être contournées en faisant usage des fonctions équivalentes<sup>6</sup> telles que `np.add`, `np.subtract`, `np.multiply`, `np.divide`.

Pour déterminer si deux (ou plus) numpy arrays sont conformants ou non (et par conséquent si certaines opérations sont licites ou pas, auquel cas Python retourne un message d'erreur `ValueError: operands could not be broadcast together`), Python effectue<sup>7</sup> les opérations suivantes :

- étape 1 : si les tableaux sont de rang (`x.ndim`) différent, Python étend le format (`x.shape`) du plus petit en le faisant précéder d'autant de 1 que nécessaire ;
- étape 2 : Python vérifie, dimension par dimension, en commençant par la dernière (de “droite à gauche” donc), qu'elles sont soit de même longueur, soit que l'une d'elle est égale à 1 auquel cas le tableau est dupliqué à la volée autant de fois que nécessaire pour atteindre la taille de l'autre tableau dans cette direction ;
- étape 3 : l'opération est effectuée élément par élément.

On notera en particulier que le numpy array de format ( $dim1, dim2$ ) est broadcastable avec les numpy arrays de format ( $dim1, 1$ ), ( $1, dim2$ ) ou encore ( $dim2, 1$ ) mais pas avec le numpy array de format ( $dim1, 1$ ). Pour pallier cette difficulté, il est parfois nécessaire “d'ajouter des dimensions”, à l'aide de la fonction `np.expand_dims`. Dans ce contexte, il sera également utile de distinguer ce que retourne par exemple la commande `x[0:1, :, :, :]` comparée à `x[0, :, :, :]`. Les exemples ci-dessous illustrent différentes situations.

- Exemple 1.

---

4. Cela correspond au caractère de groupe de  $(\mathcal{M}_{m \times n}(\mathbb{K}), +)$ , “+” étant la loi de composition interne.

5. Une autre définition, tout aussi naturelle consisterait à additionner le vecteur colonne à chaque colonne de la matrice, et imposerait que les formats de la matrice et du vecteur soient respectivement ( $dim1, dim2$ ) et ( $dim1, 1$ ) pour que l'opération soit convenable définie. Cela n'a pas été le choix des développeurs de Python.

6. À chacun de trouver son équilibre entre clarté et légèreté syntaxique... À noter la possibilité offerte par ces fonctions de n'effectuer les opérations que sur certains éléments seulement, à l'aide de “masques”.

7. L'implémentation réelle est bien entendue toute autre, sans copies inutiles de données mais avec des algorithmes reposant sur des manipulations d'indices implémentés, de manière optimale, en C.

shape=(3,5)	shape=(2,5)			(Mêmes rangs)																																																		
<table border="1"> <tr><td>0</td><td>1</td><td>2</td><td>3</td><td>4</td></tr> <tr><td>0</td><td>1</td><td>2</td><td>3</td><td>4</td></tr> <tr><td>0</td><td>1</td><td>2</td><td>3</td><td>4</td></tr> </table>	0	1	2	3	4	0	1	2	3	4	0	1	2	3	4	<table border="1"> <tr><td>0</td><td>10</td><td>20</td><td>30</td><td>40</td></tr> <tr><td>0</td><td>10</td><td>20</td><td>30</td><td>40</td></tr> </table>	0	10	20	30	40	0	10	20	30	40	+	<table border="1"> <tr><td>0</td><td>1</td><td>2</td><td>3</td><td>4</td></tr> <tr><td>0</td><td>1</td><td>2</td><td>3</td><td>4</td></tr> <tr><td>0</td><td>1</td><td>2</td><td>3</td><td>4</td></tr> </table>	0	1	2	3	4	0	1	2	3	4	0	1	2	3	4	<table border="1"> <tr><td>0</td><td>10</td><td>20</td><td>30</td><td>40</td></tr> <tr><td>0</td><td>10</td><td>20</td><td>30</td><td>40</td></tr> </table>	0	10	20	30	40	0	10	20	30	40
0	1	2	3	4																																																		
0	1	2	3	4																																																		
0	1	2	3	4																																																		
0	10	20	30	40																																																		
0	10	20	30	40																																																		
0	1	2	3	4																																																		
0	1	2	3	4																																																		
0	1	2	3	4																																																		
0	10	20	30	40																																																		
0	10	20	30	40																																																		
		✖	ValueError...	Rien à faire sur la dernière dim.																																																		

- Exemple 2.

shape=(3,5)	shape=(5,)	shape=(3,5)	shape=(1,5)	(Rangs init. $\neq$ )																									
<table border="1"> <tr><td>0</td><td>1</td><td>2</td><td>3</td><td>4</td></tr> <tr><td>0</td><td>1</td><td>2</td><td>3</td><td>4</td></tr> <tr><td>0</td><td>1</td><td>2</td><td>3</td><td>4</td></tr> </table>	0	1	2	3	4	0	1	2	3	4	0	1	2	3	4	<table border="1"> <tr><td>0</td><td>10</td><td>20</td><td>30</td><td>40</td></tr> </table>	0	10	20	30	40	+	<table border="1"> <tr><td>0</td><td>10</td><td>20</td><td>30</td><td>40</td></tr> </table>	0	10	20	30	40	Modif. shape
0	1	2	3	4																									
0	1	2	3	4																									
0	1	2	3	4																									
0	10	20	30	40																									
0	10	20	30	40																									
		✖	<table border="1"> <tr><td>0</td><td>1</td><td>2</td><td>3</td><td>4</td></tr> <tr><td>0</td><td>1</td><td>2</td><td>3</td><td>4</td></tr> <tr><td>0</td><td>1</td><td>2</td><td>3</td><td>4</td></tr> </table>	0	1	2	3	4	0	1	2	3	4	0	1	2	3	4	Rien à faire sur la dernière dim.										
0	1	2	3	4																									
0	1	2	3	4																									
0	1	2	3	4																									
		✖	<table border="1"> <tr><td>0</td><td>1</td><td>2</td><td>3</td><td>4</td></tr> <tr><td>0</td><td>1</td><td>2</td><td>3</td><td>4</td></tr> <tr><td>0</td><td>1</td><td>2</td><td>3</td><td>4</td></tr> </table>	0	1	2	3	4	0	1	2	3	4	0	1	2	3	4	Duplication de l'avant-dernière dim.										
0	1	2	3	4																									
0	1	2	3	4																									
0	1	2	3	4																									
		✖	<table border="1"> <tr><td>0</td><td>11</td><td>22</td><td>33</td><td>44</td></tr> <tr><td>0</td><td>11</td><td>22</td><td>33</td><td>44</td></tr> <tr><td>0</td><td>11</td><td>22</td><td>33</td><td>44</td></tr> </table>	0	11	22	33	44	0	11	22	33	44	0	11	22	33	44	Calcul élément par élément										
0	11	22	33	44																									
0	11	22	33	44																									
0	11	22	33	44																									

- Exemple 3.

shape=(3,5)	shape=(3,1)			(Mêmes rangs)																																																	
<table border="1"> <tr><td>0</td><td>1</td><td>2</td><td>3</td><td>4</td></tr> <tr><td>5</td><td>6</td><td>7</td><td>8</td><td>9</td></tr> <tr><td>10</td><td>11</td><td>12</td><td>13</td><td>14</td></tr> </table>	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	<table border="1"> <tr><td>0</td></tr> <tr><td>1</td></tr> <tr><td>2</td></tr> </table>	0	1	2	**	<table border="1"> <tr><td>0</td><td>1</td><td>2</td><td>3</td><td>4</td></tr> <tr><td>5</td><td>6</td><td>7</td><td>8</td><td>9</td></tr> <tr><td>10</td><td>11</td><td>12</td><td>13</td><td>14</td></tr> </table>	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	<table border="1"> <tr><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td></tr> <tr><td>1</td><td>1</td><td>1</td><td>1</td><td>1</td></tr> <tr><td>2</td><td>2</td><td>2</td><td>2</td><td>2</td></tr> </table>	0	0	0	0	0	1	1	1	1	1	2	2	2	2	2	** Duplication de la dernière dim.
0	1	2	3	4																																																	
5	6	7	8	9																																																	
10	11	12	13	14																																																	
0																																																					
1																																																					
2																																																					
0	1	2	3	4																																																	
5	6	7	8	9																																																	
10	11	12	13	14																																																	
0	0	0	0	0																																																	
1	1	1	1	1																																																	
2	2	2	2	2																																																	
		✖	<table border="1"> <tr><td>0</td><td>1</td><td>2</td><td>3</td><td>4</td></tr> <tr><td>5</td><td>6</td><td>7</td><td>8</td><td>9</td></tr> <tr><td>10</td><td>11</td><td>12</td><td>13</td><td>14</td></tr> </table>	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	<table border="1"> <tr><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td></tr> <tr><td>1</td><td>1</td><td>1</td><td>1</td><td>1</td></tr> <tr><td>2</td><td>2</td><td>2</td><td>2</td><td>2</td></tr> </table>	0	0	0	0	0	1	1	1	1	1	2	2	2	2	2	Rien à faire sur l'avant-dernière dim.																		
0	1	2	3	4																																																	
5	6	7	8	9																																																	
10	11	12	13	14																																																	
0	0	0	0	0																																																	
1	1	1	1	1																																																	
2	2	2	2	2																																																	
		✖	<table border="1"> <tr><td>1</td><td>1</td><td>1</td><td>1</td><td>1</td></tr> <tr><td>5</td><td>6</td><td>7</td><td>8</td><td>9</td></tr> <tr><td>100</td><td>121</td><td>144</td><td>169</td><td>196</td></tr> </table>	1	1	1	1	1	5	6	7	8	9	100	121	144	169	196	Calcul élément par élément																																		
1	1	1	1	1																																																	
5	6	7	8	9																																																	
100	121	144	169	196																																																	

À l'évidence, ces règles, illustrées sur des exemples avec des tableaux de rang 1 ou 2, sont valables pour les tableaux de rang supérieur. Le tableau ci-dessous illustre quelques possibilités (et impossibilités) avec des tableaux dont les formats correspondent aux lignes / colonnes :

(5,3,1)	(1,3,4)	(2,1,4)	(3,1)	(3,2)	(3,4)	(4,)
✓	✓	✗	✓	✓	✓	✓
✓	✓	✓	✓	✗	✓	✓
✗	✓	✓	✓	✗	✓	✓
✓	✓	✓	✓	✓	✓	✓
✓	✗	✗	✓	✓	✗	✗
✓	✓	✓	✓	✗	✓	✓
✓	✓	✓	✓	✗	✓	✓

### III - 1 Opérations élément par élément

Sans chercher à être exhaustif, on peut citer :

- les fonctions trigonométriques, hyperboliques, d'arrondi, d'exponentiation et logarithme..., habituellement définies sur  $\mathbb{K}$  et étendues ici aux numpy arrays ;
- les opérations d'addition/soustraction, de multiplication/division, d'élévation à la puissance, et de comparaison entre un numpy array et un scalaire ou un tableau conformant.

À noter la possibilité éventuelle d'effectuer ces opérations sur certains éléments seulement, à l'aide d'un “masque”.

### III - 2 Opérations sur l'ensemble du tableau

Sans chercher à être exhaustif, on peut citer la somme/le produit de tous les éléments, des aspects de statistiques sur ces éléments (max/min, moyenne arithmétique et écart-type, moyenne pondérée, médiane, ...).... À noter la possibilité d'effectuer ces opérations suivant certaines dimensions seulement.

### III - 3 Références

La documentation en ligne de Python permet de trouver les noms (et les détails) sur les fonctions présentées ci-dessus (et les autres) :

- <https://docs.scipy.org/doc/numpy/reference/routines.math.html>
- <https://docs.scipy.org/doc/numpy/reference/routines.statistics.html>
- <https://docs.scipy.org/doc/numpy/reference/routines.random.html>

## IV Exercices

### IV - 1 Exercice 1 : slicing

#### IV - 1.1 Énoncé

Créer un tableau  $[a_{ijk}]$  avec  $i \in \{0, i_{\max} - 1\}$ ,  $j \in \{0, j_{\max} - 1\}$ ,  $k \in \{0, k_{\max} - 1\}$ , les valeurs des indices  $i_{\max}$ ,  $j_{\max}$  et  $k_{\max}$  étant générés aléatoirement à partir de la loi uniforme discrète sur  $[10; 20]$ , les valeurs des éléments  $a_{ijk}$  étant générées aléatoirement à partir de la loi normale  $\mathcal{N}(\mu = 0, \sigma = 1)$ . Écrire une fonction qui retourne la “valeur du milieu du tableau”  $a_{i_{1/2}j_{1/2}k_{1/2}}$  (par exemple  $i_{1/2} = 2$  si  $i \in \{0, 4\}$  mais également si  $i \in \{0, 5\}$ , et de même pour  $j_{1/2}, k_{1/2}$ ), la valeur moyenne du tableau pour les  $i$  pairs (avec 0 considéré comme nombre pair), la valeur moyenne du tableau pour les  $j$  multiples de 3 (avec 0 n'étant pas considéré comme multiple de 3), la moyenne sur la première moitié des indices  $k$ , la moyenne sur les 3 puis les 13 dernières valeurs de  $k$  et enfin la valeur moyenne de tous les éléments du tableau supérieurs ou égaux à la valeur médiane du tableau. On pourra se “faire la main” dans un premier temps en considérant un tableau de format  $(2, 3, 4)$  remplis des premiers entiers successifs, avant de tester le programme sur le tableau généré à l'aide de la fonction `generate_testarray` du fichier `numpy_utils.py` disponible sur le portail des études.

#### IV - 1.2 Corrigé

```

1 #!/usr/bin/env python3
2 # -*- coding: utf-8 -*-
3
4 """
5 Descriptif du fichier
6 """
7
8 # Importation des librairies
9
10 # Definition des fonctions
11

```

```

12 # Programme principal
13 if __name__ == "__main__":
14     pass

```

## IV - 2 Exercice 2 : broadcasting

### IV - 2.1 Énoncé

Créer un tableau  $[a_{ijkl}]$  puis déterminer successivement les valeurs médianes sur tous les indices, sur tous les indices sauf 1, sur tous les indices sauf 2 et sur tous les indices sauf 3. À fin de comparaison, donner les valeurs moyennes (sur tous les indices) du tableau complet auquel on soustrait ces différentes valeurs médianes. On pourra se “faire la main” dans un premier temps en considérant un tableau de format  $(2, 3, 4)$  remplis des premiers entiers successifs (et en s’arrêtant à l’étape “tous les indices sauf 2”).

### IV - 2.2 Corrigé

```

1 #!/usr/bin/env python3
2 # -*- coding: utf-8 -*-
3
4 """
5 Descriptif du fichier
6 """
7
8 # Importation des librairies
9
10 # Definition des fonctions
11
12 # Programme principal
13 if __name__ == "__main__":
14     pass

```

## IV - 3 Exercice 3 : différences finies

### IV - 3.1 Énoncé

Créer un tableau  $[a_{ij}]$ ,  $i \in \{0, 50\}$ ,  $j \in \{0, 50\}$  dont les éléments correspondent à l’échantillonnage sur une grille uniforme de la fonction  $f(x, y) = e^{-x^2} \times \sin(y)$  définie sur  $[-5 : 5] \times [0 : 2\pi]$ . Estimer les dérivées partielles  $\partial_x f(x, y)$  et  $\partial_y f(x, y)$  à l’aide des différences finies (*i.e.* de formules de la forme  $g'(x) \simeq (g(x+\delta) - g(x))/\delta + \mathcal{O}(\delta)$  ou  $g'(x) \simeq (g(x) - g(x-\delta))/\delta + \mathcal{O}(\delta)$  ou encore  $g'(x) \simeq (g(x+\delta) - g(x-\delta))/2\delta$ ). Pour simplifier le traitement des bords du domaine, la fonction réellement considérée correspondra à la “périodisation” de  $f(x, y)$  à l’extérieur du domaine. Comparer les différents résultats avec ceux obtenus à l’aide de la fonction `np.gradient`.

### IV - 3.2 Corrigé

```

1 #!/usr/bin/env python3
2 # -*- coding: utf-8 -*-
3
4 """
5 Descriptif du fichier
6 """
7
8 # Importation des librairies
9
10 # Definition des fonctions
11
12 # Programme principal

```

```
13 if __name__ == "__main__":
14     pass
```

## IV - 4 Exercice 4 : stéganographie

La stéganographie<sup>8</sup> consiste à dissimuler un message dans un autre message, en particulier une image. Un chapitre étant consacré au traitement d'images, il suffit d'admettre à ce stade que l'on peut associer à une image un tableau d'entiers non-signés codés sur 8 bits `uint8` dont les dimensions sont  $(n_x, n_y, 3)$  :  $n_x$  et  $n_y$  correspondent aux dimensions spatiales de l'image, et à chaque pixel correspond une couleur (par synthèse additive) à l'aide de 3 entiers (codés sur 8 bits, *i.e.* de 3 octets) associés à chacun des canaux R, G, B. La technique de stéganographie mise en œuvre dans cet exercice repose sur le fait que la couleur d'un pixel est peu affectée par la modification des "bits de poids faible" (méthode dite LSB, *least significant bit*). Ainsi, la couleur "jaune or" dont le code RGB est (255, 215, 0), correspondant au triplet d'octets (11111111, 11010111, 00000000), est très difficilement distinguable du jaune (240, 208, 0), correspondant au triplet d'octets (11110000, 11010000, 00000000). Il est alors facile de se convaincre que le rendu d'une image sera satisfaisant dès lors que l'on conserve, pour chaque pixel, uniquement les 4 bits de poids fort pour chacun des canaux RGB. Les 4 bits de poids faible étant ainsi "sans intérêt ou presque", on peut les utiliser pour "insérer" dans l'image initiale les 4 bits de poids fort d'une seconde image et construire ainsi le message caché. Tout le jeu consistera à faire le tri, pour chaque octet de chaque canal de chaque pixel, entre les bits codant l'image "de premier plan" et ceux de l'image "d'arrière-plan".

### IV - 4.1 Énoncé

Créer un tableau  $[a_{ijk}]$  d'entier non-signés codés sur 8 bits, de dimension  $(n_x, n_y, 3)$ ;  $n_x$  et  $n_y$  sont choisis au hasard sur l'intervalle  $[400; 800[$ , et le tableau est rempli d'entiers (entre 0 et 255) pour former *in fine* une image constituée de  $4 \times 4$  patchs de couleur (chaque couleur étant aléatoire). Créer de la même manière un second tableau.

Définir à l'aide de ces deux tableaux deux images de dimensions identiques, qui correspondent donc aux images de premier et d'arrière plan.

À l'aide de la fonction `np.unpackbits`, écrire la fonction "d'entrelacement" des deux images, et visualiser le résultat à l'aide de la fonction `plot_stegano` du fichier `numpy_stegano_utils.py` disponible sur le portail des études. De la même façon, écrire la fonction de reconstruction de l'image "cachée". Le résultat attendu (avec une initialisation du générateur aléatoire avec la `seed = 1691988472`) est présenté sur la figure Fig. 1.

Le lecteur attentif notera les quelques défauts que l'on peut repérer dans l'image "entrelacée", en particulier dès que l'on a des aplats de couleurs. Il est donc opportun de choisir avec perspicacité l'image de premier plan si l'on veut être parfaitement satisfait par le résultat, comme celui illustré sur la figure Fig. 2

### IV - 4.2 Corrigé

```
1 #!/usr/bin/env python3
2 # -*- coding: utf-8 -*-
3
4 """
5 Descriptif du fichier
6 """
7
8 # Importation des librairies
9
10 # Definition des fonctions
11
12 # Programme principal
13 if __name__ == "__main__":
14     pass
```

---

8. <https://fr.wikipedia.org/wiki/Stéganographie>

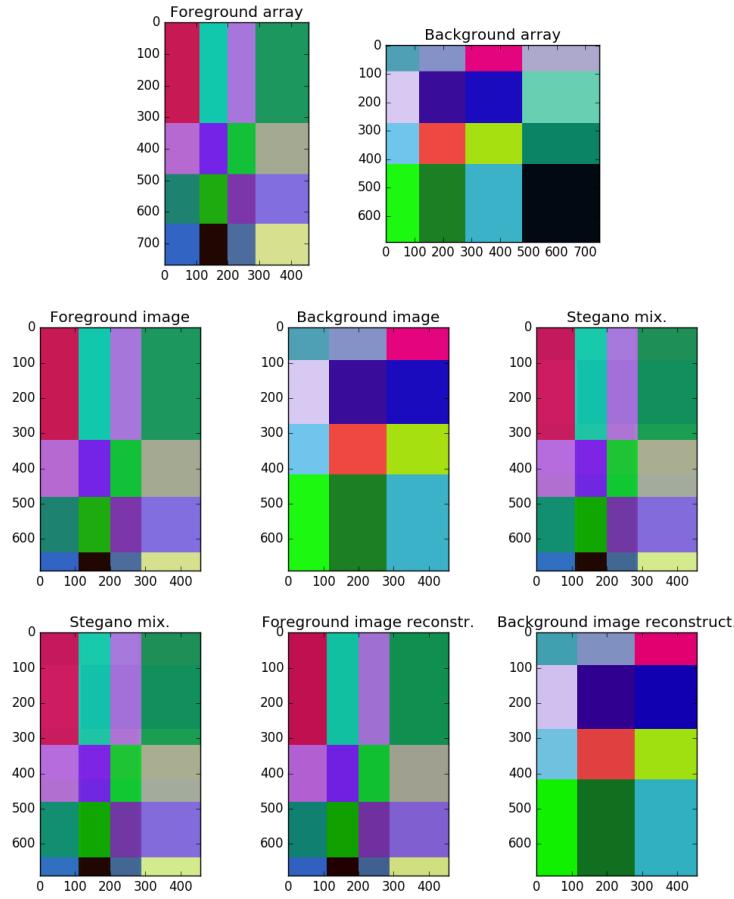


FIGURE 1 – Résultat attendu pour l'exercice portant sur la stéganographie.

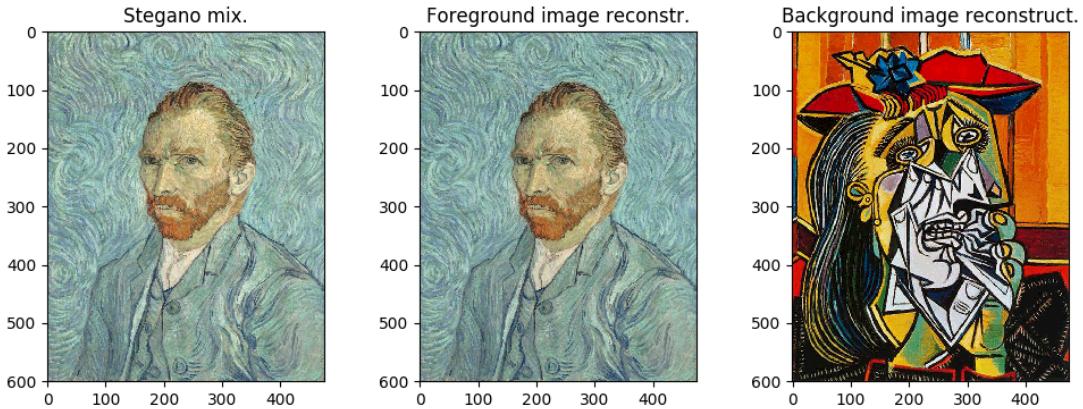


FIGURE 2 – Deux chefs-d'œuvre pour le prix d'un...

#### IV - 5 Exercice 5 : dynamique moléculaire

La dynamique moléculaire est une technique de simulation numérique permettant de modéliser l'évolution d'un système de particules au cours du temps. Elle consiste à résoudre (numériquement) les équations du mouvement des  $N$  différentes particules considérées (supposées contenues dans un volume  $V$ ) sous l'effet des différentes interactions interparticulaires, et obtenir ainsi un échantillonnage (pour un ensemble discret d'instants  $t_i$ ) de leurs trajectoires. Des techniques standards permettent de simuler l'effet d'un thermostat, et l'on parle alors de simulations dans l'ensemble "NVT". Si l'on suppose qu'après une phase de relaxation des conditions initiales, le système atteint un régime stationnaire et que celui-ci est représentatif de l'équilibre thermodynamique, la connaissance des trajectoires permet d'évaluer certaines quantités thermodynamiques (par exemple l'énergie interne du système, en évaluant les contributions potentielles et cinétiques pour chaque particule), ou des propriétés de transport (par exemple les coefficients de diffusion).

## IV - 5.1 Énoncé

On considère le résultat d'une simulation correspondant à 64 molécules d'eau dans une boîte cubique de  $12.42 \text{ \AA}$  de côté, en contact avec un thermostat à la température  $T$ . Cette simulation, réalisée par Ruben Staub, a tourné près de 49 heures sur une station de travail dédiée. Le fichier `WATER-pos-1.xyz`, obtenu à l'issue de cette simulation, correspond aux positions des  $64 \times 3 = 192$  atomes aux instant  $t_i = i \times \Delta t$ , avec  $i \in \langle 0, 9999 \rangle$  et  $\Delta t = 0.5 \text{ fs}$ . Le format des données correspond à un "standard" de la communauté des chimistes, pour lequel à chaque instant  $i$  sont écrits, sur une première ligne le nombre total d'atomes considérés, puis sur une seconde ligne quelques "commentaires", et enfin sur les lignes suivantes les coordonnées  $x, y, z$  (en  $\text{\AA}$ ) de chaque atome précédées par son symbole chimique, avant de passer à l'itération suivante (*cf* le résultat de la commande `more WATER-pos-1.xyz`). La fonction `extract_data` du fichier `numpy_MD_utils.py` disponible sur le portail des études permet de récupérer ces données sous la forme d'un numpy array `data[i,j,k,1]` qui correspond à la  $i^{\text{ieme}}$  coordonnée du  $k^{\text{ieme}}$  atome de la  $j^{\text{ieme}}$  molécule d'eau à l'instant  $t_i = i\Delta t$ . Les données sont disponibles sous la forme d'une archive compressée `MD_archive.7z` téléchargeable à l'adresse suivante : <https://filesender.ens-lyon.fr/?vid=2120ddfc-d316-f949-43f9-00007bfaecb7>.

Déterminer à tout instant l'énergie cinétique totale  $K_{\text{tot}}$  du système, puis sa valeur moyenne et son écart-type. Avec le théorème d'équipartition de l'énergie (supposé valide) qui donne la relation  $\langle E_{\text{cin}} \rangle = 3k_B T/2$  pour chaque atome, déterminer la température  $T$  du thermostat utilisé dans la simulation. Déterminer également la contribution cinétique de l'ensemble des centres de masse des différentes molécules  $K_{\text{CM}}$ , et la comparer à  $K_{\text{tot}}$ . Enfin, déterminer (à tout instant) la moyenne (sur l'ensemble des 64 molécules considérées) du carré du déplacement de leur centre de masse  $\langle (\Delta \mathbf{r})^2 \rangle \equiv \langle (\mathbf{r}_j^{\text{CM}}(t_i = i\Delta t) - \mathbf{r}_j^{\text{CM}}(t_0 = 0))^2 \rangle$ , et en déduire le coefficient de diffusion en admettant que dans le régime diffusif (dans le cas 3D), on a la relation suivante  $\langle (\Delta \mathbf{r})^2 \rangle = 3 \times 2D(t_i - t_0)$ . Pour la représentation graphique (*cf* Fig. 3), on pourra utiliser la fonction `plot_1D` du fichier `numpy_MD_utils.py`.

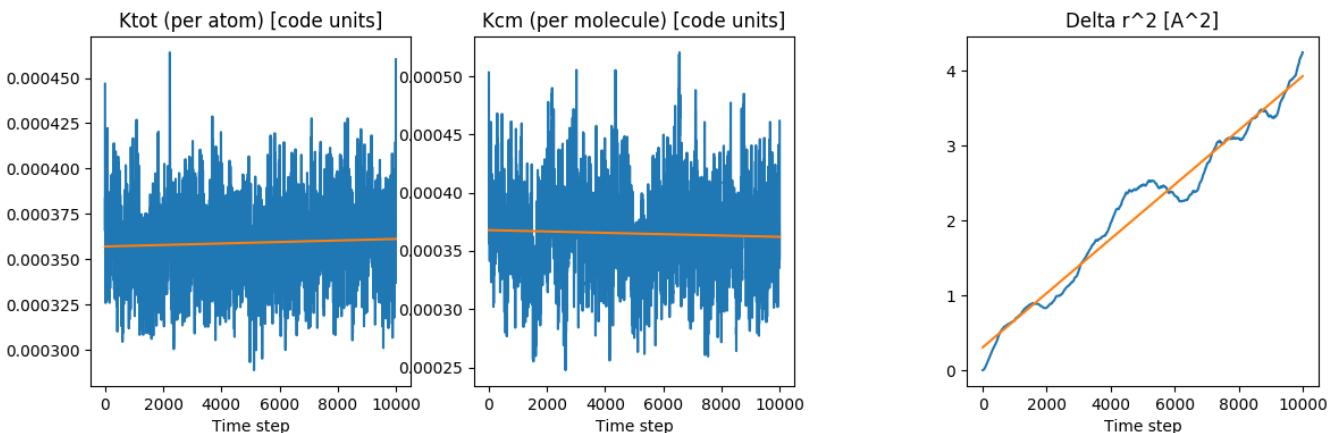


FIGURE 3 – Résultat attendu pour l'exercice portant sur la dynamique moléculaire. Pour l'ensemble des atomes (resp. des centres de masse), la valeur moyenne de l'énergie cinétique vaut 0.263 Ha (resp. 0.089 Ha), correspondant à une température de 288 K (resp. 293 K). Le coefficient de diffusion est estimé à  $0.12 \times 10^{-3} \text{ \AA}^2/\text{fs}$ .

## IV - 5.2 Corrigé

```

1 #!/usr/bin/env python3
2 # -*- coding: utf-8 -*-
3
4 """
5 Descriptif du fichier
6 """
7
8 # Importation des librairies
9
10 # Definition des fonctions
11

```

```

12 # Programme principal
13 if __name__ == "__main__":
14     pass

```

## V Miscellanées

### V - 1 Modules annexes

Numpy possède une (sous-)librairie d'algèbre linéaire `np.linalg` qui permet, entre autres, pour des tableaux de rang 2, d'effectuer un produit matriciel<sup>9</sup> “traditionnel”, d'obtenir leur décomposition de Cholesky et leur factorisation QR (utiles dans un contexte de résolution de système linéaire, *cf* cours de M1, “Analyse numérique”), de déterminer des valeurs propres (et singulières), ou encore de calculer norme, déterminant, conditionnement (propriété cruciale dans un contexte d'analyse numérique, *cf* le même cours de M1), ... La fonction `np.linalg.solve` permet en particulier de résoudre les systèmes linéaires de la forme  $Mx = y$ . La (sous-)librairie `np.fft` offre différentes fonctionnalités relatives à la transformée de Fourier discrète, et `np.polynomial` permet quant à elle de manipuler facilement les polynômes.

### V - 2 I/O

Il arrive très fréquemment que l'on soit amené à manipuler des fichiers, que ce soit pour importer des données numériques (par exemple des relevés de mesures expérimentales) ou, à l'inverse, pour exporter des résultats. Numpy permet de faire cela très facilement. Ainsi, pour les fichiers texte “.txt” (humainement lisibles), la fonction `np.loadtxt` permet d'importer les données, avec notamment la possibilité de spécifier les lignes et/ou colonnes à considérer et le caractère de “commentaire” (la fonction `np.genfromtxt` présente en outre l'avantage de pouvoir traiter les éventuelles données manquantes); la fonction `np.savetxt` permet l'opération inverse. De la même façon, les fonctions `np.load` et `np.save` permettent d'importer ou exporter des données sous le format binaire de `numpy`, ce qui peut être utile pour les “gros” fichiers.

---

9. À noter l'existence du type `matrix` (défini dans la librairie `numpy`) auquel sont attachées différentes méthodes couvrant essentiellement les mêmes fonctionnalités d'algèbre linéaire, mais dont l'utilisation est appelée à être obsolète.



# Matplotlib

## I Préambule

Le présent chapitre concerne la présentation de la librairie graphique **Matplotlib**, et plus précisément essentiellement de sa sous-librairie<sup>1</sup> **pyplot**. L'approche adoptée est d'ordre pratique, pour qu'à l'issue de ce chapitre, tout un chacun soit capable d'obtenir, en toute circonstance et avec un investissement minimal, un résultat parfaitement satisfaisant.

### I - 1 La règle d'or

En ce qui concerne Matplotlib, le bon réflexe, ne serait-ce que pour partir d'une "bonne base", consiste à chercher dans la galerie de Matplotlib <https://matplotlib.org/gallery.html> ce qui se rapproche le plus de l'objectif visé. En général, très rapidement, on aboutit à une solution qui génère de manière tout-à-fait convenable le résultat escompté. On s'apercevra très rapidement qu'il existe différentes syntaxes conduisant toutes aux mêmes résultats, les différences se faisant sur la lisibilité et la flexibilité des scripts.

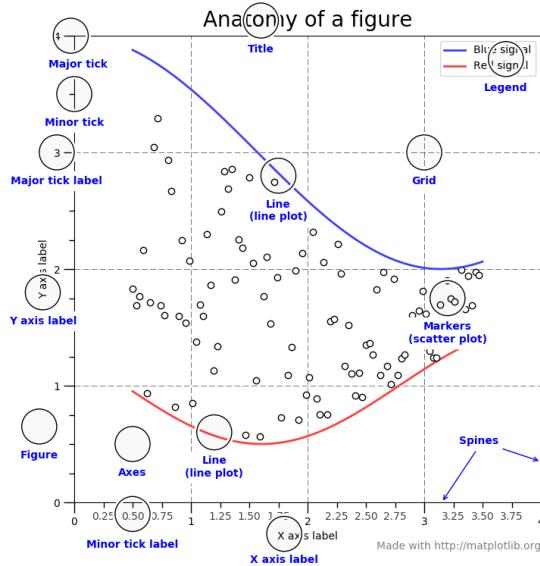


FIGURE 4 – Exemple de figure réalisée avec Matplotlib, illustrant les différents éléments constituant une figure (*cf* <https://matplotlib.org/examples/showcase/anatomy.html>), merci Ruben !

## II Exercices

Dans toute la suite du document, la sous-librairie **pyplot** est importée au préalable à l'aide de la commande `import matplotlib.pyplot as plt`, et la librairie **matplotlib** à l'aide de la commande `import matplotlib as mpl` si nécessaire.

1. La sous-librairie **pylab** correspond essentiellement à la fusion des fonctionnalités de **pyplot** et **numpy**, avec des syntaxes très proches de celles de MATLAB. C'en est à peu près l'unique "intérêt", surtout pour ceux qui "auraient du mal à décrocher" ...

## II - 1 Exercice 1 : pour s'échauffer

### II - 1.1 Énoncé

Récupérer les données correspondant au quartet d'Anscombe<sup>2</sup> sous la forme d'un fichier `anscombe.dat`. Faire une figure de taille 10x10 (en pouces...), avec un unique “plot” représentant le premier jeu de données, et la droite de meilleure régression sur l'intervalle des abscisses correspondant, droite dont on donnera l'équation en légende. Les axes doivent être nommés (“axe X” et “axe Y” respectivement) et un titre doit figurer sur la figure. Le résultat attendu est présenté sur la figure Fig. 5.

*Indices : plt.scatter, np.polyfit*

### II - 1.2 Corrigé

```

1 #!/usr/bin/env python3
2 # -*- coding: utf-8 -*-
3
4 """
5 Descriptif du fichier
6 """
7
8 # Importation des librairies
9
10 # Definition des fonctions
11
12 # Programme principal
13 if __name__ == "__main__":
14     pass

```

### II - 1.3 Énoncé

De la même façon, représenter les quatre jeux de données, avec les titres associés. Le résultat attendu est présenté sur la figure Fig. 5.

*Indices : plt.subplot*

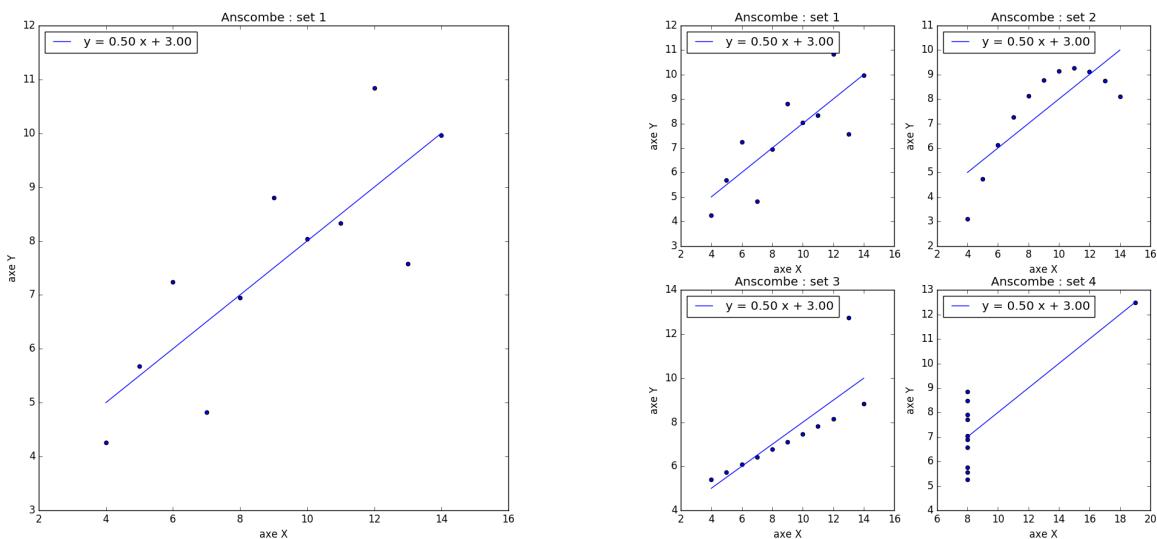


FIGURE 5 – Résultats attendus pour les exercices “pour s'échauffer”.

2. [https://en.wikipedia.org/wiki/Anscombe%27s\\_quartet](https://en.wikipedia.org/wiki/Anscombe%27s_quartet) pour ceux qui n'aurait pas d'idée...

## II - 1.4 Corrigés

```

1 #!/usr/bin/env python3
2 # -*- coding: utf-8 -*-
3
4 """
5 Descriptif du fichier
6 """
7
8 # Importation des librairies
9
10 # Definition des fonctions
11
12 # Programme principal
13 if __name__ == "__main__":
14     pass

```

## II - 1.5 Énoncé

Ajouter sur la figure précédente, pour chacun des graphes, la droite horizontale correspondant à la valeur moyenne des ordonnées. Cette droite doit couvrir la totalité des abscisses représentées sur chaque graphe, mais rien d'autre ne doit changer. Faire de même avec la valeur moyenne des abscisses.

*Indices : plt.xlim, plt.axhline*

## II - 1.6 Corrigé

```

1 #!/usr/bin/env python3
2 # -*- coding: utf-8 -*-
3
4 """
5 Descriptif du fichier
6 """
7
8 # Importation des librairies
9
10 # Definition des fonctions
11
12 # Programme principal
13 if __name__ == "__main__":
14     pass

```

## II - 2 Exercice 2

À ce stade, toute personne sachant glaner les informations pertinentes dans la documentation de `Matplotlib` et faire des copier-coller est capable d'obtenir sans trop de peine la figure espérée. Cependant, les scripts ainsi obtenus, sans plus de réflexion, sont en général très “rigides”, et nécessitent de nombreux correctifs dès que l'on veut modifier un (ou plusieurs) détails<sup>3</sup>. L'objectif des énoncés suivants est d'inviter le lecteur à adopter “les bons réflexes” pour spécifier, de manière efficace et souple, les propriétés des différents éléments que l'on souhaite personnaliser.

### II - 2.1 Énoncé

Représenter sur une figure les différents jeux de données du quadruplet d'Anscombe, regroupés par paires. Il y a donc 6 “plots” sur cette figure. Chaque jeu de données doit correspondre à un symbole (forme, couleur, taille)

3. C'est une situation plus commune qu'il ne peut sembler. Il suffit par exemple de penser au fait que la même figure ne peut pas être utilisée dans un rapport écrit et au cours d'une soutenance, sauf à renoncer dès le départ à ce que le public puisse lire les légendes sur les axes, distinguer la courbe en trait pointillé...

identique pour tout plot sur lequel il apparaît. La solution proposée doit permettre d'envisager très facilement la modification de ces symboles.

*Indices : itertools.combinations*

## II - 2.2 Corrigé

```

1 #!/usr/bin/env python3
2 # -*- coding: utf-8 -*-
3 """
4 Descriptif du fichier
5 """
6
7 # Importation des librairies
8
9 # Definition des fonctions
10
11 # Programme principal
12 if __name__ == "__main__":
13     pass

```

## II - 2.3 Énoncé

Ajouter à la figure précédente la droite de meilleure régression, ainsi que les droites correspondant aux valeurs moyennes des abscisses et des ordonnées. La solution proposée doit permettre d'envisager très facilement la modification de l'épaisseur des lignes et des tailles de police, pour disposer d'une version de type "rapport écrit", et d'une version "présentation orale". Les deux figures attendues sont présentées Fig. 6.

*Indices : mpl.rcParams*

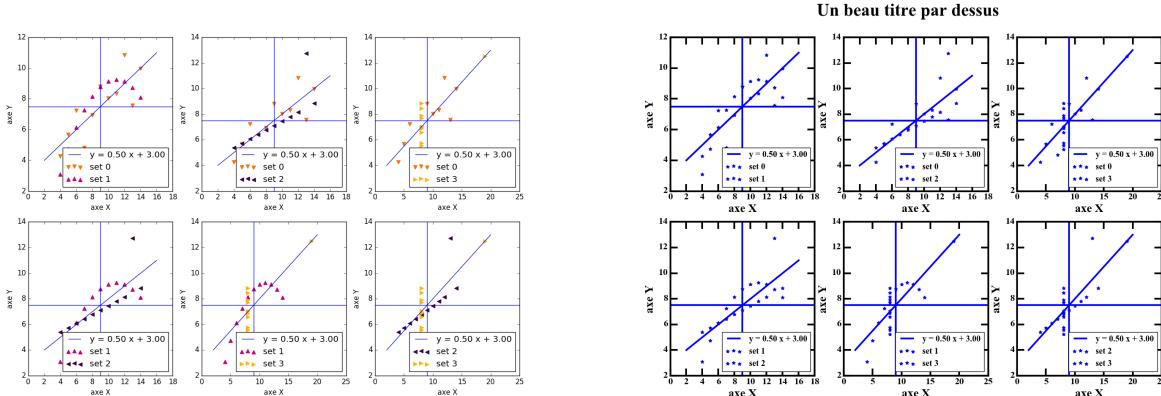


FIGURE 6 – Résultats attendus pour les versions rapport écrit / présentation orale.

## II - 2.4 Corrigé

```

1 #!/usr/bin/env python3
2 # -*- coding: utf-8 -*-
3 """
4 Descriptif du fichier
5 """
6
7 # Importation des librairies
8
9 # Definition des fonctions
10
11

```

```

12 # Programme principal
13 if __name__ == "__main__":
14     pass

```

De cette présentation, on pourra retenir les éléments suivants :

- pour faire rapidement une figure, rien de plus efficace que de regarder dans la galerie de Matplotlib “ce qui se fait” et d’adapter le script à ses besoins ;
- si l’on envisage de réutiliser le script (notamment pour créer des figures qui seront présentées dans différents contextes), il est opportun de prendre le temps d’identifier les paramètres que l’on sera amené à personnaliser, et d’adapter en conséquence le script pour que les syntaxes mises en œuvre permettent cette flexibilité.

## II - 3 Exercice 3

### II - 3.1 Énoncé

Reprendre l’exercice du chapitre “Numpy” portant sur l’estimation du gradient d’une fonction par différences finies, et visualiser les écarts entre les résultats obtenus à l’aide des différents schémas. Le résultat attendu est présenté sur la figure Fig. 7.

*Indices : plt.imshow, plt.contour*

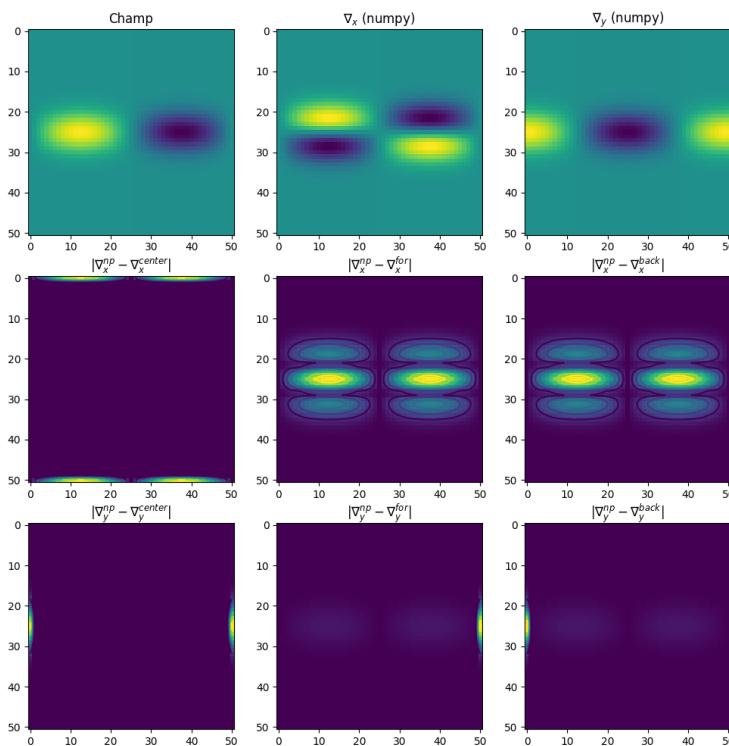


FIGURE 7 – Résultats attendus pour l’exercice portant sur l’opérateur gradient. Attention, avec `plt.imshow`, l’axe “x” (resp. “y”) correspond à l’axe vertical (resp. horizontal). La comparaison entre les résultats `np.gradient` vs schéma centré met en évidence la “périodisation” de la fonction  $f$  considérée dans le second cas. Pour ce qui est de la comparaison entre les schémas forward/backward vs `np.gradient`, ce sont les erreurs  $O(h)$  vs  $O(h^2)$  qui sont mis en évidence.

### II - 3.2 Corrigé

```

1 #!/usr/bin/env python3

```

```
2 # -*- coding: utf-8 -*-
3 """
4 Descriptif du fichier
5 """
6
7 # Importation des librairies
8
9 # Definition des fonctions
10
11 # Programme principal
12 if __name__ == "__main__":
13     pass
```

# Scipy

## I Introduction

L'objet de ce chapitre est de mettre en pratique l'utilisation de Python (et de ses librairies) dans différentes problématiques qui apparaissent de façon récurrente en physique. La librairie `scipy` est extrêmement précieuse dans ce contexte. Elle peut être essentiellement considérée comme une "extension" des fonctionnalités de `numpy` : les concepts-clés sont les mêmes (avec notamment les `numpy arrays`), la liste des modules disponibles plus riche. Associée à `matplotlib`, elle permet de donner des solutions élégantes et efficaces aux différents exercices proposés ci-dessous. Les exercices suivants permettent de découvrir quelques-unes de ces fonctionnalités, et, j'espère, de se convaincre de l'intérêt d'utiliser ces outils numériques.

## II Ajustement de données

### II - 1 Exercice 1

#### II - 1.1 Énoncé

Récupérer le fichier `electricite.csv` sur le portail des études, qui correspond à la consommation mensuelle d'électricité en France, de Septembre 1991 à Décembre 2015. Déterminer le meilleur ajustement (au sens des moindres carrés) quadratique de cet ensemble de données à l'aide de la fonction `numpy.polyfit`. À l'aide d'une analyse de Fourier (menée à l'aide de la librairie `numpy.fft`), identifier dans les données la présence d'une (ou plusieurs) composante(s) périodique(s). En déduire finalement (à l'aide<sup>1</sup> de `scipy.optimize.leastsq` ou de `scipy.optimize.least_squares`) un ajustement des données combinant une composante quadratique et une composante harmonique. Le résultat attendu est présenté sur la figure Fig. 8.

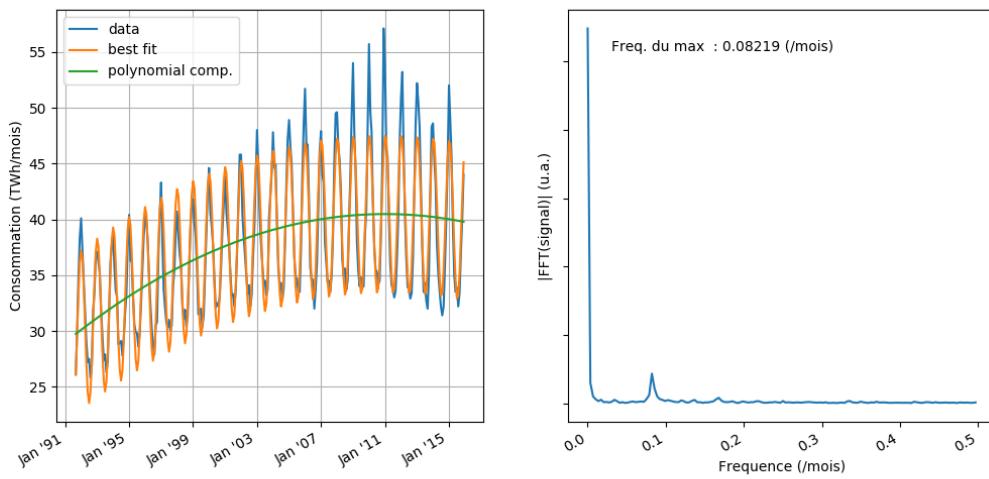


FIGURE 8 – Résultat attendu pour l'exercice portant sur la consommation électrique.

1. Un exemple d'un tel ajustement peut être facilement trouvé sur le web, par exemple <https://scipy-lectures.org/intro/summary-exercises/optimize-fit.html>.

## II - 1.2 Corrigé

```

1 #!/usr/bin/env python3
2 # -*- coding: utf-8 -*-
3
4 """
5 Descriptif du fichier
6 """
7
8 # Importation des librairies
9
10 # Definition des fonctions
11
12 # Programme principal
13 if __name__ == "__main__":
14     pass

```

## II - 2 Exercice 2

### II - 2.1 Énoncé

Récupérer le fichier `4br_grossc.duration` sur le portail des études, qui correspond aux durées relevées pour 1234 événements détectés par BATSE ([cf `https://gamma-ray.nsstc.nasa.gov/batse/grb/catalog/4b/`](https://gamma-ray.nsstc.nasa.gov/batse/grb/catalog/4b/) pour plus de précisions). À l'aide de la fonction `pyplot.hist`, tracer l'histogramme des durées à 90% de la fluence totale ( $T_{90}$ ) de chaque événement ; en déterminer (à l'aide de `scipy.optimize.curve_fit`) un meilleur ajustement sous la forme d'une distribution bimodale de lois log-normales (par rapport à  $T_{90}$ ) ou d'une distribution bimodale de lois normales (par rapport à  $\log_{10}(T_{90})$ ). Le résultat attendu est présenté sur la figure Fig. 9.

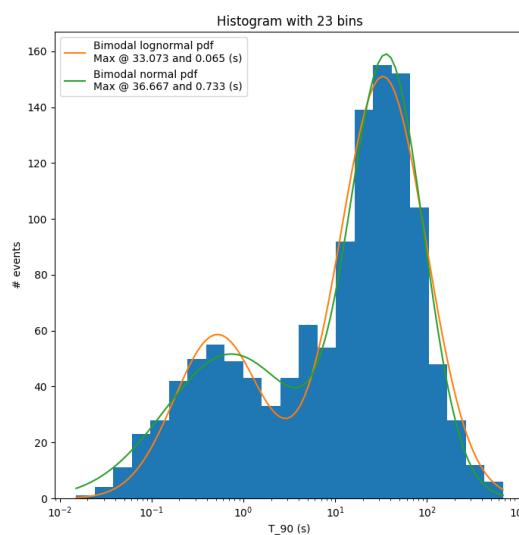


FIGURE 9 – Résultat attendu pour l'exercice portant sur les durées  $T_{90}$  du catalogue BATSE.

### II - 2.2 Corrigé

```

1 #!/usr/bin/env python3
2 # -*- coding: utf-8 -*-
3
4 """
5 Descriptif du fichier
6 """

```

```

7
8 # Importation des librairies
9
10 # Definition des fonctions
11
12 # Programme principal
13 if __name__ == "__main__":
14     pass

```

### III Intégrales

Numériquement, une intégrale peut être calculée de (très) nombreuses manières différentes. On peut ainsi par exemple considérer les “formules ouvertes de Newton-Cotes”, pour lesquelles, pour une intégrale  $\int_a^b f(x)dx$ , on introduit les abscisses  $x_i = a + i(b - a)/n$  et les valeurs  $f_i \equiv f(x_i)$  (avec  $i \in \langle 0; n \rangle$ ), et qui correspondent aux estimations suivantes :

Nom	$n$	
formule du point milieu	2	$(b - a)f_1$
formule du trapèze	3	$(b - a)/2 \times (f_1 + f_2)$
formule de Milne	4	$(b - a)/3 \times (2f_1 - f_2 + 2f_3)$

On notera que ces formules permettent de traiter les situations où l'intégrande n'est définie que sur l'intervalle ouvert  $]a : b[$ .

#### III - 1 Exercice 1

##### III - 1.1 Énoncé

On considère l'intégrale suivante :

$$I = \int_0^1 dt e^{-t^2}. \quad (1)$$

Implémenter les trois méthodes précédentes pour estimer cette intégrale ; l'intervalle  $[0; 1]$  sera successivement subdivisé en  $[0; 1/2] \cup [1/2; 1]$ ,  $[0; 1/4] \cup [1/4; 1/2] \cup [1/2; 3/4] \cup [3/4; 1]$ , ... pour estimer “au mieux” l'intégrale. Comparer les résultats à ceux obtenus à l'aide de la fonction `scipy.integrate.quad`, en particulier en terme d'erreur par rapport au pas de subdivision entre l'estimation numérique et la valeur théorique (qui est  $\sqrt{\pi}/2 \times \text{erf}(x)$ , accessible dans la librairie `scipy.special`). Le résultat attendu est présenté sur la figure Fig. 10, qui illustre en particulier le fait que les méthodes du point-milieu ou du trapèze sont d'ordre 2 (*i.e.*  $|\text{erreur}| \propto \Delta x^n$  avec  $n = 2$ ) alors que la méthode de Milne est d'ordre 4.

Reprendre l'exercice précédent pour l'intégrale apparaissant dans le calcul de la période d'un pendule pesant (longueur  $\ell$ , champ de gravité  $g$ , amplitude  $\theta_0 = \pi/2$ ) :

$$T = T_0 \times \frac{\sqrt{2}}{\pi} \int_0^{\theta_0} \frac{d\theta}{\sqrt{\cos \theta - \cos \theta_0}}, \quad (2)$$

où  $T_0 = 2\pi\sqrt{\ell/g}$  correspond à la période aux petites amplitudes. La valeur théorique s'obtient avec l'intégrale elliptique de première espèce (accessible dans la librairie `scipy.special`) :  $T/T_0 = 2/\pi \times K(\sin^2(\theta_0/2))$ . Le résultat attendu est présenté sur la figure Fig. 10. On pourra remarquer au passage que “l'ordre de convergence” du schéma de Milne est affecté par le comportement “pathologique” de l'intégrande.

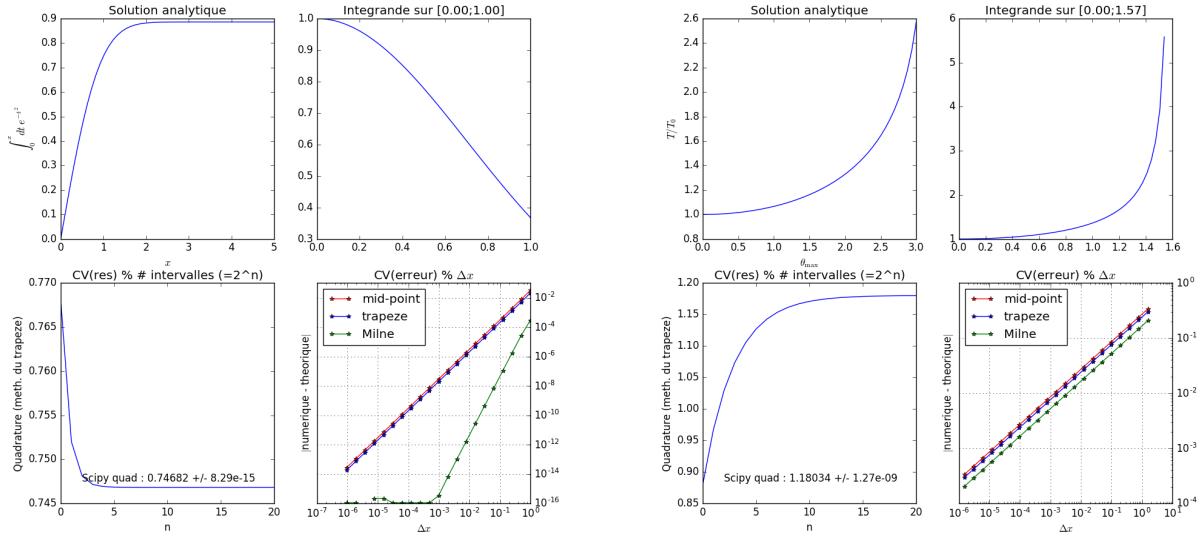


FIGURE 10 – Résultats attendus pour les exercices de calcul d'intégrales.

### III - 1.2 Corrigé

```

1 #!/usr/bin/env python3
2 # -*- coding: utf-8 -*-
3
4 """
5 Descriptif du fichier
6 """
7
8 # Importation des librairies
9
10 # Definition des fonctions
11
12 # Programme principal
13 if __name__ == "__main__":
14     pass

```

## IV Solution d'une équation non-linéaire

### IV - 1 Exercice 1

#### IV - 1.1 Énoncé

On cherche à déterminer la caractéristique d'une diode, *i.e.* la courbe  $I = f(U)$  correspondant au courant  $I$  qui la traverse lorsqu'on lui applique une tension  $U$ . Les équations à considérer sont les équations de Kirchhoff et de Shockley :

$$U_j = U - R_{\text{dyn}}I \quad \text{et} \quad I = I_{\text{sat}} \left( e^{U_j/U_0} - 1 \right), \quad (3)$$

où  $U_j$  désigne la tension aux bornes de la jonction,  $R_{\text{dyn}}$  la résistance dynamique de la diode,  $I_{\text{sat}}$  le courant de saturation et  $U_0$  la tension thermique. Le problème se ramène donc à déterminer, pour tout  $U$ , la solution de l'équation suivante :

$$\frac{R_{\text{dyn}}(I + I_{\text{sat}})}{U_0} e^{R_{\text{dyn}}(I + I_{\text{sat}})/U_0} = \frac{R_{\text{dyn}} I_{\text{sat}}}{U_0} e^{(U + R_{\text{dyn}} I_{\text{sat}})/U_0}, \quad (4)$$

qui est une équation transcendante de la forme  $w e^w = x$ . À l'aide de la fonction `scipy.optimize.brentq`, résoudre cette équation pour tracer la caractéristique de la diode sur l'intervalle  $U \in ]0;3]$  V, et comparer le

résultat obtenu avec la solution analytique donnée à l'aide de la (branche principale de la) fonction de Lambert  $W_0$ , disponible dans la librairie `scipy.special` :

$$I = \frac{U_0}{R_{\text{dyn}}} W_0 \left( \frac{R_{\text{dyn}} I_{\text{sat}}}{U_0} e^{(U + R_{\text{dyn}} I_{\text{sat}})/U_0} \right) - I_{\text{sat}}. \quad (5)$$

Les données numériques considérées correspondent à une diode de signal au silicium à 300 K, à savoir  $I_{\text{sat}} = 10 \text{ nA}$ ,  $R_{\text{dyn}} = 25 \text{ m}\Omega$ , et  $U_0 = 26 \text{ mV}$ . Le résultat attendu est présenté sur la figure Fig. 11.

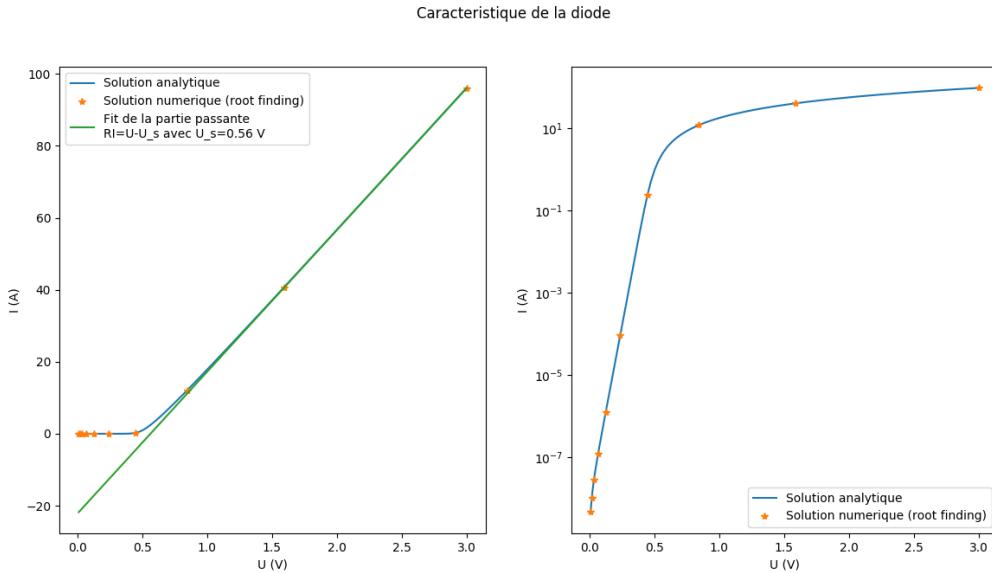


FIGURE 11 – Résultat attendu pour la caractéristique de la diode.

#### IV - 1.2 Corrigé

```

1 #!/usr/bin/env python3
2 # -*- coding: utf-8 -*-
3 """
4 Descriptif du fichier
5 """
6
7 # Importation des librairies
8
9 # Définition des fonctions
10
11 # Programme principal
12 if __name__ == "__main__":
13     pass

```

## V Résolution d'ODE

### V - 1 Exercice 1

#### V - 1.1 Énoncé

On considère le mouvement unidimensionnel correspondant à une chute libre avec frottements. La force de frottement est modélisée sous la forme  $f = 1/2\rho SC_x v^2$ , où  $\rho$  désigne la masse volumique du fluide,  $S$  la “surface projetée” du projectile,  $v$  sa vitesse (relativement au fluide), et  $C_x$  le coefficient de trainée. Pour une sphère de rayon  $R$ ,  $S = \pi R^2$ , et  $C_x$  est une fonction du nombre de Reynolds  $Re \equiv vR/\nu$  (où  $\nu$  correspond à la viscosité

cinématique du fluide). Il s'agit donc d'un problème aux conditions initiales portant sur la fonction  $v : \mathbb{R}^+ \rightarrow \mathbb{R}$ , correspondant à l'équation différentielle (du premier ordre) suivante :

$$m \frac{dv}{dt} \pm \frac{1}{2} \rho S C_x v^2 = -mg, \quad (6)$$

avec les conditions initiales  $v(t = 0) = 0$  (ou autre chose..). Les données numériques considérées correspondent à une balle de golf (diamètre : 43 mm, masse : 45 g) placée dans l'air à 25 °C ( $\rho = 1.2 \text{ kg/m}^3$ ,  $\nu = 1.6 \cdot 10^{-5} \text{ m}^2/\text{s}$ ) sur Terre. Le coefficient de trainée tel que proposé (pour une sphère) dans "Bubbles, Drops, and Particles", Clift, Grace & Weber [Academic Press, 1978] est implémenté dans le fichier `scipy_utils.py`.

À l'aide<sup>2</sup> de la fonction `scipy.integrate.odeint`, déterminer la solution du problème. Le résultat attendu est présenté sur la figure Fig. 12. On pourra remarquer au passage le "décrochement" du  $C_x$  au voisinage de  $Re \simeq 4 \cdot 10^5$  appelé "crise de trainée".

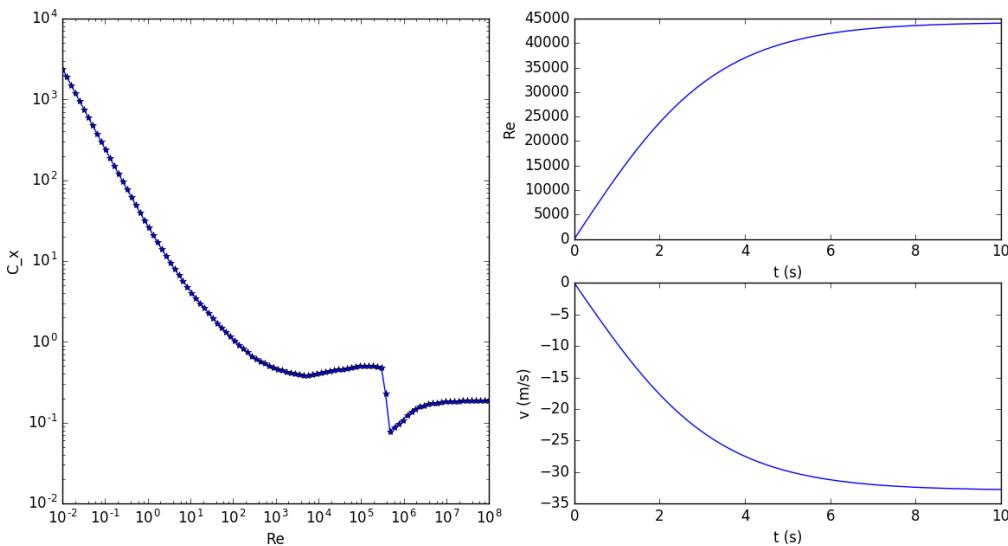


FIGURE 12 – Résultats attendus pour la chute libre 1D avec frottements.

## V - 1.2 Corrigé

```

1 #!/usr/bin/env python3
2 # -*- coding: utf-8 -*-
3
4 """
5 Descriptif du fichier
6 """
7
8 # Importation des librairies
9
10 # Définition des fonctions
11
12 # Programme principal
13 if __name__ == "__main__":
14     pass

```

2. Un exemple d'une telle utilisation peut être facilement être trouvée sur le web, par exemple <https://scipy-lectures.org/intro/scipy.html#integrating-differential-equations>.

## V - 2 Exercice 2

### V - 2.1 Énoncé

Il est naturel de préférer à l'exercice précédent celui consistant à déterminer la trajectoire complète d'une balle de golf. En supposant la trajectoire plane, il s'agit donc d'un problème aux conditions initiales portant sur deux fonctions ( $x(t)$  et  $y(t)$ ), chacune d'elles étant solution d'une équation différentielle du second ordre. Chaque équation du second ordre est réécrite comme un système de deux équations du premier ordre (en introduisant par exemple les variables intermédiaires  $v_x \equiv \dot{x}$  et  $v_y \equiv \dot{y}$ ) et l'on est donc amené à considérer le système de 4 équations du premier ordre portant sur les 4 fonctions  $x(t), v_x(t), y(t), v_y(t)$  suivant :

$$\dot{x} = v_x, \quad \dot{v}_x = -f_x, \quad \dot{y} = v_y, \quad \dot{v}_y = -g - f_y. \quad (7)$$

À l'aide de la fonction `scipy.integrate.odeint`, déterminer la solution du problème en fonction de l'angle initial imposé à la trajectoire de la balle (on suppose que la norme de la vitesse initiale vaut 80 m/s, indépendamment de l'angle imposé par le golfeur). Le résultat attendu est présenté sur la figure Fig. 13.

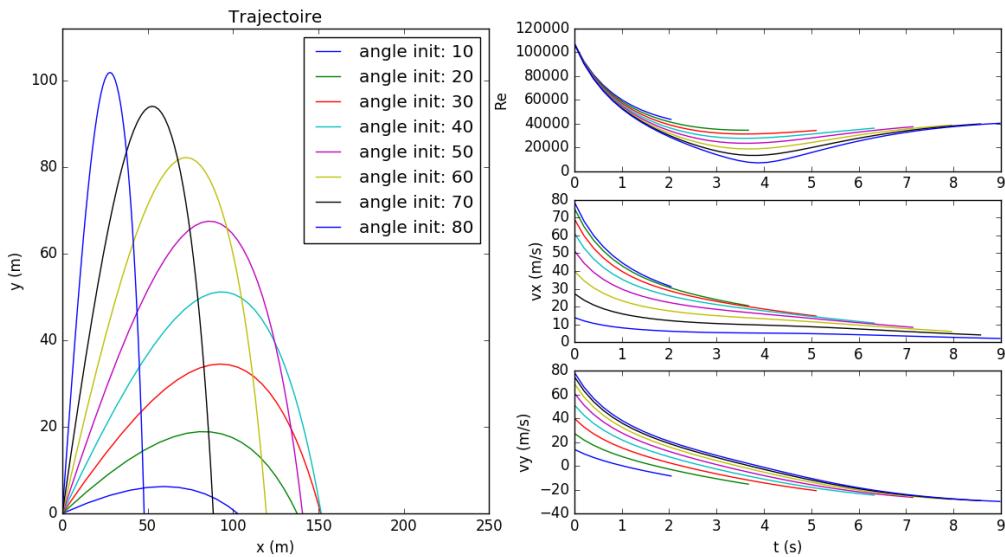


FIGURE 13 – Résultats attendus pour la trajectoire d'une balle de golf.

### V - 2.2 Corrigé

```

1 #!/usr/bin/env python3
2 # -*- coding: utf-8 -*-
3 """
4 Descriptif du fichier
5 """
6
7
8 # Importation des librairies
9
10 # Definition des fonctions
11
12 # Programme principal
13 if __name__ == "__main__":
14     pass

```



# Images

## I Introduction

L'objet de ce chapitre est d'aborder quelques unes des possibilités offertes par Python pour le traitement d'images numériques. Ces images peuvent être de deux types : les images dites matricielles (ou images "bitmap"), qui seront celles considérées dans ce chapitre, et les images vectorielles<sup>1</sup>. Une image matricielle, comme le dit le nom, correspond à une matrice de points à plusieurs dimensions ; chacune d'entre elle peut être associée à une dimension spatiale, mais également temporelle<sup>2</sup>, ou autre (une longueur d'onde pour une caméra multi-spectrale telle que MUSE du VLT, un niveau de résolution, une composante de couleur [RGB ou autre], ...). On ne considèrera ici que des images (spatialement) bidimensionnelles, l'extension aux dimension(s) supplémentaire(s) étant directe (mais la visualisation des résultats plus compliquée). Le codage des couleurs (éventuelles) est supposé correspondre au système RGB. Dans ce contexte, une image n'est rien d'autre qu'un tableau dont les deux premières dimensions correspondent aux nombres de pixels dans les directions verticale et horizontale (de haut en bas, et de droite à gauche). Si les éléments de ce tableau sont des scalaires, ils correspondent au codage des pixels en niveau de gris (avec 0 pour le noir, 255 pour le blanc pour le codage sur 8 bits). Si les éléments sont des triplets (voire des quadruplets dans le cas d'un codage des couleurs dans le système RGBA, le A correspondant au contrôle de la transparence), le premier élément du triplet correspond au "niveau de rouge", le second au "niveau de vert", le troisième au "niveau de bleu ; le noir correspond au triplet (0, 0, 0), le blanc à (255, 255, 255) (sur 8 bits). À ce stade, l'image est entièrement caractérisée. Traiter une telle image consiste donc à manipuler les données présentes dans ce tableau. L'objet de ce chapitre est de présenter quelques unes des manipulations aisément accessibles à l'aide certaines librairies Python.

Pour ce qui est du rendu de l'image, sur un dispositif supposé supporter le système RGB (un écran par exemple), sans instruction particulière, "l'ordinateur" est libre de "choisir" comment afficher l'image : échelle de couleur, transformations permettant d'afficher "en plein écran" (alors que l'image, elle, à une dimension parfaitement définie en terme de pixels)... Ces aspects peuvent être contrôlés en général par l'opérateur, par exemple, pour la fonction `matplotlib.pyplot.imshow`, à l'aide des arguments optionnels associés au choix d'une échelle de couleur particulière (`colormap`, mot-clé : `cmap`), d'un processus d'interpolation (mot-clé : `interpolation`), ... Le lecteur est invité à garder cette problématique à l'esprit pour obtenir des rendus "sympathiques".

## II Manipulations élémentaires

Le programme ci-après (`images_basic.py`) présente quelques-unes des opérations les plus élémentaires permettant de manipuler des images à l'aide de la librairie `imageio` (qui "arrive" avec un lot d'images) et `scipy.ndimage`. Les résultats correspondants sont présentés sur la figure Fig. 14.

```
1 #!/usr/bin/env python3
2 # -*- coding: utf-8 -*-
3
4 """
5 Manipulations élémentaires d'images
6 """
7
8 # Importation des librairies
9 import sys
10 import urllib3
11 import imageio
12 import scipy.ndimage as scndim
```

1. Ces images doivent être converties en images matricielles avant de pouvoir être visualisées, soit à l'aide d'un logiciel approprié (en général le logiciel ayant créé cette image), soit éventuellement avec des librairies Python (si de telles librairies ont été développées!). On peut ainsi citer la librairie `pdf2image` pour convertir un "fichier" PDF ou `cairosvg` pour un "fichier" SVG.

2. Pensez à un oscilloscope !

```

13 import copy
14 import numpy as np
15 import matplotlib.pyplot as plt
16
17 # Definition des fonctions
18 def get_image(img_name, pilmode=None, url=None):
19     """ Recupere une image, avec telechargement eventuel et copie locale """
20     try:
21         img = imageio.imread(img_name, pilmode=pilmode)
22     except OSError: # File does not exist locally
23         try:
24             img = imageio.imread(url, pilmode=pilmode)
25             imageio.imwrite(img_name, img) # Copy the image locally
26         except urllib3.HTTPError: # File cannot be downloaded
27             sys.exit("Unable to (down)load the image")
28     return img
29
30 def get_tags(img):
31     """ Recupere des caracteristiques de l'image sous la forme d'un tuple """
32     tags = img.shape, img.dtype
33     print("Format de l'image : {}\nType des elements : {}".format(tags[0], tags[1]))
34     return tags
35
36 def basic_IO(img):
37     """ Lecture et escriture d'une image """
38     # Recuperation des caracteristiques de l'image
39     tags = get_tags(img)
40     if tags[1] == 'uint8': # image codee sur 2**8 bits (0-255)
41         # Compute the reverse image
42         img_rev = 255 - img
43         # Distinction nuances de gris / RGB
44         if len(tags[0]) == 2: # image 2d en niveaux de gris
45             img = np.expand_dims(img, axis=2) # homogeneise le format
46             img_rev = np.expand_dims(img_rev, axis=2) # avec le cas couleur
47             cmap = 'gray'
48         elif len(tags[0]) == 3: # image 2d en couleur (RGB)
49             cmap = 'Reds'
50         else: # image de dimensionnalite superieure
51             sys.exit("Unable to process this image")
52     # Visualisation
53     fig, axes = plt.subplots(1,3,figsize=(8, 4))
54     fig.subplots_adjust(wspace=0.5)
55     axes[0].imshow(img[:, :, 0])
56     axes[1].imshow(img[:, :, 0], cmap = cmap)
57     axes[2].imshow(img_rev[:, :, 0], cmap = cmap)
58     axes[0].set_title('Default cmap')
59     axes[1].set_title('cmap = '+cmap)
60     axes[2].set_title('Reversed')
61     # Save and display
62     plt.savefig('images_basic_IO.png')
63     plt.show()
64 else:
65     # Do whatever you want...
66     sys.exit("This kind of image is not implemented yet")
67 return
68
69 def basic_colorspace(img):
70     """ Pour se familiariser avec l'espace des couleurs RGB """
71     # Recuperation des caracteristiques de l'image
72     tags = get_tags(img)
73     # Visualisation
74     fig, axes = plt.subplots(1,4,figsize=(16, 4))
75     fig.subplots_adjust(wspace=0.5)
76     axes[0].imshow(img)

```

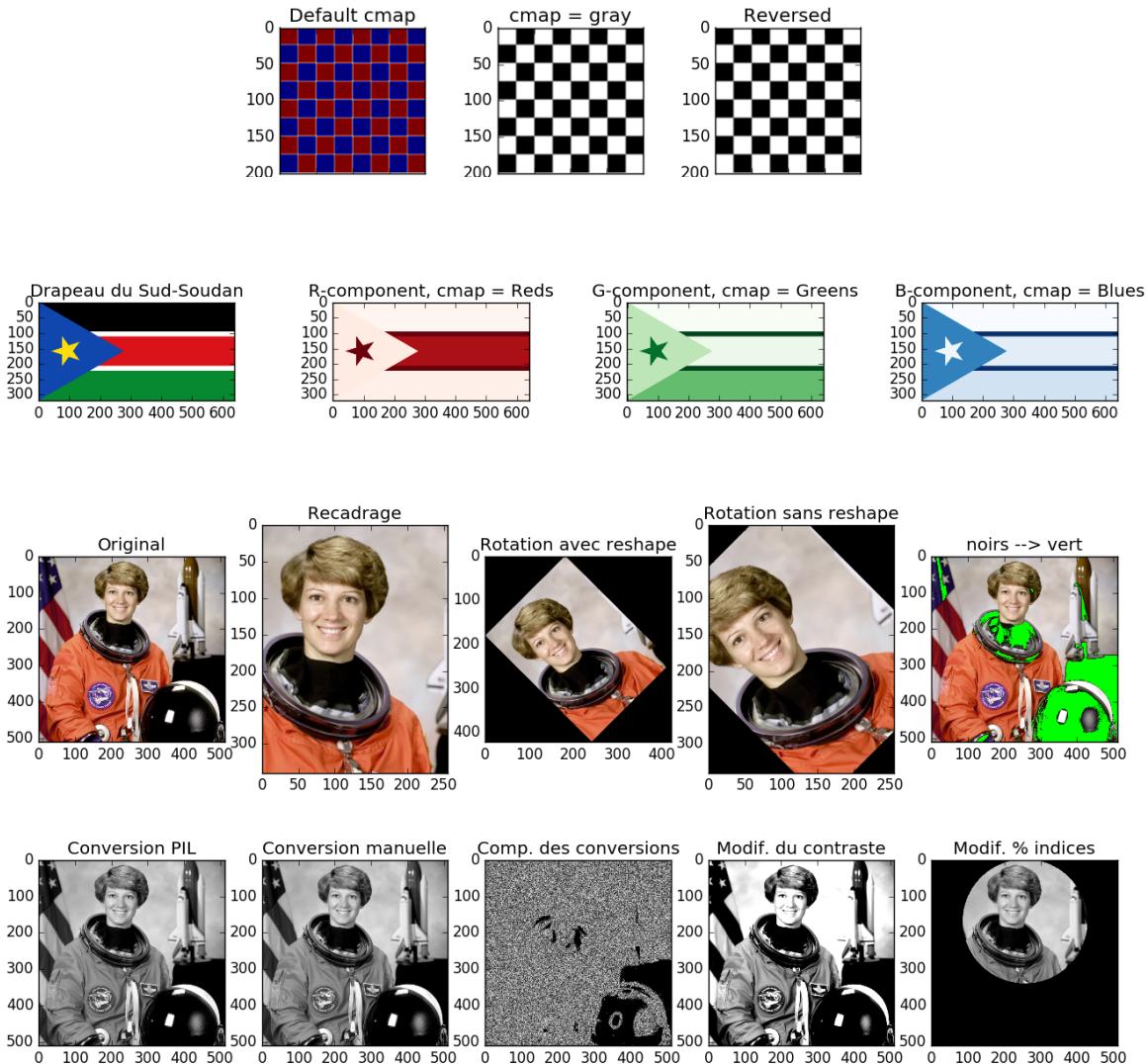
```

77     axes[1].imshow(img[:, :, 0], cmap='Reds')
78     axes[2].imshow(img[:, :, 1], cmap='Greens')
79     axes[3].imshow(img[:, :, 2], cmap='Blues')
80     axes[0].set_title('Drapeau du Sud-Soudan')
81     axes[1].set_title('R-component, cmap = Reds')
82     axes[2].set_title('G-component, cmap = Greens')
83     axes[3].set_title('B-component, cmap = Blues')
84     # Save and display
85     plt.savefig('images_basic_colorspace.png')
86     plt.show()
87     return
88
89 def basic_transfo(img, img_gray):
90     """ Transformations basiques """
91     # Recuperation des caracteristiques de l'image originale
92     tags = get_tags(img)
93     lx, ly = tags[0][0:2]
94     # Quelques statistiques sur l'image en niveaux de gris
95     gray_mean = img_gray.mean()
96     gray_max = img_gray.max()
97     gray_min = img_gray.min()
98     # Conversion a la main de l'image originale en niveaux de gris
99     # Comme l'indique la commande print(imageio.help('png')), "when translating
100    # a color image to grayscale (mode 'L', 'I' or 'F'), the library uses the
101    # ITU-R 601-2 luma transform: L = R * 299/1000 + G * 587/1000 + B * 114/1000
102    img_convert = np.dot(img, [0.299, 0.587, 0.114])
103    # Recadrage
104    img_crop = img[:2*lx//3, ly//4:-ly//4]
105    # Rotation
106    img_rot_wr = scndim.rotate(img_crop, 45)
107    img_rot_nr = scndim.rotate(img_crop, 45, reshape=False)
108    # Masques
109    # --> remplacement des "noirs profonds" par du vert
110    img_green = copy.copy(img) # attention au caractere mutable...
111    img_green[img_gray<10] = [0, 255, 0]
112    # --> remplacement par du noir base sur les indices
113    img_mask = copy.copy(img_gray) # attention au caractere mutable...
114    xmask, ymask = np.ogrid[0:lx, 0:ly]
115    mask = (xmask-lx/3)** 2 + (ymask-ly/2)** 2 > lx*ly/9
116    img_mask[mask] = 0
117    # Visualisation
118    fig, axes = plt.subplots(2,5,figsize=(16, 8))
119    axes[0,0].imshow(img)
120    axes[0,1].imshow(img_crop)
121    axes[0,2].imshow(img_rot_wr)
122    axes[0,3].imshow(img_rot_nr)
123    axes[0,4].imshow(img_green)
124    axes[1,0].imshow(img_gray, cmap='gray')
125    axes[1,1].imshow(img_convert, cmap='gray')
126    axes[1,2].imshow(img_convert-img_gray, cmap='gray')
127    axes[1,3].imshow(img_gray, cmap='gray', vmin = (gray_min+gray_mean)/2,\n                                vmax = (gray_max+gray_mean)/2)
128    axes[1,4].imshow(img_mask, cmap='gray')
129    axes[0,0].set_title('Original')
130    axes[0,1].set_title('Recadrage')
131    axes[0,2].set_title('Rotation avec reshape')
132    axes[0,3].set_title('Rotation sans reshape')
133    axes[0,4].set_title('noirs --> vert')
134    axes[1,0].set_title('Conversion PIL')
135    axes[1,1].set_title('Conversion manuelle')
136    axes[1,2].set_title('Comp. des conversions')
137    axes[1,3].set_title('Modif. du contraste')
138    axes[1,4].set_title('Modif. % indices')
139    # Save and display
140
```

```

141 plt.savefig('images_basic_transfo.png')
142 plt.show()
143 return
144
145 # Programme principal
146 if __name__ == "__main__":
147     # Manipulation de base
148     img = get_image('imageio:checkerboard.png')
149     #img = get_image('imageio:astronaut.png')
150     basic_IO(img)
151     # Espace des couleurs (visuel avec le drapeau du Sud-Soudan ou des Seychelles)
152     img = get_image('640px-Flag_of_South_Sudan.svg.png', \
153                     url='https://upload.wikimedia.org/wikipedia/commons/\
154                     /thumb/7/7a/Flag_of_South_Sudan.svg/\
155                     640px-Flag_of_South_Sudan.svg2')
156     basic_colorspace(img)
157     # Transformations de base
158     img = get_image('imageio:astronaut.png') # Eileen Collins
159     img_gray = get_image('imageio:astronaut.png', pilmode='L') # en echelle de gris
160     basic_transfo(img, img_gray)

```

FIGURE 14 – Résultats obtenus à partir du script `images_basic.py`.

On notera à cette occasion (par exemple sur l'image correspondant à `img_mask[mask]`) que la fonction

`plt.imshow` place l'origine de l'image dans le coin supérieur gauche, contrairement au positionnement “standard”, ce qui est parfois à l'origine d'un peu de confusion (*cf* figure Fig. 15).

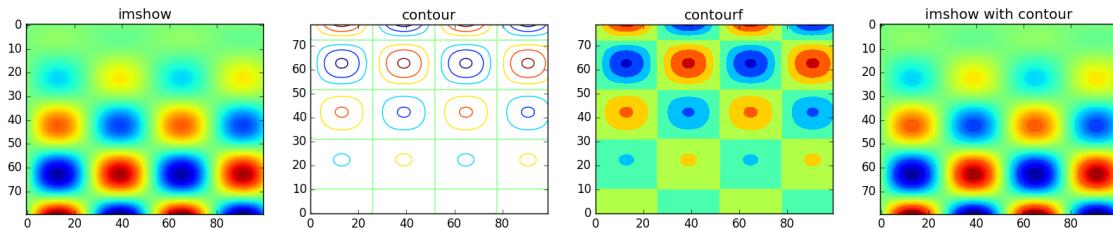


FIGURE 15 – Illustration du positionnement de l'origine dépendant des fonctions mises en œuvre.

## III Manipulations avancées

### III - 1 Filtrage et autre

Il est bien entendu beaucoup plus intéressant d'aller au-delà des manipulations illustrées dans le paragraphe précédent, afin de parvenir à extraire des informations présentes dans l'image. Pour ne citer que quelques unes des idées naturelles, on peut vouloir “rendre plus nette” une image, ou à l'inverse chercher à la “flouter” (étape en général indispensable avant toute opération dans laquelle intervient des estimations de gradient, de laplacien, ...), éliminer des structures parasites (le “bruit” ou des détails non pertinents), reconnaître certaines formes, considérer certains aspects de topologie... Les justifications théoriques des opérations à effectuer s'obtiennent dans un cadre dépassant très largement celui de ce cours (filtrage, morphologie mathématique, ...), et il n'est donc pas question de les présenter. Tout comme il n'est pas question d'effectuer une redite des informations présentes dans les documentations des librairies correspondantes. Le lecteur est donc invité à consulter (notamment) les pages web présentant<sup>3</sup> les librairies `scipy.ndimage`, `scikit`, ou des librairies tierces<sup>4</sup> (pour lesquelles très souvent une interface avec Python existe d'ores-et-déjà), et les exemples qui sont proposés. Une illustration des résultats aisément accessibles est présentée sur la figure Fig. 17.

```

1 #!/usr/bin/env python3
2 # -*- coding: utf-8 -*-
3
4 """
5 Manipulations avancees d'images
6 """
7
8 # Importation des librairies
9 import urllib3
10 import imageio
11 import scipy.ndimage as scndim
12 import cv2
13 import copy
14 import numpy as np
15 import matplotlib.pyplot as plt
16 import images_basic
17
18
19 # Definition des fonctions
20 def img_filters(img):
21     " Pour illustrer quelques fonctionnalites de filtrage "
22     # Add some gaussian noise
23     gauss_level = 10.
24     img_gauss = img + gauss_level * img.std() * np.random.random(img.shape)
25     # Add some peppered-salted noise

```

3. <https://docs.scipy.org/doc/scipy/reference/ndimage.html> et <https://scikit-image.org/>

4. OPENCV en est une, *cf* [https://opencv-python-tutorials.readthedocs.io/en/latest/py\\_tutorials.html](https://opencv-python-tutorials.readthedocs.io/en/latest/py_tutorials.html)

```

26 p_level = 0.1
27 s_level = 0.1
28 mask_p = np.random.choice(a=[True, False], size=img.shape, p=[p_level, 1-p_level])
29 mask_s = np.random.choice(a=[True, False], size=img.shape, p=[s_level, 1-s_level])
30 img_ps = copy.copy(img)
31 img_ps[mask_p] = 0
32 img_ps[mask_s] = 255
33 # Visualisation
34 fig, axes = plt.subplots(4,3,figsize=(1.5*3*3, 1.*4*3))
35 for i, im in enumerate([img, img_gauss, img_ps]):
36     axes[0,i].imshow(im, cmap = 'gray')
37     axes[1,i].hist(im.ravel(),256,[0,256])
38     axes[1,i].set_xlim(0,256)
39     axes[2,i].imshow(scndim.gaussian_filter(im, 2), cmap = 'gray')
40     axes[3,i].imshow(scndim.median_filter(im, 3), cmap = 'gray')
41 # Le decorum
42 for axe in np.ravel(axes):
43     #axe.axis('off') # not working as label is removed too!
44     axe.set_xticks([])
45     axe.set_yticks([])
46 axes[0,0].set_title('Original (B&W)')
47 axes[0,1].set_title('Bruit gaussien')
48 axes[0,2].set_title('Bruit poivre et sel')
49 axes[1,0].set_ylabel('Histogramme des niveaux')
50 axes[2,0].set_ylabel('Gaussian filter')
51 axes[3,0].set_ylabel('Median filter')
52 # Save and display
53 plt.subplots_adjust(wspace=0.02, hspace=0.02, top=0.95, bottom=0, \
54                     left=0.05, right=0.95)
55 plt.savefig('images_advanced_filters.png')
56 plt.show()
57 return
58
59 def img_edges(img):
60     " Pour illustrer quelques fonctionnalites de detection des contours "
61     # Blur the image
62     img.blur = cv2.GaussianBlur(img,(27,27),0)
63     # Set the output frame
64     fig, axes = plt.subplots(4,3,figsize=(1.5*3*5, 1.*4*5))
65     # Detect edges
66     for idepth, depth in enumerate([cv2.CV_64F, cv2.CV_8U]):
67         img_lapl = cv2.Laplacian(img.blur,depth) # Laplacian filter
68         img_sobX = cv2.Sobel(img.blur,depth,1,0,ksize=5) # Sobel filter along X
69         img_sobY = cv2.Sobel(img.blur,depth,0,1,ksize=5) # Sobel filter along Y
70     # Visualisation
71     for i, im in enumerate([img_lapl, img_sobX, img_sobY]):
72         axes[0+2*idepth,i].imshow(im, cmap = 'gray')
73         axes[1+2*idepth,i].hist(im.ravel(),256,[0,256])
74 # Le decorum
75 for axe in np.ravel(axes):
76     #axe.axis('off') # not working as label is removed too!
77     axe.set_xticks([])
78     axe.set_yticks([])
79 axes[0,0].set_title('Filtre laplacien', fontsize=26)
80 axes[0,1].set_title('Filtre de Sobel %X', fontsize=26)
81 axes[0,2].set_title('Filtre de Sobel %Y', fontsize=26)
82 axes[1,0].set_ylabel('Histogramme des niveaux', fontsize='xx-large')
83 axes[3,0].set_ylabel('Histogramme des niveaux', fontsize='xx-large')
84 # Save and display
85 plt.subplots_adjust(wspace=0.02, hspace=0.02, top=0.95, bottom=0.05, \
86                     left=0.05, right=0.95)
87 plt.savefig('images_advanced_edges.png')
88 plt.show()
89 return

```

```

90
91 # Programme principal
92 if __name__ == "__main__":
93     # Recuperation de l'image
94     img = images_basic.get_image('Tsunami_by_hokusai_19th_century.jpg', pilmode='L', \
95                                 url=('https://upload.wikimedia.org/wikipedia/commons/\
96                                     'a/a5/Tsunami_by_hokusai_19th_century.jpg'))
97     # Filtrage
98     #img_filters(img)
99     # Detection des contours
100    img_edges(img)

```

### III - 2 Conversion image-espace réel

Certaines situations exigent de convertir la position des pixels dans l'image en position dans l'espace réel. Si le plan observé (le plan "objet" en optique géométrique) est orthogonal à l'axe optique défini par l'appareil qui enregistre l'image, les distorsions correspondent aux aberrations géométriques classiques (coussinet, bâillet, ...). En fonction de la qualité de l'objectif utilisé, celles-ci peuvent être plus ou moins importantes<sup>5</sup>, mais peuvent être (relativement) aisément corrigées<sup>6</sup>. Ces distorsions corrigées, il reste à prendre en compte les effets de perspective, qui apparaissent donc lorsque le plan observé n'est pas orthogonal<sup>7</sup> à l'axe optique. Pour cela, la méthode standard consiste à faire figurer dans le plan d'observation un objet dont les dimensions sont connues (une "mire"), à repérer sur l'image la position (d'un nombre suffisant) de points particuliers de cette mire, et d'effectuer enfin la conversion "pixel → position dans l'espace réel" permettant de "redresser" l'image. La librairie tierce OpenCV permet d'effectuer aisément ces opérations<sup>8</sup>, et une illustration est proposée sur la figure Fig. 16.

```

1  #!/usr/bin/env python3
2  # -*- coding: utf-8 -*-
3
4 """
5 Conversion image/espace reel pour une image
6 """
7
8 # Importation des librairies
9 import urllib3
10 import imageio
11 import cv2
12 import numpy as np
13 import matplotlib.pyplot as plt
14 import matplotlib.patches as patches
15 import itertools
16 import images_basic
17
18
19 # Definition des fonctions
20 def spatial_calib(img, pts1, pts2):
21     " Geometric transformation for camera calibration "
22     # Projection orthographique
23     M = cv2.getAffineTransform(pts1[0:3], pts2[0:3])
24     img_ortho = cv2.warpAffine(img, M, (1800, 1800))
25     # Avec la perspective
26     M = cv2.getPerspectiveTransform(pts1, pts2)
27     img_persp = cv2.warpPerspective(img, M, (1800, 1800))
28     # Visualisation

```

5. C'est en particulier le cas pour les objets qui tiennent plus du sténopé (webcam, ...) que d'un matériel d'optique digne de ce nom.

6. C'est ce que font (presque automatiquement) les logiciels de photographie type Adobe® Lightroom, Darktable, ... à partir des données Exif attachées à l'image considérée et des données constructeurs.

7. À l'évidence, on supposera que les problématiques de profondeur de champ et de netteté de l'image ont été résolues à ce stade.

8. Cette même librairie offre même la possibilité de repérer, de façon automatique, les points particuliers correspondant à une mire de type "grille de pastilles" (cv2.findCirclesGrid) ou de type "échiquier" (cv2.findChessboardCorners).

```

29 fig, axes = plt.subplots(1,3,figsize=(12,4))
30 # --> les images
31 axes[0].imshow(img)
32 axes[1].imshow(img_ortho)
33 axes[2].imshow(img_persp)
34 # --> les points particuliers
35 radius = 50
36 colors = ['r', 'g', 'b', 'm']
37 c_colors = itertools.cycle(colors)
38 for pt in pts1:
39     circ = patches.Circle(pt, radius, color=next(c_colors), alpha=0.4)
40     axes[0].add_patch(circ)
41 # Le decorum
42 axes[0].set_title('Original')
43 axes[1].set_title('Correction orthographique')
44 axes[2].set_title('Correction complete')
45 # Save and display
46 plt.savefig('images_geo.png')
47 plt.show()
48 return
49
50 # Programme principal
51 if __name__ == "__main__":
52     # Recuperation de l'image: un echiquier en perspective
53     img = images_basic.get_image('ChessStartingPosition.jpg', \
54                                 url=('https://upload.wikimedia.org/wikipedia/commons/\
55                               '3/30/ChessStartingPosition.jpg'))
56     # Definition des points particuliers sur lesquels s'effectue la correction
57     # de la perspective
58     # --> 4 points particuliers sur l'image
59     pts1 = np.float32([[570,1758],[2922,1387],[151,878],[1715,874]])
60     # ... correspondant aux 4 points suivants dans l'espace reel
61     # (pour un echiquier 1600x1600)
62     pts2 = np.float32([[100,1700],[1700,1700],[100,100],[1300,500]])
63     # Redressement
64     spatial_calib(img, pts1, pts2)

```

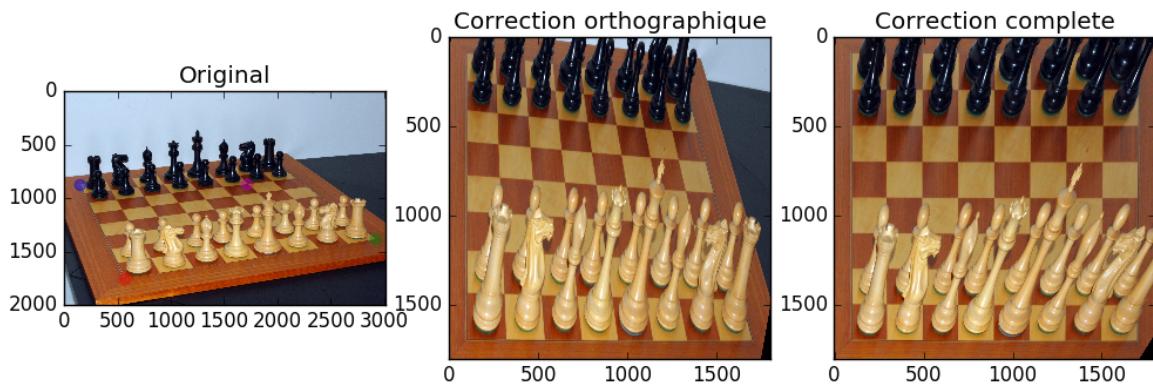


FIGURE 16 – Résultats obtenus à partir du script `images-geo.py`. La projection orthographique (souvent utilisée en architecture, dessin industriel, ...) correspond à la situation où le centre optique est situé à l'infini, tous les rayons lumineux étant alors parallèles à l'axe optique. Elle n'est adaptée qu'aux situations où l'objet observé est “petit et distant”.

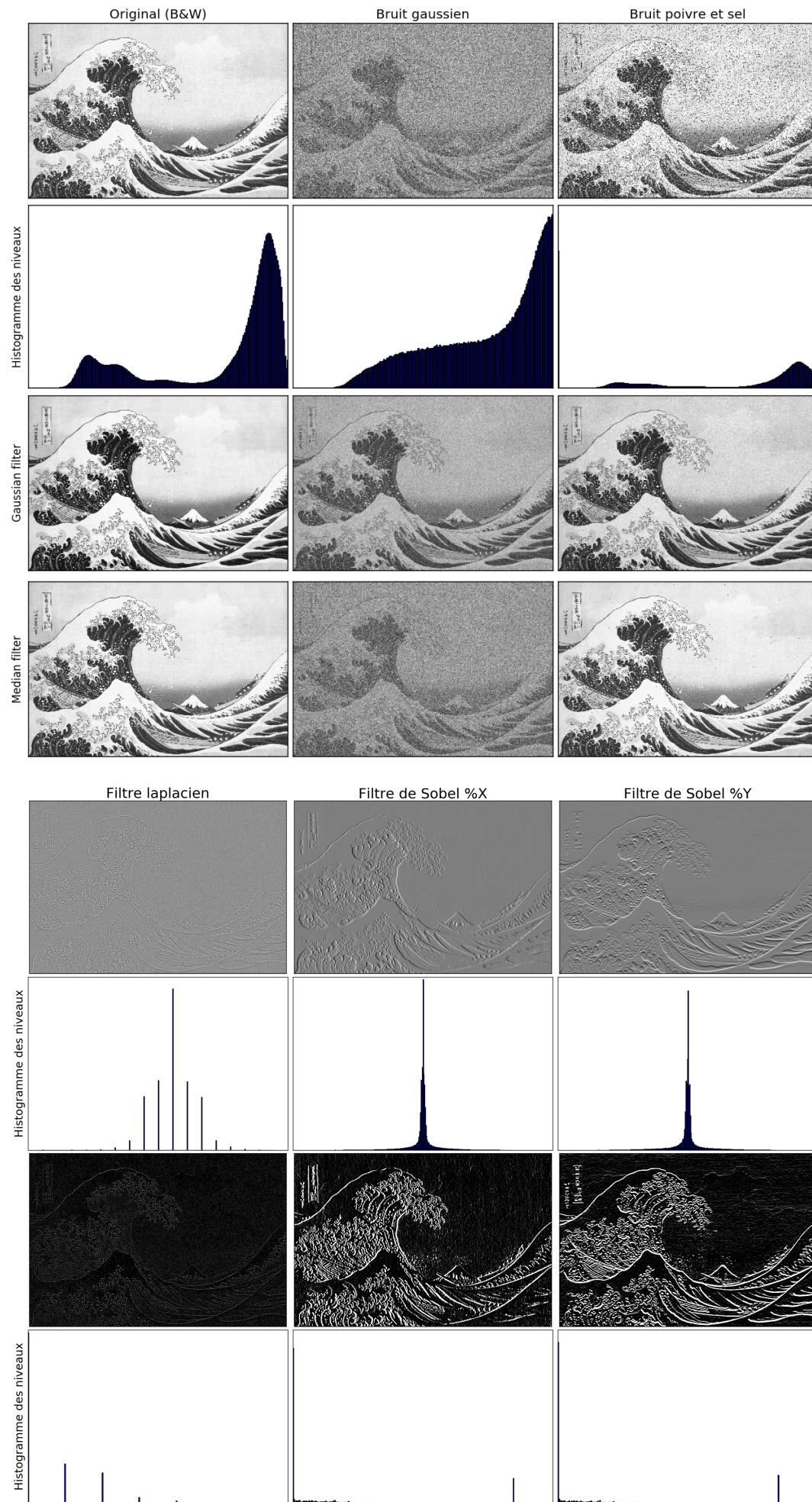


FIGURE 17 – Résultats obtenus à partir du script `images_advanced.py`.

### III - 3 Exercice

#### III - 3.1 Énoncé

Sur la figure de votre choix, proposer un traitement d'image permettant d'extraire une information quantitative.

#### III - 3.2 Corrigé

```

1 #!/usr/bin/env python3
2 # -*- coding: utf-8 -*-
3
4 """
5 Descriptif du fichier
6 """
7
8 # Importation des librairies
9
10 # Definition des fonctions
11
12 # Programme principal
13 if __name__ == "__main__":
14     pass

```

Un exemple est présenté ci-dessous. Sur une idée de Ruben Staub, il s'agit de “vérifier les chiffres” présent dans un article<sup>9</sup> concernant la déforestation d'une région du Brésil, l'Apyterewa. L'image originale est récupérée sur Google. Deux étapes sont nécessaires : la première consiste à détourer la région concernée, identifiée grâce au liseret rouge qui en marque la frontière. La seconde consiste à identifier les régions de couleur vert clair, assimilées à des terres déforestées, par opposition aux régions vert sombre, couvertes de forêt primaire, puis à déterminer la proportion relative. Tout le travail est effectué à l'aide de contours. Le résultat final donne une estimation de la fraction déforestée de 11.1% (`sigma=3, level=0.2`) ou de 11.7% (`sigma=2, level=0.2`), les paramètres `sigma` et `level` étant des paramètres libres permettant d'optimiser (à l'œil) la sélection des régions. Les images correspondant au détourage du territoire Apyterewa et des régions déforestées sont présentées sur la figure Fig. 18.

---

<sup>9</sup>. [https://news.mongabay.com/2018/07/brazils-political-storm-driving-amazon-deforestation-higher/#attachment\\_208140](https://news.mongabay.com/2018/07/brazils-political-storm-driving-amazon-deforestation-higher/#attachment_208140).

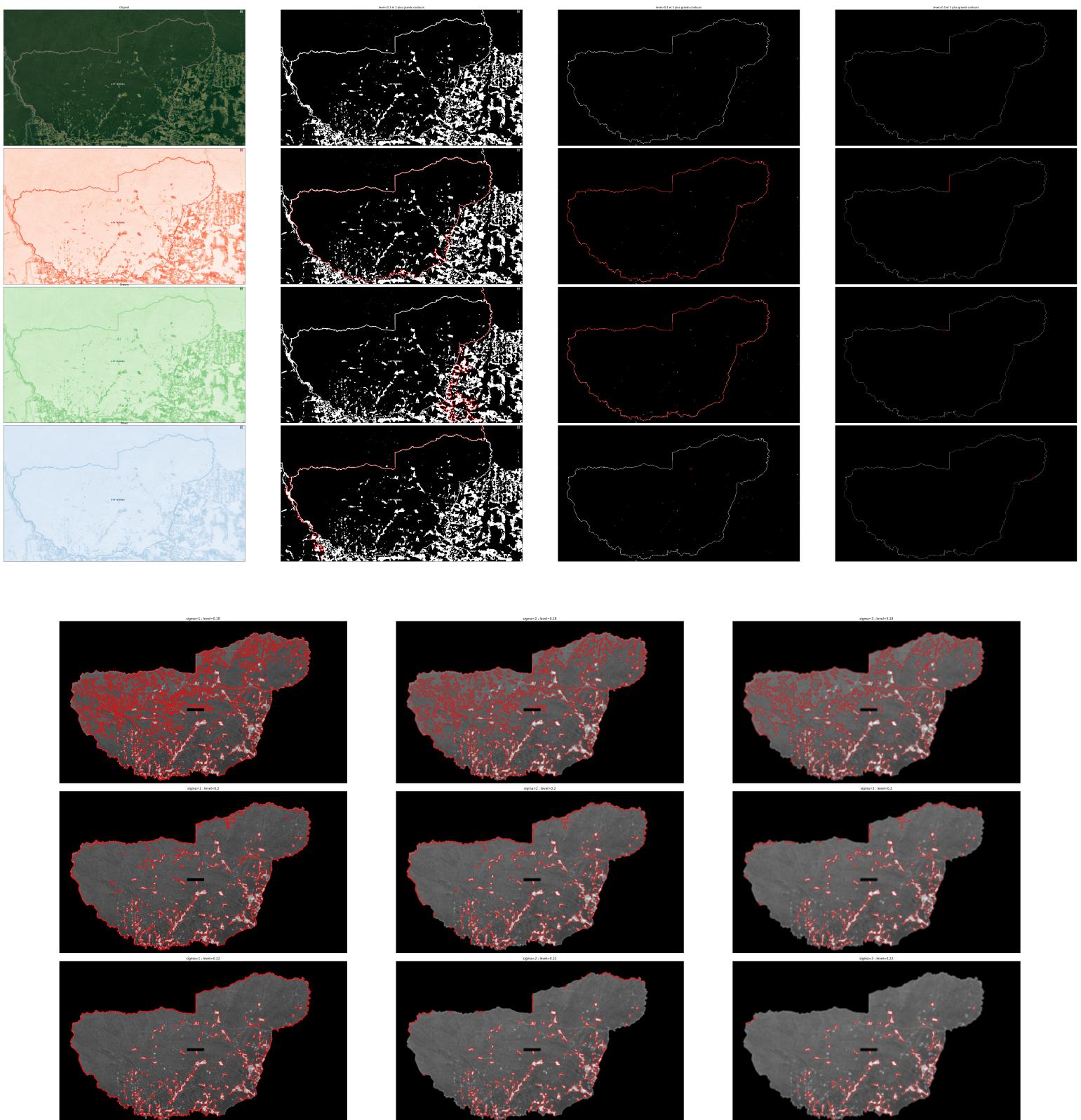


FIGURE 18 – Résultats obtenus pour l'analyse de la déforestation de l'Apyterewa.

