Calcul scientifique avec Python/Numpy/matplotlib
Guide de démarrage rapide

Module M3202C
Modélisation Mathématique

1 Généralités

Dans ce module de modélisation mathématique, nous implémenterons des algorithmes et effectuerons de nombreux calculs numériques. Nous devrons également visualiser les résultats de nos expérimentations au moyens de graphiques. Par manque de temps, il ne sera pas possible de programmer tous nos modèles en Java ou en C. Par ailleurs, les tableurs des suites bureautiques ne sont pas adaptés à une véritable démarche de calcul scientifique. Il existe des logiciels dédiés au calcul numérique spécialement conçus pour faciliter le travail de programmation du scientifique, même pour l'utilisateur ne disposant pas d'une formation classique de développeur. On peut citer, entre autres Matlab, Scilab, R, Julia, Octave etc … . Ces environnements de travail combinent un langage de programmation de haut niveau avec des librairies de calcul scientifique pour l'ingénieur. Dans ce module, nous utiliserons une voie intermédiaire entre ces deux approches. Le langage `python` (dans sa version 2.7) sera complété par la librairie de calcul numérique avancé `numpy`. Les résultats seront visualisés grâce à `matplotlib`, une librairie graphique en python basée sur `numpy`. Le tout sera associé au sein de l'environnement de développement intégré `spyder`.

Python est un langage de programmation généraliste interprété et non pas compilé : le code à exécuter ne se présente pas sous la forme d'un binaire exécutable. Un interpréteur se charge d'exécuter le code source à la manière d'un script shell. Il existe plusieurs distributions complètes de `python`, c'est-à-dire un interpréteur et des librairies, pour la plupart des systèmes d'exploitation.

`Python` n'est pas spécifiquement adapté au calcul scientifique. Pour cela, le module `numpy` a été conçu pour étendre les fonctionnalités de base du langage afin de le rendre plus performant pour une utilisation en sciences de l'ingénieur. `Numpy` inclut de nouveaux types de données (essentiellement des tableaux numériques) et des fonctions pour les manipuler. La partie coûteuse du code de `numpy` est écrite en C en utilisant des algorithmes particulièrement bien optimisés. L'utilisateur n'a pas à rentrer dans le code C, mais se contente d'utiliser les fonctions de haut niveau déjà optimisées. Nous verrons que, bien que cela soit très efficace, il est nécessaire de changer son paradigme de programmation. Pareillement, les fonctionnalités graphiques de base de python peuvent aussi être étendues par la librairie `matplotlib` qui permet de visualiser les objets de `numpy` en restant à un niveau assez élevé. D'une manière générale, on doit s'efforcer de programmer «comme on pense» sans chercher à rentrer dans le détail de l'objet.

`Spyder` est l'un des environnements de développement intégré qui associe ces trois ingrédients pour un travail efficace. Il est constitué d'une console qui exécute l'interpréteur, d'un éditeur de code, de fenêtres graphiques et de divers outils de développement : navigateur de variables, historique de commande, etc … . Des menus facilitent les tâches d'administration. Le workflow classique consiste à entrer des commandes de manière interactive dans la console, de même qu'on le ferait dans un terminal, et à lire les résultats affichés par l'exécution de cette commande. On procède ainsi pour des calculs simples ou lorsqu'on est en phase d'exploration. Une fois que la suite de commandes est stable, on l'écrit dans un fichier source au moyen de l'éditeur, pour l'exécuter ensuite depuis la console ou directement depuis le système d'exploitation, comme

on le ferait pour un script bash. L'état d'une session en cours peut être sauvegardé pour pouvoir reprendre le travail ultérieurement à partir d'un point précis.

2 IDE spyder

Le logiciel se lance en tapant `spyder` dans le tableau de bord Unity, ou en le sélectionnant dans les menus. On obtient un bureau divisé en plusieurs parties comme sur la Figure 1.

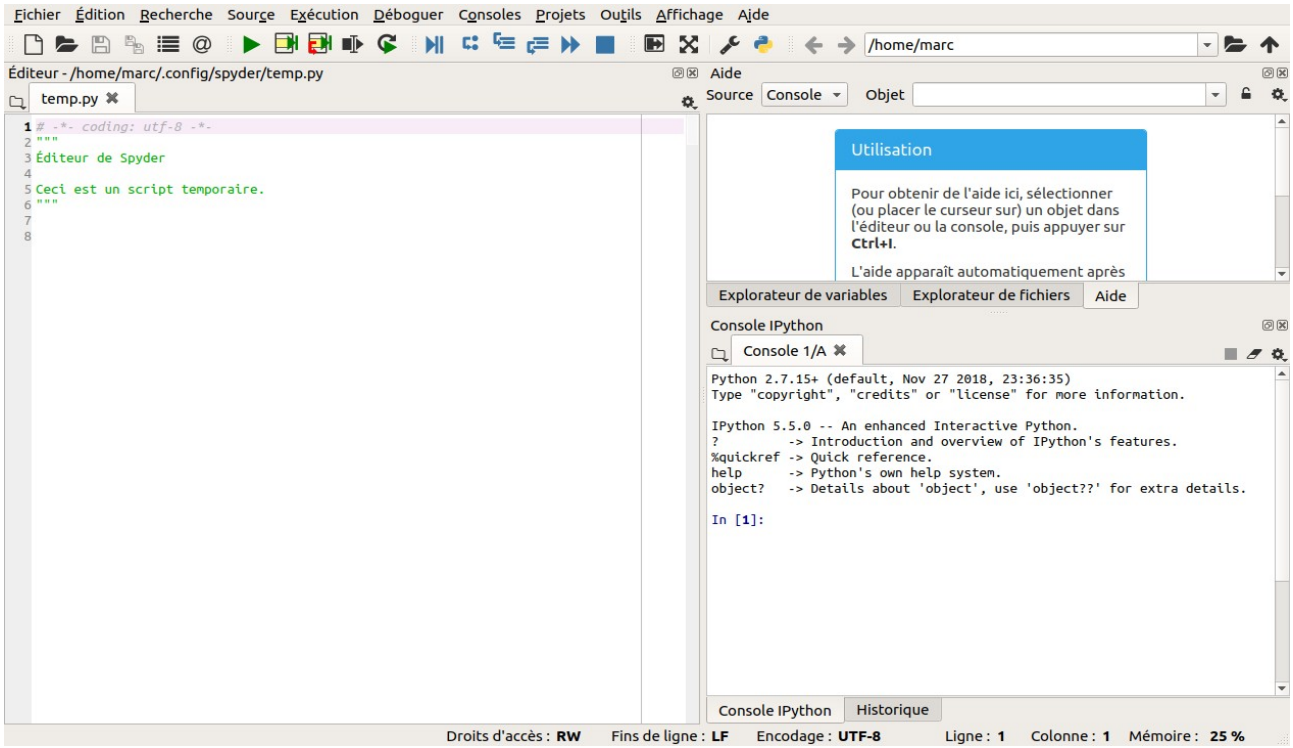


Figure 1: Bureau Spyder

On distingue alors

1. La **Console** en bas à droite. C'est dans cette fenêtre que l'on tapera les commandes interactives. L'historique de commandes est disponible dans l'onglet **Historique**.
2. L'**Aide** juste au-dessus de la console, ainsi que les deux onglets **Explorateur de variables** et **Explorateur de fichiers**. On utilisera principalement le premier pour inspecter les variables créées.
3. L'**Éditeur** de fichiers sur la partie gauche. Il permet d'écrire les fichiers sources et de les exécuter entièrement ou en partie.
4. **La barre de menus** en dessous du bandeau.
5. **La barre d'outils** en dessous de la précédente. On notera à la fin de cette barre le moyen de changer le répertoire courant de la console

À ces fenêtres de base s'ajouteront des fenêtres graphiques créées par les scripts. La disposition de toutes ces fenêtres peut être modifiée dans le menu **Affichage**. Noter tout de même l'existence de l'option **Affichage/Disposition de fenêtres personnalisées** pour rétablir la disposition par défaut en cas de problème.

3 Premiers pas

La suite de ce document ne constitue pas une introduction au langage **python**, mais plutôt une sélection de commandes qui seront utiles au projet. On pourra consulter avec profit des ouvrages plus complets pour un apprentissage systématique de **python**/**numpy**.

3.1 Calculs mathématiques de base

Python peut s'utiliser comme une simple calculette pour les opérations usuelles en tapant les calculs dans la console, à la suite de l'invite de commande :

```
In [1]: 2 * 6 + (-5 / 4)
Out[1]: 10
```

Nous observons tout de suite l'importance du type des objets : le calcul de la division a été fait par python en considérant que les opérandes sont des entiers. Pour effectuer le calcul avec des flottants, il faut que l'un au moins des opérandes de la division soit flottant :

```
In [2]: 2 * 6 + (-5. / 4) # noter le 5. au lieu de 5
Out[2]: 10.75
```

Par précaution, il est préférable de toujours utiliser l'écriture flottante (point décimal) pour des flottants. Notons aussi que, comme la plupart des langages, **python** utilise une représentation des flottants sur 53 bits. Il s'ensuit que le calcul en virgule flottante n'est pas toujours exact comme le montre l'exemple suivant :

```
In [3]: 0.1 + 0.2
Out[3]: 0.30000000000000004

In [4]: 0.1 + 0.2 - 0.3
Out[4]: 5.551115123125783e-17
```

Il ne s'agit pas d'une erreur de langage, mais de l'utilisation classique de la norme IEEE-754.

Le type d'une variable peut être interrogé :

```
In [10]: type(2)
Out[10]: int

In [11]: type(2.)
Out[11]: float
```

Python dispose aussi des types booléen et chaîne de caractères :

```
In [47]: 0.1 < 0.2
Out[47]: True

In [48]: type(0.1 < 0.2)
Out[48]: bool

In [49]: type('mot')
Out[49]: str
```

```
In [50]: type("""phrase sur
    ...: plusieurs
    ...: lignes"""
    ...: )
Out[50]: str
```

Les fonctions mathématiques usuelles ne font pas partie des commandes de base. Il faut importer un des modules qui les définit. Le plus courant de ces modules est `math`, mais nous allons directement utiliser `numpy` qui est énormément plus puissant. L'import d'un module rend disponible l'ensemble des fonctions qu'il définit. On peut utiliser l'une des trois syntaxes suivantes :

```
In [18]: import numpy

In [19]: numpy.sqrt(3)
Out[19]: 1.7320508075688772
```

Toutes les fonctions de `numpy` sont importées. L'appel se fait en donnant le nom complet de la fonction, au sein de son module. On peut se dispenser du nom de module en important de la manière suivante :

```
In [18]: from numpy import *

In [19]: sqrt(3)
Out[19]: 1.7320508075688772
```

Cette pratique n'est pas recommandée car elle génère de la confusion lors de l'appel d'une fonction. Enfin, on peut également abréger le nom du module. C'est ce que nous ferons dans ce document :

```
In [29]: import numpy as np

In [30]: np.sqrt(3)
Out[30]: 1.7320508075688772
```

Dorénavant, nous supposerons que la ligne [29] ci-dessus a été exécutée au début de chaque travail. Nous disposerons alors des fonctions racine carrée (`np.sqrt`), exponentielle (`np.exp`), cosinus et sinus (`np.cos` et `np.sin`) ainsi que de la variable `np.pi` et bien d'autres.

Dans `spyder`, on peut bénéficier de la complétion automatique au moyen de la touche `TAB` :

```
In [2]: np.cos
              np.cos
              np.cosh
```

Dans cet exemple, l'appui sur la touche `TAB` après `np.cos` fait apparaître les fonctions du module dont le nom commence par `cos`.

3.2 Affectation

Les variables n'ont pas à être déclarées mais sont créées automatiquement au moment de leur affectation par le signe `=`

```
In [26]: a = 2 + np.sqrt(3)

In [27]: b = 2 - np.sqrt(3)

In [28]: develop =  a**2 + b**2

In [29]: develop
Out[29]: 14.0
```

Nous constatons alors que trois variables sont apparues dans l'explorateur de variables. Il est possible, mais pas recommandé, de modifier la valeur des variables directement depuis cet explorateur. De plus, le nouveau type de donnée **float64** a également été défini par le module **numpy** à l'importation :

```
In [10]: type(a)
Out[10]: numpy.float64
```

La tentative d'utilisation d'une variable non affectée produit une erreur :

```
In [30]: developp
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-30-2e5ae570986d> in <module>()
----> 1 developp

NameError: 'developp' is not defined
```

3.3 tuples et listes

Python possède plusieurs conteneurs : les listes, les chaînes de caractères, les tuples, les dictionnaires et les ensembles. Nous utiliserons fréquemment les tuples qui sont des listes indexables d'éléments de type éventuellement différents :

```
In [68]: t = ('rouge', 255, 0, 0, 0.1)

In [69]: type(t)
Out[69]: tuple

In [70]: type(t[0])
Out[70]: str

In [71]: type(t[1])
Out[71]: int

In [72]: type(t[4])
Out[72]: float
```

Les indices commencent à 0. La différence essentielle avec les listes **python** est que les éléments d'un tuple ne sont pas modifiables individuellement. Il faut affecter une valeur à l'ensemble du tuple :

```
In [74]: t[2] = 0
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-74-a2a35ef6d23c> in <module>()
----> 1 t[2] = 0

TypeError: 'tuple' object does not support item assignment

In [75]: t = ('bleu', 255, 0, 0, 0.1)

In [76]: t
Out[76]: ('bleu', 255, 0, 0, 0.1)
```

À l'inverse, le « dépaquetage » d'un tuple permet d'affecter plusieurs variables en une fois :

```
In [110]: couleur, R, V, B, alpha = t # dépaquetage de t en 4 valeurs

In [111]: couleur # couleur a bien reçu t[0]
Out[111]: 'bleu';

In [112]: alpha # et alpha t[3]
Out[112]: 0.1
```

Nous n'utiliserons que très peu les listes natives de **python**, qui se définissent par :

```
In [77]: l = ['bleu', 255, 0, 0, 0.1]

In [78]: type(l)
Out[78]: list
```

3.4 Tableaux numpy (utilisation de base)

La puissance de **numpy** réside dans le traitement efficace des tableaux. Pour cela, **numpy** définit un type **array** qui est essentiellement une liste dont les éléments sont obligatoirement de même type.

```
In [79]: tab = np.array([1, 0, 5])

In [80]: type(tab)
Out[80]: numpy.ndarray

In [81]: type(tab[0])
Out[81]: numpy.int64
```

Dans l'explorateur de variables **spyder**, nous voyons apparaître une nouvelle variable de type **array**, dont les éléments sont tous de type **int64** (entiers longs de **numpy**). Un double-clic sur le nom du tableau permet de visualiser ou modifier (pas recommandé) les éléments. Le type des éléments peut être forcé à la création :

```
In [82]: tab = np.array([255, 0, 2], dtype = np.uint8)
```

```
In [83]: tab[0] + tab[2]
/usr/bin/ipython:1: RuntimeWarning: overflow encountered in ubyte_scalars
  #! /bin/sh
Out[83]: 1
```

Ici, on a défini un tableau d'entiers non-signés sur 8 bits (**uint8**), c'est-à-dire des entiers entre 0 et 255. La somme 255 + 2 est donc égale à 1 dans ce cas, avec un dépassement arithmétique indiqué dans un avertissement.

Les tableaux à deux dimensions, sont des tableaux à une dimension dont les éléments sont eux-même des tableaux à une dimension :

```
In [93]: A = np.array([[0., 1, 3],[-1, -5, 2]] )

In [94]: A
Out[94]:
array([[ 0.,  1.,  3.],
       [-1., -5.,  2.]])
```

Le tableau est donc vu ici comme une liste de lignes. L'indexation se fait par :

```
In [96]: A[0, 2] = 4

In [97]: A
Out[97]:
array([[ 0.,  1.,  4.],
       [-1., -5.,  2.]])

In [98]: A[0, 4]
---------------------------------------------------------------------------
IndexError                                Traceback (most recent call last)
<ipython-input-98-e3f4f7241903> in <module>()
----> 1 A[0, 4]

IndexError: index 4 is out of bounds for axis 1 with size 3
```

Sur la dernière commande, on note qu'une tentative d'accéder à un élément hors des dimensions du tableau provoque une erreur. La dimension du tableau est son attribut **shape** :

```
In [112]: A.shape # notation objet
Out[112]: (2, 3)
```

Notons que le résultat est un tuple. Le nombre d'éléments du tableau est son attribut **np.size** et son type est **np.dtype**.

Attention : l'affectation d'un tableau ne crée pas de copie, mais seulement une nouvelle référence vers le même objet.

```
In [121]: A = np.array([[0., 1, 3],[-1, -5, 2]] )

In [122]: B = A # pas de copie

In [123]: B[0,0] = 100 # on modifie B
```

```
In [124]: A # A a été modifié
Out[124]:
array([[ 100.,  1.,  3.],
       [  -1.,   -5.,    2.]])
```

La copie d'un tableau se fait par la commande :

```
In [128]: B = A.copy()

In [129]: B[0, 0] = 0 # On modifie B

In [130]: A # A n'est pas modifié
Out[130]:
array([[ 100.,  1.,  3.],
       [  -1.,   -5.,    2.]])
```

La dimension d'un tableau **numpy** peut être supérieure à 2. Les éléments peuvent être réarrangés avec la méthode **reshape** :

```
In [160]: A = np.array([0, 1, 2, 3, 4, 5, 6, 7])

In [161]: A.shape # une dimension
Out[161]: (8,)

In [162]: A.reshape(2,4) # deux dimensions
Out[162]:
array([[0, 1, 2, 3],
       [4, 5, 6, 7]])

In [163]: B = A.reshape(2,2,2) # trois dimensions, pas de copie

In [164]: B
Out[164]:
array([[[0, 1],
        [2, 3]],

       [[4, 5],
        [6, 7]]])

In [165]: B[0,0,0] = 100 # On modifie B

In [166]: A # A est modifié
Out[166]: array([100, 1, 2, 3, 4, 5, 6, 7])
```

Sur les deux dernières commandes, on voit que **A** et **B** désignent toujours les mêmes données, mais la dimension n'est pas la même. **Python** dit que **B** est une **view** de **A**.

3.5 Le slicing

On peut accéder à des sous-tableaux d'un tableau **numpy** en utilisant des indices dont la forme générale est **debut:fin:pas**. Cette syntaxe désigne les indices allant de **debut** (inclus) à **fin** (exclu) avec un incrément de **pas**.

```
In [179]: A = np.array([0, 1, 2, 3, 4, 5, 6, 7])
```

```
In [180]: A[0:4:1] # indices 0 à 3
Out[180]: array([0, 1, 2, 3])

In [181]: A[0:8:1] # indices 0 à 7
Out[181]: array([0, 1, 2, 3, 4, 5, 6, 7])

In [182]: A[0:8:2] # indices pairs de 0 à 7
Out[182]: array([0, 2, 4, 6])

In [183]: A[1:8:2] # indices impairs de 0 à 7
Out[183]: array([1, 3, 5, 7])
```

S'il est omis, le `pas` par défaut est 1. Les indices `debut` et/ou de `fin` sont aussi optionnels et prennent les valeurs par défaut de premier et dernier indice respectivement :

```
In [184]: A[3:6] # indices de 3 à 5
Out[184]: array([3, 4, 5])

In [185]: A[:6] # indices de 0 à 5
Out[185]: array([0, 1, 2, 3, 4, 5])

In [186]: A[3:] # indices de 3 à 7
Out[186]: array([3, 4, 5, 6, 7])
```

L'incrément peut être négatif :

```
In [197]: A[::-1] # tous éléments en sens inverse
Out[197]: array([7, 6, 5, 4, 3, 2, 1, 0])

In [198]: A[5::-2] # indices impairs de 5 à 0
Out[198]: array([5, 3, 1])
```

La sélection de tout les éléments se fait par :

```
In [11]: A[:]
Out[11]: array([0, 1, 2, 3, 4, 5, 6, 7])
```

En effet, `debut` étant omis, il vaut 0, de même que `fin` vaut 7 par défaut. L'incrément n'est pas spécifié car il n'y a qu'un seul caractère `:` et vaut donc 1 par défaut.

3.6 La boucle `for`

Pour bénéficier de toute la puissance de calcul de `numpy`, il est déconseillé de recourir systématiquement au parcours des tableaux par des boucles. Il est très souvent possible de l'éviter en ayant recours aux méthodes de haut niveau de `numpy` pour les opérations de tri, de recherche, de calcul etc … . Cependant, `python` possède toutes les structures de contrôle usuelles et notamment la boucle `for`, qui permet d'itérer sur de nombreux types d'objets dits **itérables**. Les tableaux `numpy` sont itérables et peuvent donc être utilisés pour réaliser les boucles les plus simples :

La manière la plus simple de construire une boucle est d'itérer sur les éléments d'un tableau défini selon le modèle précédent. Dans l'exemple qui suit, nous calculons les carrés des 5 premiers entiers :

```
In [205]: I = np.arange(5) # tableau d'entiers de 0 à 4

In [206]: for i in I:
     ...:     print(i*i)
     ...:
0
1
4
9
16
```

Nous observons l'utilisation de la fonction `arange` (consulter l'aide) et de la fonction `print`. L'une des caractéristique principale de `python` est l'absence de délimitation explicite des blocs logiques : le début et la fin du bloc à exécuter dans le corps de la boucle est délimité uniquement par **l'indentation** (au moins un espace mais préférer 4). Il existe des utilisations bien plus avancées de la boucle `for` que nous n'utiliserons probablement pas.

3.7 Graphiques 2D

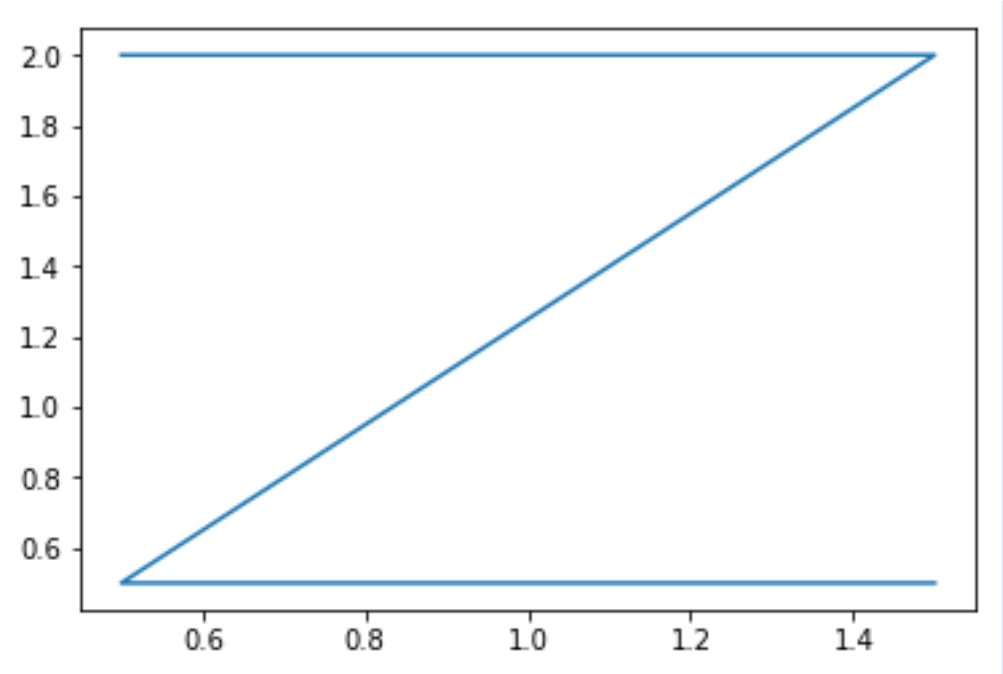
Le package `matplotlib` fournit un ensemble de modules `python` pour le traitement des objets graphiques, en lien avec `numpy`. Parmi ces modules, `matplotlib.pyplot` définit la fonction `plot` qui trace dans une ligne polygonale dans une fenêtre graphique. Dans sa forme la plus simple, cette fonction prend en paramètres deux tableaux `x` et `y` contenant les coordonnées des points définissant la ligne :

```
In [216]: import matplotlib.pyplot as plt

In [217]: x = [0.5, 1.5, 0.5, 1.5] # abcisses des 4 points

In [218]: y = [2, 2, 0.5, 0.5] # ordonnées

In [219]: plt.plot(x,y) # tracé de la ligne
Out[219]: [<matplotlib.lines.Line2D at 0x7f99aea1e5d0>]
<matplotlib.figure.Figure at 0x7f99aea9fc50>
```



On remarque que la figure a été tracée dans la console `spyder`. Ce comportement, utile en phase de développement, n'est pas toujours souhaitable en production puisque le code est destiné à être utilisé en dehors de l'IDE. On suggère donc de modifier cela dans `Outils/Préférences/Console Ipython`, onglet `Graphiques`, cadre `Sortie graphique`, choix `Automatique`. Ensuite `Fichier/Redémarrer` pour prendre en compte les changements. Les graphiques seront maintenant tracés dans des fenêtre à part, avec des outils permettant de les éditer et de les sauvegarder. Notons également qu'en dehors de l'IDE, il est nécessaire d'utiliser la commande `plt.show()` pour afficher effectivement la figure.

3.8 Quelques commandes utiles dans `spyder`

`clear` efface la console

`reset` détruit toutes les variables définies par l'utilisateur. Les variables peuvent aussi être détruites indépendamment par un clic droit dans l'explorateur de variables.

`cd`, `pwd`, `mkdir` etc … ont les mêmes effets que leurs homologues Unix. Elles ont pour effets respectifs de changer le répertoire de travail, de l'afficher et de créer un nouveau répertoire. Le résultat est renvoyé dans une chaîne de caractères.

`F9` exécute la sélection

`F5` exécute le fichier

`Ctrl+I` affiche l'aide pour la fonction où se trouve le curseur

4 Opérations et fonctions avancées sur les tableaux

4.1 Quelques tableaux particuliers

numpy fournit plusieurs fonctions pour générer des tableaux sans avoir à les entrer élément par élément.

- Tableau nul à **m** lignes et **n** colonnes

```
In [223]: np.zeros((3,2)) # matrice nulle 3 lignes x 2 colonnes
Out[223]:
array([[ 0.,  0.],
       [ 0.,  0.],
       [ 0.,  0.]])

In [224]: np.zeros((3,2), dtype = np.uint8) # la même, en entiers non-signés 8 bits
Out[224]:
array([[0, 0],
       [0, 0],
       [0, 0]], dtype=uint8)
```

Le paramètre obligatoire est un tuple donnant la **shape** du tableau à créer. Voir aussi la fonction **np.zeros_like**. Un paramètre optionnel **dtype** permet de spécifier le type des éléments.

- Tableau rempli de 1 à **m** lignes et **n** colonnes

```
In [225]: np.ones((1,6))
Out[225]: array([[ 1.,  1.,  1.,  1.,  1.,  1.]])

In [226]: np.ones((2,3), dtype = np.uint8)
Out[226]:
array([[1, 1, 1],
       [1, 1, 1]], dtype=uint8)
```

- Tableau à **m** lignes et **n** colonnes ayant des 1 sur la diagonale principale et des 0 ailleurs. On l'utilise souvent pour générer la matrice identité (carrée) d'ordre **m**

```
In [231]: np.eye(3) # matrice identité 3x3
Out[231]:
array([[ 1.,  0.,  0.],
       [ 0.,  1.,  0.],
       [ 0.,  0.,  1.]])

In [232]: np.eye(3,2) # attention, la taille n'est pas un tuple
Out[232]:
array([[ 1.,  0.],
       [ 0.,  1.],
       [ 0.,  0.]])
```

- Tableau de **num** nombres régulièrement espacés sur l'intervalle **[deb ; fin]**

```
In [236]: np.linspace(0,1,11) # découpage de [0;1] en dix intervalles
Out[236]: array([ 0. ,  0.1,  0.2,  0.3,  0.4,  0.5,  0.6,  0.7,  0.8,  0.9,  1. ])
```

```
In [237]: np.linspace(1,0,11) # même chose en sens inverse
Out[237]: array([ 1. ,  0.9,  0.8,  0.7,  0.6,  0.5,  0.4,  0.3,  0.2,  0.1,  0. ])
```

En particulier, on est sûr que les extrémités sont bien **deb** et **fin**, au contraire de la fonction **arange** déjà rencontrée.

- Tableau d'entiers aléatoires. Utile pour créer des exemples :

```
In [248]: np.random.randint(10, size = (3,2)) # matrice 3x2 d'entiers entre 0 et 9
Out[248]:
array([[2, 6],
       [0, 0],
       [9, 4]])

In [249]: np.random.randint(-5, 6, size = (2,3)) # matrice 2x3 d'entiers entre -5 et 5
Out[249]:
array([[-3, -2,  5],
       [-1, -1, -5]])

In [250]: np.random.randint(0, 256, size = (2,3), dtype = np.uint8) # matrice 2x3
d'entier uint8
Out[250]:
array([[163, 243, 146],
       [226,  61, 128]], dtype = uint8)
```

4.2 Opérations sur les tableaux

Les opérateurs scalaires usuels comme **+**, **-**, ***, / et **** s'appliquent élément par élément aux tableaux de mêmes dimensions. Si les tableaux sont de dimensions différentes, **numpy** essaiera de compléter le plus petit tableau par un mécanisme de règles appelé **broadcasting**.

```
In [309]: A = np.arange(6)

In [310]: A
Out[310]: array([0, 1, 2, 3, 4, 5])

In [311]: mat = A.reshape((3,2)) # matrice 3x2

In [312]: mat
Out[312]:
array([[0, 1],
       [2, 3],
       [4, 5]])

In [313]: row = np.array([1, 0]) # vu comme une ligne

In [314]: row
Out[314]: array([1, 0])

In [315]: a = 2 * np.ones((3, 1) # vu comme une colonne 3x1

In [316]: mat + a # a est dupliqué en 3x2
Out[316]:
array([[2, 3],
       [4, 5],
```

```
       [6, 7]])

In [317]: mat + row # row est dupliqué en 3x2
Out[317]:
array([[1, 1],
       [3, 3],
       [5, 5]])
```

Les choses sont plus simples pour les tableaux de mêmes dimensions :

```
In [321]: R = np.random.randint(10,size=(3,2))

In [322]: R
Out[322]:
array([[4, 5],
       [8, 2],
       [3, 8]])

In [323]: R ** 2 # carré des éléments de R
Out[323]:
array([[16, 25],
       [64,  4],
       [ 9, 64]])

In [324]: Q = 3 * np.eye(3, 2)

In [325]: Q
Out[325]:
array([[ 3.,  0.],
       [ 0.,  3.],
       [ 0.,  0.]])

In [326]: R * Q
Out[326]:
array([[ 12.,   0.],
       [  0.,   6.],
       [  0.,   0.]])
```

4.3 Sélections de lignes et colonnes

Nous avons déjà utilisé la technique du **slicing** pour les tableaux à une dimension. Elle s'étend naturellement aux tableaux à deux dimensions pour extraire des lignes et des colonnes :

```
In [5]: A = np.eye(5,5)

In [6]: A
Out[6]:
array([[ 1.,  0.,  0.,  0.,  0.],
       [ 0.,  1.,  0.,  0.,  0.],
       [ 0.,  0.,  1.,  0.,  0.],
       [ 0.,  0.,  0.,  1.,  0.],
       [ 0.,  0.,  0.,  0.,  1.]])

In [7]: A[0,:] # sélectionne la première ligne
Out[7]: array([ 1.,  0.,  0.,  0.,  0.])
```

```
In [8]: A[:,1] # sélectionne la deuxième colonne
Out[8]: array([ 0.,  1.,  0.,  0.,  0.])

In [12]: A[2:4,3:] # sélectionne les deux dernière colonnes des lignes 3 et 4
Out[12]:
array([[ 0.,  0.],
       [ 1.,  0.]])
```

Les indices n'étant pas forcément croissants, on peut lire une matrice de droite à gauche par :

```
In [21]: A[:,::-1]
Out[21]:
array([[ 0.,  0.,  0.,  0.,  1.],
       [ 0.,  0.,  0.,  1.,  0.],
       [ 0.,  0.,  1.,  0.,  0.],
       [ 0.,  1.,  0.,  0.,  0.],
       [ 1.,  0.,  0.,  0.,  0.]])
```

4.4 Indexation par des tableaux

Le slicing permet déjà de spécifier des indices qui ne sont pas des entiers, mais des tableaux d'entiers : la syntaxe 1:10:3 est en fait le tableau [1, 4, 7]. D'une manière générale, l'indexation par des tableaux quelconques d'entiers est valable, dès lors qu'ils sont unidimensionnels de même taille :

```
In [75]: A = np.arange(20).reshape(4,5)

In [76]: A
Out[76]:
array([[ 0,  1,  2,  3,  4],
       [ 5,  6,  7,  8,  9],
       [10, 11, 12, 13, 14],
       [15, 16, 17, 18, 19]])

In [77]: I = np.array([3, 0]) # deux indices de lignes

In [78]: J = np.array([4, 2]) # deux indices de colonnes

In [79]: A[I, J] # renvoie [A[3, 4], A[0, 2]]
Out[79]: array([19,  2])
```

En combinant cette possibilité avec le slicing, on peut par exemple répliquer une ligne :

```
In [90]: I = 2 * np.ones(5, dtype = np.int64) # noter le type entier pour indexer

In [91]: I # on va prendre 5 fois la ligne 2
Out[91]: array([2, 2, 2, 2, 2])

In [92]: A[I,:]
Out[92]:
array([[10, 11, 12, 13, 14],
       [10, 11, 12, 13, 14],
       [10, 11, 12, 13, 14],
```

```
       [10, 11, 12, 13, 14],
       [10, 11, 12, 13, 14]])
```

4.5 Tableaux de booléens

De même que pour les opérations numériques, les opération de comparaisons appliquées à des tableaux renvoient un tableau de booléens où la comparaison est appliquée à chaque élément.

```
In [94]: A = np.arange(20).reshape(4,5)

In [95]: A < 10 #
Out[95]:
array([[ True,  True,  True,  True,  True],
       [ True,  True,  True,  True,  True],
       [False, False, False, False, False],
       [False, False, False, False, False]], dtype=bool)
```

Les indices des éléments pour laquelle la condition est vraie son donnés par la fonction `where` de `numpy` :

```
In [99]: np.where(A<10)
Out[99]: (array([0, 0, 0, 0, 0, 1, 1, 1, 1, 1]), array([0, 1, 2, 3, 4, 0, 1, 2, 3,
4]))
```

En récupérant ces coordonnées dans des tableaux d'indices, on peut accéder aux éléments en question :

```
In [102]: I, J = np.where(A<10) # On dépaquette le tuple de tableaux renvoyé

In [103]: A[I, J] # indexation par des tableaux de unidimensionnels de taille 10
Out[103]: array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

Cependant, `numpy` propose une façon beaucoup plus intuitive de réaliser cela, car il permet d'indexer un tableau par un tableau de booléens dès lors qu'il possède la même `shape` :

```
In [104]: A[A<10]
Out[104]: array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

Noter que dans les deux cas, on obtient un tableau unidimensionnel en sortie.

4.6 Concaténation de tableaux

À l'inverse de l'extraction, on peut empiler les lignes, colonnes et blocs de matrices, à l'aide la fonction `block` :

```
In [68]: R = np.arange(6).reshape((3,2))

In [69]: R
Out[69]:
array([[0, 1],
       [2, 3],
       [4, 5]])
```

```
In [70]: row = 100 * np.ones((1,2)) # même nb de colonnes que R

In [71]: row
Out[71]: array([[ 100.,  100.]])

In [72]: Q = np.block([[R], [row]]) # ajoute une ligne en bas

In [73]: Q
Out[73]:
array([[   0.,    1.],
       [   2.,    3.],
       [   4.,    5.],
       [ 100.,  100.]])

In [74]: col = np.array([-10, -10, -10, -10]).reshape(4,1) # même nb de lignes que Q

In [75]: col
Out[75]:
array([[-10],
       [-10],
       [-10],
       [-10]])

In [76]: np.block([Q, col]) # ajoute une colonne à droite
Out[76]:
array([[   0.,    1.,  -10.],
       [   2.,    3.,  -10.],
       [   4.,    5.,  -10.],
       [ 100.,  100.,  -10.]])
```

On peut aussi voir les fonctions **concatenate**, **hstack** et **vstack** de **numpy**, les deux dernières étant destinées à disparaître.

4.7 Algèbre linéaire

Numpy possède un grand nombre de fonctionnalités de calcul d'algèbre linéaire où les tableaux bi-dimensionnels sont traités comme des matrices. Nous n'utiliserons pratiquement pas ces fonctionnalités, mais signalons qu'il existe le produit matriciel **dot** et la transposition des matrices (échange les lignes et le colonnes) **transpose** :

```
In [29]: R = np.random.randint(10,size=(3,2))

In [30]: R
Out[30]:
array([[6, 0],
       [9, 8],
       [0, 2]])

In [31]: np.dot(R, R) # R pas carré : produit matriciel impossible
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-31-86cfc044268b> in <module>()
----> 1 np.dot(R, R) # R pas carré : produit matriciel impossible

ValueError: shapes (3,2) and (3,2) not aligned: 2 (dim 1) != 3 (dim 0)
```

```
In [32]: Q = R.transpose() # dimension 2 x 3

In [33]: Q
Out[33]:
array([[6, 9, 0],
       [0, 8, 2]])

In [34]: np.dot(Q, R) # dimensions compatibles : (2x3) * (3x2) => 2x2
Out[34]:
array([[117,  72],
       [ 72,  68]])
```

5 Fonctions

La déclaration la plus simple d'une fonction qui prend des paramètres en entrée et retourne une valeur en sortie est par exemple :

```
In [87]: def prod(a, b):
    ...:     return(a*b)
    ...:

In [88]: prod(3,2)
Out[88]: 6
```

qui renvoie bêtement le produit de ses deux arguments d'appel. Le corps de la fonction est encore une fois écrit au sein d'un bloc logique défini uniquement par son **indentation**.

On peut donner des valeurs par défaut aux paramètres :

```
In [89]: def prod(a, b=2):
    ...:     return(a*b)
    ...:

In [90]: prod(4,5)
Out[90]: 20

In [91]: prod(4)
Out[91]: 8
```

Ici, on fait le produit par 2 lorsque le deuxième argument est omis.

La valeur de retour peut être un tuple si on veut affecter le résultat à plusieurs variables simultanément :

```
In [92]: def somprod(a, b):
    ...:     s = a + b
    ...:     p = a * b
    ...:     return((s, p))
    ...:

In [93]: somme, produit = somprod(3, 5)

In [94]: somme
Out[94]: 8
```

```
In [95]: produit
Out[95]: 15
```

Noter que le tuple a été « dépaqueté » lors de l'affectation de `somme` et `produit`.

Attention : `python` a un traitement très particulier pour le passage des paramètres que nous n'expliciterons pas ici. Signalons simplement que les types dits **non-mutables** comme les `int`, `float`, ou tuples ne seront jamais modifiés dans la fonction. Il est donc recommandé de ne les utiliser qu'en lecture seule :

```
In [102]: def double(x):
     ...:     x = 2. * x
     ...:

In [103]: x = 3

In [104]: double(x)

In [105]: x # x n'est pas doublé
Out[105]: 3
```

6 Images PNG

Dans ce projet, nous travaillerons exclusivement sur des fichiers d'images **matricielles** au format PNG sur 24 bits (« couleurs vraies »). Dans ce format, chaque pixel (zone homogène du support d'affichage), est représenté par une triplet (Rouge, Vert, Bleu) d'entiers non-signés sur 8 bits, donc des valeurs comprises entre 0 et 255. On rappelle que ce codage de couleur RVB s'appuie sur la **synthèse additive** des couleurs, pour représenter le noir par le triplet (0, 0, 0) et le blanc par (255, 255, 255). Puisque nous nous proposons de faire des calculs avec ces images, il faut disposer d'outils pour importer des fichiers images dans des tableaux `numpy`, et vice versa. Le module `matplotlib.image` est l'un des outils qui permettent de réaliser ces opérations.

6.1 Import d'un fichier PNG

La fonction `imread` du module `matplotlib.image` lit un fichier PNG et renvoie un tableau `numpy`.

```
In [1]: ls # nous avons un fichier png dans le répertoire courant
baboon.png

In [2]: import matplotlib.image as mpg

In [3]: baboon = mpg.imread('baboon.png', 'PNG') # import dans un tableau numpy

In [4]: baboon.shape
Out[4]: (100, 100, 3)
```

On constate qu'il s'agit d'une image carrée de 100x100 pixels. Le tableau `numpy` renvoyé est de dimension 3. On peut le voir comme la superposition de trois matrices 100x100, donnant la valeur de rouge, vert et bleu respectivement.

6.2 Visualisation avec matplotlib

La fonction **imshow** de **matplotib.pyplot** affiche un tableau **numpy** de dimension MxNx3 comme une image RVB.

```
In [7]: import matplotlib.pyplot as plt

In [8]: plt.imshow(baboon)
Out[8]: <matplotlib.image.AxesImage at 0x7f59cdb697d0>

In [9]: plt.show()
```

L'image affichée n'est pas à l'échelle, au sens où chaque valeur du tableau occupe une zone plus large qu'un pixel sur l'écran. Ce n'est pas forcément gênant pour le travail que nous aurons à faire. De plus, en zoomant sur l'image, on peut examiner le détail de chaque pixel, ce qui est parfois utile.

6.3 Manipulation avec numpy

Dans l'explorateur de variables, le tableaux tri-dimensionnel d'entiers **uint8** est apparu. On peut lire ses valeurs en double-cliquant sur sa ligne. Dans la fenêtre de lecture affichée, on peut lire le contenu des trois matrices (R, V, B) en choisissant **Axe : 2** puis **indice : 0, 1** ou **2** selon que l'on veut lire le plan R, V ou B. On peut vérifier toutes ses informations depuis la ligne de commande :

```
In [12]: type(baboon[0,0,0]) # il s'agit bien d'entier non-signés 8bits
Out[12]: numpy.uint8

In [13]: nw = baboon[0, 0,:] # valeurs RVB du pixel en haut à gauche

In [14]: nw
Out[14]: array([100,  88,  41], dtype=uint8)

In [15]: Rouge = baboon[:,:,0] # on extrait les valeurs rouge de tous les pixels

In [16]: Rouge.shape # on a une matrice 100x100
Out[16]: (100, 100)

In [17]: Rouge[0, 0] # valeur de rouge du pixel en haut à gauche doit être 100
Out[17]: 100
```

6.4 Export d'un tableau numpy en PNG

La fonction **imsave** de **matplotlib.mpg** créée un fichier PNG à partir d'un tableau **numpy** de dimension MxNx3. Cet exemple construit une image de 200x100 pixels avec un carré noir en haut et un carré blanc en bas. Le code est maintenant écrit dans l'éditeur de texte et exécuté depuis le menu.

```
#!/usr/bin/env python2
# -*- coding: utf-8 -*-
"""
```

```
@author: m3202C
Exemple simple de construction d'une image
"""

import numpy as np
import matplotlib.pyplot as plt
import matplotlib.image as mpg
#
# On prépare les trois matrices à empiler
R = 255 * np.ones((200, 100), dtype = np.uint8)
V = 255 * np.ones((200, 100), dtype = np.uint8)
B = 255 * np.ones((200, 100), dtype = np.uint8)
#
# Pour l'instant, la superposition donnerait du blanc.
# On mets à 0 les 100 premières lignes de chacune des matrices
R[:100, :] = 0
V[:100, :] = 0
B[:100, :] = 0
#
# On empile les trois matrices
carres = np.stack((R, V, B), axis = 2)
#
# visualisation avec imshow
plt.imshow(carres)
plt.show() # inutile en interactif
#
# Sauvegarde avec imsave
mpg.imsave('carres.png', carres)
```

La fonction **stack** prend en entrée un tuple de tableaux **numpy** de mêmes dimensions et un argument optionnel **axis** pour donner l'axe selon lequel on empile les tableaux. Ici, les tableaux **R**, **V** et **B** sont bi-dimensionnels. Le tableau résultant **carres** sera donc tridimensionnel, mais on précise que les trois tableaux 200x100 seront empilés selon le troisième axe (**axis = 2**). Le comportement par défaut de **stack** est de créer le troisième axe comme axe 0, ce qui n'est pas le format attendu par **imshow**.