Table des matières

I. Algorithmes : rappels

I.1 Variable globale et affectation

L'affectation : on affecte de la droite vers la gauche.

Algorithm 1: Affectation

```
x=1
y=2
z=x
x=y
y=z
```

I.2 Le test booléen

Algorithm 2: Tests booléens

```
a==1 # test d'égalité
a<=1 # test d'inégalité large
a>1 # test d'inégalité large
x in L # test d'appartenance
```

I.3 Syntaxe Python d'une fonction : def et return

Pour définir une fonction en Python, il faut faire **attention à l'indentation** (retrait de ligne de 3 caractères espaces) à l'aide de shift+entrée, et penser à renvoyer un résultat via la commande **return** (qui ne provoque pas d'affichage, ne la confondez pas avec la commande **print**) :

```
def mafonction(x): # fonction en Python
    return x**2+x-1
```

I.4 Instruction conditionnelle avec test booléen

L'instruction conditionnelle :

si *condition*
alors *instructions A*

(éventuellement : **sinon** *instructions B*).

Attention en Python à l'indentation (retrait de 3 carreaux sous les if, etc...) et il ne faut pas oublier les : en fin de

ligne.

Algorithm 3: If

```
x=rand()
if x<0.5 :
    x=1-x  ;
```

Algorithm 4: If Else

```
x=rand()
if x<0.5 :
    x=1-x ;
else :
    x=x/2 ;
```

Algorithm 5: If, Elseif, Elsee

```
x=rand()
if x<0.3 :
    x=1-x ;
elif x<0.6 :
    x=x/2  ;
    ;                                                    # else if
else :
    x=x/3 ;                                              # else
```

I.5 Boucles « Pour » avec compteur : FOR

La boucle For :

Pour k **allant de** n_0 **à** n
faire *instructions A*
On repète le blocs des instructions A pour un compteur de boucle k incrémenté d'une unité de n_0 jusqu'à n. L'itérateur k est muet dans la boucle

N.B. : Attention avec range(n), on décrit les entiers entre 0 et $n-1$, mais l'entier n n'est pas atteint !

Algorithm 6: Boucle For

```
S=2
for j in range(1,10):
    S+=S*j
    # !!! j prend les valeurs de 1 à 10-1
```

commande range : AVERTISSEMENT !!!

La commande range(n) correspond aux entiers allant de 0 à $\boxed{n-1}$ ...
Elle possède la variante range(m,n) correspond aux entiers allant de m à $\boxed{n-1}$ ...

```
>>> list(range(5))
[0,1,2,3,4]
>>> list(range(3,5))
[3,4]
```

I.6 Boucles « Tant que » avec condition d'arrêt : WHILE

L'itération conditionnelle Tant Que :

tant que *condition*
  faire *instructions A*
On repète le bloc des instructions A tant que la condition est réalisée.

Algorithm 7: Boucle While

```
1 k=0 ; T=1
2 while k<10 :
3     k=2*k ;
4     T+=T+k
```

I.7 Boucle For ou While ?

La boucle **pour** peut s'obtenir comme un cas particulier d'une boucle **tant que**. Les deux schémas d'algorithmes ci-dessous réalisent les mêmes instructions.

Algorithme 1	**Algorithme 1 bis**
	$k = n_0$
Pour k allant de n_0 à n	**Tant que** $k \leq n$
instruction A	*instruction A*
	$k = k + 1$ # *incrémentation du compteur*
k vaut n	k vaut $n+1$
on a fait $n - n_0 + 1$ tours de boucle for	on a fait $n - n_0 + 1$ tours de boucle while

Différence notable : la valeur de k à la fin de l'algorithme 1 bis est $n+1$; par convention, à la fin de l'algorithme 1, la valeur de k est n.

I.8 Position des problèmes

Trois questions vont attirer notre attention.

1. Lorsqu'on utilise une boucle itérative conditionnelle (*tant que*), il faut s'assurer que l'algorithme ne va pas nous entrainer dans une suite d'instructions qui ne s'arrêtre pas : c'est le problème de la **terminaison**.
2. Il faut s'assurer que l'algorithme va faire exactement ce qu'on attend de lui : c'est le problème de la **correction**.
3. Il faut maitriser la quantité de calcul que va nous demander un algorithme, notamment en vue d'améliorer la solution au problème posé : c'est le problème de la **complexité** en temps.
4. Dans le cas de stockage de calculs intermédiaires en mémoire, il faut maitriser la quantité de données écrites en mémoire vive : c'est le problème de la **complexité** en espace.

Exercice 1

Donner des instructions Python pour calculer et afficher le premier entier naturel n tel que $\left|\frac{\ln(n+1)}{\ln(n+2)} - 1\right| < 10^{-3}$.

Algorithm 8: exo 1

```
1
2
3
4
5
```

Exercice 2

Algorithm 9: exo 2

```
1 S=0
2 for k in range(100):
3     for j in range(k):
4         S+=1
```

A l'aide d'une somme souble utilisant le symbole Σ, calculer la valeur de S à l'issue de ces instructions.

II. Listes

Les listes sont des *structures de données* qui permettent de regrouper de manière structurée des ensembles des données (numériques ou autres). Nous verrons quelles opérations (ou *méthodes*) sont disponibles sur cette structure et nous définirons des fonctions qui opèrent sur ces structures.

II.1 Déclarer une liste

Une liste est une séquence ordonnée d'éléments. Elle appartient à la classe (ou type) `list`.
Elle n'est pas nécessairement homogène ; ses éléments peuvent appartenir à des types différents : `int`, `float`, `str` ou même `list`.
La longueur d'une liste est obtenue à l'aide de la fonction `len`, son affichage à l'aide de la fonction `print`.

```
>>> maListe=[2,'abc',[1,'a'],7.32]
>>> print(maListe)
[2, 'abc', [1, 'a'],7.32]
>>> len(maListe)
4
```

La fonction `range` permet de générer des séquences d'entiers consécutifs.
Pour créer des listes, Python propose une syntaxe très simple utilisant la boucle `for` parcourant une structure séquentielle (liste, chaine de caractères...) :

[f(x) `for x in` sequence]

Pour créer et filtrer une liste, on peut rajouter une instruction conditionnelle.

[f(x) `for x in` sequence `if` *condition*]

```
>>> entier=range(1,10)
>>> L=list(entier)
>>> print(L)
[1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> [-k for k in L]
[-1, -2, -3, -4, -5, -6, -7, -8, -9]
>>> [k for k in L if k% 2 == 0 ]
[2, 4, 6, 8]
```

La liste vide se définie avec `[]` ; elle a pour longueur 0 (c'est d'ailleurs la seule).

Exercice 3
Donner des instructions Python pour construire une liste contenant les carrés des 100 premiers entiers naturels.

Algorithm 10: exo

```
1
2
```

II.2 Accès aux éléments d'une liste

Pour accéder aux éléments d'une liste on utilise la même syntaxe que pour les chaînes de caractères.

- `L[0]` est le premier élément de L ;
- `L[k]` est l'élément d'indice k de L ;
- `L[-1]` est le dernier élément L ;
- `L[k1:]` est la sous-liste des éléments de L à partir de l'indice k_1 ;
- `L[:k2]` est la sous-liste des éléments de L jusqu'à l'indice k_2-1 .
- `L[k1:k2]` est la sous-liste des éléments de L entre les indices k_1 et k_2-1 ;

```
>>> L=[k for k in range(5)]
[0,1,2,3,4]
>>> L[0]
0
>>> L[-1]
4
>>> L[ :2]
[0,1]
>>> L[1 :4]
[1,2,3]
```

`Attention` : la numérotation commence à 0 ; si une liste est de longueur N son dernier élément a un indice $N-1$. La tentative d'accès à une position hors de la liste retourne une erreur.

Remarque. La numérotation peut se faire en négatif ; ainsi `L[-1]` désigne le dernier élément de la liste, `L[-2]` l'avant-dernier... `L[-N]` le premier.

II.3 Range : AVERTISSEMENT

Attention :

La commande range(n) correspond aux entiers allant de 0 à $\boxed{n-1}$ ...
Elle possède la variante range(m,n) correspond aux entiers allant de m à $\boxed{n-1}$ ...

```
>>> list(range(5))
[0,1,2,3,4]
>>> list(range(3,5))
[3,4]
```

II.4 Opération sur les listes

L'opérateur + fonctionne comme sur les chaînes de caractères : il permet de créer une nouvelle liste, concaténée des deux listes opérandes.
L'opérateur * permet de recopier un certain nombre de fois une liste.
On peut ainsi créer une liste contenant N zéros avec la commande :

```
>>> L=[1,2,3]
>>> L+[4,5,6]
[1,2,3,4,5,6]
>>> L*3
[1,2,3,1,2,3,1,2,3]
>>> [0]*5
[0,0,0,0,0]
```

II.5 Objet mutable

Contrairement aux chaînes de caractères ou aux t-uples, les listes sont des objets séquentiels *mutables*[1] : c'est à dire que vous pouvez modifier certains attributs de la liste. Ainsi vous avez la possibilité de modifier un élément d'une liste.

```
>>> maListe=[2,'abc',[1,'a'],7.32]
>>> maListe[1]='XYZ' # Redéfinit l'élément d'indice 1
>>> print(maListe)
[2,'XYZ',[1,'a'],7.32]
```

Vous pouvez même modifier une plage de valeurs, insérer des éléments dans la liste.

```
>>> maListe=[0,1,2,3,4]
>>> maListe[1 :3]= ['a','b','c'] # Modifie une plage d'éléments
>>> print(maListe)
[0, 'a', 'b', 'c', 3, 4]
>>> maListe[4 :4]= ['d','e'] # Insertion en position 4
>>> print(maListe)
[0, 'a', 'b', 'c', 'd', 'e', 3, 4]
```

II.6 Méthodes sur les listes

Les listes sont des *objets* Python auxquels sont attachés des méthodes ; celles-ci sont très efficaces (notamment en terme d'espace mémoire) pour modifier ou effectuer une opération bien déterminée sur une liste existante.

La syntaxe générale suivante permet de modifier l'objet `liste` :

```
liste.methode(args)
```

1. Contrairement aux objets de type *int*, *float*, *str*, *tuple*.

Exemple. La méthode append permet d'ajouter un élément à la fin d'une liste.

```
>>> L=[1,2,3]
>>> L.append(4)
>>> L
[1,2,3,4]
```

Exercice 4

Voici une liste de méthodes. A quoi peuvent-elles servir ?

Méthode	Action
MaListe.append(x)	Ajoute x en fin de liste
MaListe.pop()	retire le dernier élément de la liste MaListe, si elle est non vide
x=MaListe.pop()	retire le dernier élément de la liste MaListe et l'affecte à x
MaListe.extend(L)	
MaListe.remove(x)	retire la première occurence de x
MaListe.index(x)	
MaListe.count(x)	
MaListe.reverse()	
MaListe.pop(n)	

Exercice 5

Expliquez les valeurs associées à chaque variable Python à la fin de l'execution.

Algorithm 11: exo

```
1 L=[i for i in range(1,6)]
2 x=L.pop()
3 L.pop()
4 L.append(x)
```

III. Représenter des données numériques

Avec la librairie numpy de calcul numérique en Python.

```
>>> import numpy as np
```

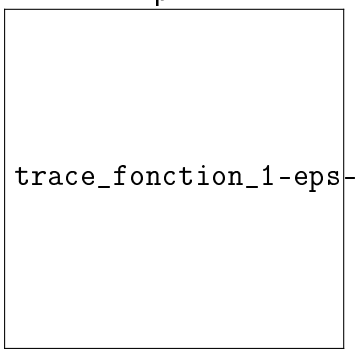
III.1 Représenter une fonction

On souhaite représenter le graphe d'une fonction f définie sur $[a,b]$ et à valeurs réelles, à l'aide de N points.

Par exemple, pour $a=-2$, $b=2$, $f : t \longmapsto 1-t^2$, $N=20$.

Pour cela, on va découper le segment $[a,b]$ en N points avec la commande `linspace` Puis on définit les valeurs des $f(t)$ pour les paramètres t correspondants.

Puis on passe au tracé




```
>>> t=np.linspace(-2,2,20) ;t
array([-2.  , -1.78947368, -1.57894737, -1.36842105, -1.15789474, -0.94736842,
-0.73684211, -0.52631579, -0.31578947, -0.10526316,  0.10526316,  0.31578947,
 0.52631579,  0.73684211,  0.94736842,  1.15789474,  1.36842105,  1.57894737,
 1.78947368,  2.  ])
>>> y=1-t**2 ;y
array([-3.  , -2.20221607, -1.49307479, -0.87257618, -0.34072022,  0.10249307,
 0.45706371,  0.72299169,  0.90027701,  0.98891967,  0.98891967,  0.90027701,
 0.72299169,  0.45706371,  0.10249307, -0.34072022, -0.87257618, -1.49307479,
-2.20221607, -3.  ])
>>> import matplotlib.pyplot as plt
>>> plt.plot (t,y) ; plt.xlabel('t') ;plt.ylabel('y=f(t)') ;plt.title('Tracé') ;plt.show()
```

Exercice 6

Donner des instructions Python pour représenter le graphe de la fonction $x \mapsto \sin x/x$ sur $]0, 2\pi]$, à l'aide de 100 points reliés par un trait continu bleu.

Algorithm 12: exo

```
1
2
3
4
```

III.2 Utiliser un script pour tracé le graphe d'une fonction

Les fonctions mathématiques usuelles sont présentes dans le package "numpy", la performante librairie graphique "matplotlib" est utile pour les tracés.

L'idéal, pour les utiliser conjointement est d'enregistrer un petit programme dans un fichier, puis de l'éxecuter avec la commande F5 dans l'interpréteur IDLE.

```
# chargement des librairies
import numpy as np
import matplotlib.pyplot as plt
# définition de la fonction f
def f(t) :
        return np.cos(2 * np.pi * t) * np.exp(-t)
# tracé du graphe avec matplotlib
x = np.linspace(0, 5.0,600)
y =f(x)
plt.plot(x, y, 'k')
plt.title('Oscillation amortie')
plt.text(2, 0.65, '$t \longmapsto \cos(2\pi t)\exp(-t)$')
plt.xlabel('Durée (s)')
plt.ylabel('Tension (mV)')
plt.show()
```

Exercice 7

Expliquez les valeurs associées à chaque variable Python à la fin de l'execution..

Algorithm 13: exo

```
1 L=[i for i in range(1,6)]
2 x=L.pop()
3 L.pop()
4 L.append(x)
```

IV. Ingénierie numérique : `odeint` et équations différentielles.

On charge la bibliothèque numérique de résolution approchée des équations différentielles :

```
>>> import numpy as np
>>> from scipy.integrate import odeint
```

IV.1 Equation différentielle d'ordre 1 (linéaire ou non)

La syntaxe de `odeint(f, y_0, t)` permet de résoudre les équations de la forme $y' = f(y,t)$ vérifiant la condition initiale $y(0) = y_0$, en appelant `odeint (f, y_0, t)` , avec y_0 valeur numérique à l'instant 0 de la fonction, et

$$\begin{array}{rccc} f : & \mathbb{R}\times\mathbb{R} & \longrightarrow & \mathbb{R} \\ & (y,t) & \longmapsto & f(y,t) \end{array}$$

Exemple de résolution : $y' = 2\ t\ y^2$

On va approcher numériquement sur $[0,2]$ la solution unique de $y' = 2t\ y^2$ avec la condition initiale $y(0) = 0.2$.

On recherche une fonction $\begin{array}{rccc} y : & [0,2] & \longrightarrow & \mathbb{R} \\ & t & \longmapsto & y(t) \end{array}$
telle qu'à chaque instant t on ait la relation :

$$y'(t) = 2t\ (y(t))^2$$

```
>>> t = np.linspace (0., 2., 10)
>>> def f (y, t) : return (2*t*y**2)
>>> odeint (f, .2, t)
array([[ 0.2       ],
       [ 0.20199509],
       [ 0.20822631],
       [ 0.2195123 ],
       [ 0.23753682],
       [ 0.26557403],
       [ 0.31034526],
       [ 0.38756059],
       [ 0.5436259 ],
       [ 1.00000647]])
```

On aurait aussi pu subdiviser le segment $[0,2]$ avec un pas de 0.1 en $[0, 0.1, 0.2, \ldots, 1.9]$ avec la commande `arange`

```
np.arange (0., 2., 0.1)
```

On pourrait prouver que cette équation s'intègre en $y : t \longmapsto \dfrac{1}{5-t^2}$, la valeur exacte en 2 est donc 1.

IV.2 La théorie

Pour une équation différentielle d'inconnue $Z : t \mapsto (z_1(t), \ldots, z_n(t))$, et une condition initiale $Z(t_0) = (z_{1,0}, \ldots, z_{n,0})$, et $f : \mathbb{R}^n \times \mathbb{R} \to \mathbb{R}^n$ continue, il existe une unique solution Z au problème de Cauchy :

$$\begin{cases} \forall t,\ Z'(t) = f(Z(t), t) \\ Z(t_0) = Z_0 \end{cases}$$

IV.3 Equation différentielle d'ordre 2

scipy sait uniquement intégrer une équation différentielle d'ordre 1. Si on cherche à intégrer l'équation différentielle d'ordre 2

$$y'' = g(y, y', t)$$

on la transforme en une équation différentielle d'ordre 1 en posant :
$Z(t) = \begin{pmatrix} y(t) \\ y'(t) \end{pmatrix}$, de sorte que $Z'(t) = \begin{pmatrix} y'(t) \\ y''(t) \end{pmatrix} = \begin{pmatrix} y'(t) \\ g(\ y(t), y'(t), t\ ) \end{pmatrix} = f(Z(t), t)$.

L'équation différentielle peut alors s'écrire vectoriellement $Z' = f(Z, t)$ en posant
$f : ((y, dy), t) \mapsto (dy, g(y, dy, t))$.

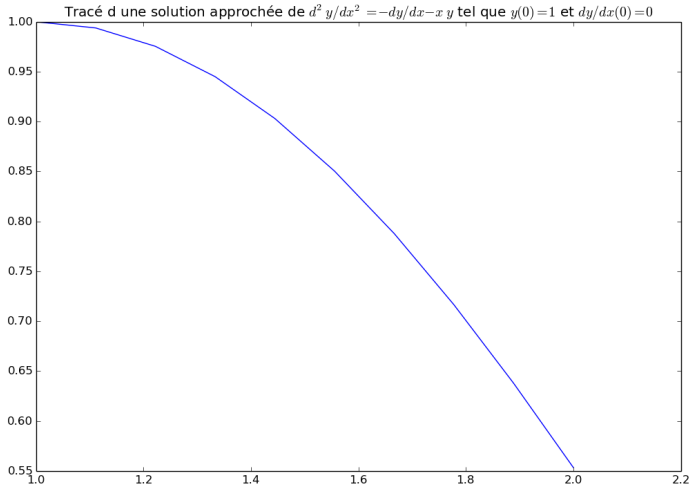
La condition initiale est ici de la forme $Z(t_0) = (y_0, y'_0)$

La subdivision $[t_0, t_1, ..., t_n]$ de l'intervalle sur lequel on intègre l'équation différentielle reste la même, par contre la condition initiale doit être un couple $(y(t_0), y'(t_0))$ et la valeur de retour est une matrice numpy dont la première colonne est constituée des valeurs $y(t_k)$ et la seconde des valeurs $y'(t_k)$.

Exemple de résolution : $y'' + y' + ty = 0$:

sur le segment $[1, 2]$ avec la condition initiale $(y(1) = 1, y'(1) = 0)$.

```
t =np.linspace (1., 2., 4)
def f(Z,t) :
    (y,dy)=Z
    return(dy,-dy-t*y)
y = odeint (f, (1., 0.), t) ;
import matplotlib.pyplot as plt
plt.plot (t, y[ :,0]) ;plt.show()
```



Attention, le résultat renvoyé ici par odeint est un tableau contenant les positions y[:,0] et les vitesses y[:,1]...

```
print (y)
array([[ 1.  ,  0.  ], [ 0.94502133, -0.32624253], [ 0.78778807, -
0.60463152], [ 0.55333316, -0.78078941]])
print (y[ :,0])
[ 1. 0.94502133 0.78778807 0.55333316]
```

Exercice 8

Donner des instructions Python pour représenter $t \mapsto (\theta(t), \dot{\theta}(t))$, pour θ solution du problème du pendule simple.

Algorithm 14: exo 1

```
1
2
3
4
5
6
7
```

IV.4 Programme PC maths : Système différentiel linéaire à coefficients quelconques

Théorème [théorème de Cauchy-Lipschitz linéaire (ADMIS, preuve HP)] Soient I un intervalle réel, $t_0 \in I$ (un instant initial), $n \in \mathbb{N}^*$, $X_0 \in \mathfrak{M}_{n,1}(\mathbb{K})$ (une position à l'instant initial), A, B deux fonctions de $\mathcal{F}(I, \mathfrak{M}_{n,1}(\mathbb{R}))$.

Supposons que A et B sont **continues sur** I, alors le problème de Cauchy

$$(\mathcal{PC}) \begin{cases} X'(t) &= A(t)\ X(t) + B(t) & (\mathcal{E}) \\ X(t_0) &= X_0 & (\mathcal{CI}) \end{cases}$$

admet une **unique solution** $X : I \to \mathfrak{M}_{n,1}(\mathbb{K})$, de classe $\mathcal{C}^1$ sur I.

V. Matrices numpy

Au préalable, on charge la bibliothèque de calcul numérique :

```
>>> import numpy as np
```

Toutes les commandes de cette librairie commenceront par np. : np.sin, np.pi, etc...

V.1 Le type array

L'opérateur `np.array` prend en argument une liste et renvoie un tableau numpy ayant les mêmes éléments. En cas de liste de listes, l'opérateur s'applique récursivement à chaque sous-liste : la valeur de retour est donc un tableau de tableaux.

Exemple :

```
>>> L =[[5,3,2],[7,4,1]]
>>> L
[[5, 3, 2], [7, 4, 1]]
>>> type(L)
<class 'list'>
>>> A=np.array(L)
>>> A
array([[5, 3, 2], [7, 4, 1]])
>>> type(A)
<class 'numpy.ndarray'>
>>> np.size(A)
6
>>> np.shape(A)
(2, 3)
```

V.2 Accès aux coefficients

Pour une matrice numpy A, il est possible d'accéder directement à l'élément de ligne i et de colonne j avec la syntaxe m[i, j] (alors qu'avec un tableau Python "classique" il faudrait écrire A[i][j]).

```
>>> A[0,0]
5
>>> A[ :,0]
array([5, 7])
>>> A[1, :]
array([7, 4, 1])
```

Attention ! ! En Python, les indices de ligne et de colonne commencent à 0.

V.3 Liste de listes

On peut par exemple créer la liste des listes de ces valeurs (en utilisant une syntaxe " en compréhension ") et convertir le résultat en un " array " :

```
>>> M = np.array([[10*i+j for j in range(10)] for i in range(5)]) ;M
array([ [ 0, 1, 2, 3, 4, 5, 6, 7, 8, 9],
[10, 11, 12, 13, 14, 15, 16, 17, 18, 19],
[20, 21, 22, 23, 24, 25, 26, 27, 28, 29],
[30, 31, 32, 33, 34, 35, 36, 37, 38, 39],
[40, 41, 42, 43, 44, 45, 46, 47, 48, 49] ])
```

V.4 Calcul matriciel et vectoriel avec numpy

4.a) Transposée

```
>>> B=np.array([[-1,0,1],[0,1,0]]);B
array([[-1, 0, 1], [ 0, 1, 0]])
>>> C=B.T ; C
array([[-1, 0], [ 0, 1], [ 1, 0]])
```

4.b) Produit matriciel

```
>>> np.dot(A,C)
array([[-3, 3], [-6, 4]])
>>> G=A.dot(C) ; G
array([[-3, 3], [-6, 4]])
>>> V=np.array([[1],[3],[5]])
>>> np.dot(A,V)
array([[24], [24]])
>>> np.linalg.matrix_power(G,2)
array([[-9, 3], [-6, -2]])
```

Attention : $G**2$ se contente d'élever chaque coefficient au carré au lieu de calculer G^2 au sens matriciel...

4.c) Matrices usuelles

```
>>> np.zeros([3, 4])
array([[ 0., 0., 0., 0.], [ 0., 0., 0., 0.], [ 0., 0., 0., 0.]])
>>> np.ones([2, 5])
array([[ 1., 1., 1., 1., 1.], [ 1., 1., 1., 1., 1.]])
>>> np.eye(4)
array([[ 1., 0., 0., 0.], [ 0., 1., 0., 0.], [ 0., 0., 1., 0.], [ 0.,
0., 0., 1.]])
>>> np.eye(4,dtype='int')
array([[1, 0, 0, 0], [0, 1, 0, 0], [0, 0, 1, 0], [0, 0, 0, 1]])
>>> np.random.randint(-5,5,size=(4,5)) array([[-2, -1, 3,
-3, 3], [ 0, 2, 3, 1, -1], [-2, -2, -1, 4, 0], [ 3, -2, 1, 3, 1]])
```

4.d) Algèbre linéaire

```
>>> P = np.array([[1, 3, 3], [1, 4, 3], [1, 3, 4]])
>>> np.linalg.det(P)
1.0
>>> np.linalg.inv(P)
array([[ 7., -3., -3.], [-1., 1., 0.], [-1., 0., 1.]])
>>> np.linalg.solve (P, [[1],[0],[0]])
array([[ 7.], [-1.], [-1.]])
>>> np.trace(P)
9
>>> np.linalg.eig(np.array([[2,1],[1,1]]))
(array([ 2.61803399, 0.38196601]), array([[ 0.85065081, -
0.52573111], [ 0.52573111, 0.85065081]]))
```

`np.linalg.eig()` permet d'obtenir des approximations numériques des valeurs propres et des vecteurs propres d'une matrice.

V.5 Problèmes de type des entrées

On peut convertir les coefficients entiers de la matrice A précédente au format `int`, `float` ou `complex` :

```
>>> A.astype(float)
array([[ 5., 3., 2.], [ 7., 4., 1.]])
>>> A.astype(complex)
array([[ 5.+0.j, 3.+0.j, 2.+0.j], [ 7.+0.j, 4.+0.j, 1.+0.j]])
>>> A.astype(int)
array([[5, 3, 2], [7, 4, 1]])
```

V.6 Divers

Calculs par blocs : on peut travailler par blocs avec la commande `np.bmat([[M,N],[P,Q]])`

Copies de matrices : si on définit une matrice A puis une matrice B en écrivant $B = A$, alors la matrice A n'est pas dupliquée en mémoire. Autrement dit, les variables A et B font référence à la même matrice et une modification de B portera également sur A (c'est comme avec les listes).

On peut utiliser la fonction `deepcopy` du module `copy`.

```
>>> from copy import deepcopy
>>> B=deepcopy(A)
```

VI. Copie de listes

Point cours : notion d'identifiant et conséquences.

En Python tout objet créé est rangé dans une « case mémoire » avec une adresse bien précise : sa référence. Lorsqu'on réalise une affectation, la variable que l'on manipule est une référence de l'objet.

```
>>> x,y=1,'coucou'
>>> id(x), id(y)
32111456,32898624
>>> z=y
>>> id(z)== id(y)
True
```

En modifiant un attribut d'une liste avec une méthode donnée, on modifie la liste en mémoire sans changer l'adresse de l'objet. Toutes les variables qui pointaient vers cette liste, retournent désormais la nouvelle version de la liste.

```
>>> L1=[1,2,3]
>>> L2=L1
>>> L2[0]=0
>>> print(L1)
[0,2,3]
```

La remarque précédente a des conséquences majeures sur la copie de liste. Pour copier une liste connue par une variable, une simple affectation peut s'avérer insuffisante si on souhaite modifier la liste et conserver l'ancienne version de la liste. En effet l'affectation copie seulement la référence vers la liste : il n'y a toujours qu'une seule liste dans la mémoire de l'ordinateur.

Exercice 9

Une méthode superficielle.

Dans un script Python `copieListe.py` on définit une fonction donnée par l'algorithme ci-contre (on utilise la méthode `append`).

Fonction : copie(L).
Entrée : une liste L
Sortie : une liste $Copie$
$Copie =$ liste vide
$n =$ longueur(L)
Pour $k = 0$ **à** $n - 1$:
    ajoute $L[k]$ à la fin de $Copie$
Retourner $Copie$

On considère les lignes de commandes suivantes. Compléter la réponse à la dernière instruction (résultats des `print` ?) :

```
>>> L=[3,2,1]
>>> M=copie(L),
>>> print(M)
>>> M[0]=5
>>> print(L,M)
?...
```

```
>>> L=[[3],[2],[1]]
>>> M=copie(L)
>>> print(M)
>>> M[0][0]=5
>>> print(L,M)
?...
```

L'exemple ci-dessus montre que la copie de liste à la main s'avère insuffisante si les objets de la liste sont aussi *mutables*. Pour obtenir une vraie copie, indépendante de la liste copiée, on peut utiliser la fonction deepcopy du module copy.

Exercice 10 *Copie profonde*

On considère les lignes de commandes ci-contre. Compléter la réponse à la dernière instruction.

```
>>> from copy import deepcopy
>>> L=[[3],[2],[1]]
>>> M=deepcopy(L)
>>> print(M)
>>> M[0][0]=5
>>> print(L,M)
?.....
```

Point cours : passage par référence.

En Python les paramètres formels d'une fonction ne reçoivent pas une copie des arguments mais reçoivent la référence des arguments. Lorsque l'argument est *mutable*, la fonction pourra donc modifier l'argument original

```
>>> L=[1,2,3]
>>> def test(param) :
        param[0]=0
>>> test(L)
>>> print(L)
[0,2,3]
```

Il faut être prudent si on souhaite rédiger des procédures dont les entrées sont de type *list*. Si on ne souhaite pas modifier les arguments il est préférable, dans le corps de la fonction, de travailler sur des copies (profonde) des arguments.