TD : La simulation numérique
Numpy et Matplotlib - Corrigé

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

Sous Python, l'import du module *numpy* permet de réaliser des opérations pratiques sur les tableaux.
Les indices de ces tableaux commencent à 0.

Exercice 1 :

Construire un objet de type *numpy.ndarray*

- dont les termes sont les multiples de 3 compris entre 0 et 27 ;

```
>>> a = np.arange(0,28,3); print(a)
```

- dont les termes sont les puissances de 2 de $2^0 = 1$ à $2^{12}=4096$;

```
>>> b = np.arange(0,13); print(2**b)
```

Exercice 2 :

Pour cet exercice, on prend n = 1 000 000.

1. Calculer $\sum_{i=0}^{n} i$ sans utiliser *numpy*. Chronométrer le temps nécessaire pour le calcul précédent, par exemple en utilisant **time.clock()**.

NB :

Le module ***time*** : Permet de mesurer le temps écoulé et le temps CPU (le temps passé par un processus en mémoire centrale) :

- **time()** mesure le temps ordinaire, celui de l'horloge.
- **clock()** donne le temps CPU consommé par le processus Python actuel depuis le début de son exécution.

Exemples :

```
import time
e0 = time.time() # temps écoulé en sécondes depuis l'époque (01-01-1970 00:00:00)
c0 = time.clock() # temps CPU total en sécondes passé dans l'exécution du script
temps_ecoule = time.time() - e0
cpu_time = time.clock() - c0
```

Solution :

```
import numpy as np
from time import clock

# 1ère méthode : avec une boucle
t0 = clock()
a = range(1000001)
t1 = clock()
S = 0
for i in a:
   S += i

t2 = clock()

print('1ère méthode : {:6.6f} + {:6.6f} = {:6.6f} secondes'.
       format(t1-t0,t2-t1,t2-t0))
```

```
# 2ème méthode : avec une liste en compréhension
t0 = clock()
a = range(1000001)
t1 = clock()
S = sum(i for i in a)
t2 = clock()
print('2ème méthode : {:6.6f} + {:6.6f} = {:6.6f} secondes'.
      format(t1-t0,t2-t1,(t1-t0)+(t2-t1)))
```

2. Utiliser un tableau *numpy* et la méthode **sum** pour calculer à nouveau la somme proposée.

```
t0 = clock()
v = np.array(range(1000001))
t1 = clock()
S = v.sum() # ou np.sum(v)
t2 = clock()
print('3ème méthode : {:6.6f} + {:6.6f} = {:6.6f} secondes'.
      format(t1-t0,t2-t1,(t1-t0)+(t2-t1)))
```

3. Comparer le temps de calcul avec la méthode précédente.

Le calcul de la somme lui-même est extrêmement rapide avec numpy.
La définition de l'itérateur range est extrêmement rapide (il n'est pas évalué lors de sa définition)
La définition d'un tableau numpy est un peu lent, car il faut compter avec les problèmes d'allocation mémoire. Mais une fois qu'il est défini, on peut faire des calculs très rapides.

Traçage des courbes : le module *matplotlib*

Exercice 3 :
Ecrire un script qui demande à l'utilisateur les coordonnées des trois points $A(x_A, y_A)$, $B(x_B, y_B)$ et $C(x_C, y_C)$, et trace le triangle ABC.

On rappelle que :
les points du triangle ABC sont A(a1;a2), B(b1;B2) et C(c1;c2)
et plt.plot([x1, x2, x3, x1],[y1, y2, y3, y1]) relie les points (x1;y1), (x2;y2), (x3;y3) et (x1;y1)
On fait donc:

```
import matplotlib.pyplot as plt
x1= float(input('x1 = '))
y1= float(input('y1 = '))
x2= float(input('x2 = '))
y2= float(input('y2 = '))
x3= float(input('x3 = '))
y3= float(input('y3 = '))
plt.plot([x1, x2, x3, x1],[y1, y2, y3, y1])
plt.show() # Afficher la fenetre graphique
```

Exercice 4 :

Réaliser le graphe de la fonction $f(x,y) = v_0 t - \frac{1}{2} g t^2$ pour $v_0 = 10$, $g = 9.81$, et $t \in [0, 2v_0/g]$.

Le label sur l'axe des x devra être "temps (s)" et le label sur l'axe des y "hauteur (m)".

```
from math import *
import numpy as np
import matplotlib.pyplot as plt
v0 = 10
g = 9.81
a=0
b= 2 *v0 / g
t = np.linspace(a,b,100)

def y(t):
   return v0 * t - 1/2 * g * t**2

plt.plot(t, y(t))
plt.xlabel("Temps (s)")
plt.ylabel("Hauteur (m)")
plt.show()
```

Exercice 5 :

La factorisation Cholesky, consiste, pour une matrice carrée **A** d'ordre **n** symétrique définie positive, à déterminer une matrice triangulaire inférieure **L** telle que **A=LL'** (avec L' = transposé de L). Les relations suivantes permettent de déterminer les éléments de **L** à partir de **A** :

- Initialement tous les éléments de **L** sont nuls
- $L_{00} = \sqrt{A_{00}}$
- $L_{j0} = \frac{A_{0j}}{L_{00}}$ avec j=1..n-1
- $L_{ii} = \sqrt{A_{ii} - \sum_{k=0}^{i-1} L_{ik}^2}$ avec i=1..n-1
- $L_{ji} = \left(A_{ij} - \sum_{k=0}^{i-1} L_{ik} L_{jk}\right) \Big/ L_{ii}$ avec i=1..n-1 et j=i+1..n-1

Le déterminant de **A** est égal alors au carré du produit des éléments de la diagonale principale de **L**.

1) Ecrire une fonction **DECOMP** qui calcule la matrice **L** à partir d'une matrice **A** d'ordre **n** en utilisant les formules précédentes.

```
def decomp(A):
   n= A.shape[0]
   #Créer une matrice nulle L
   L=np.zeros((n,n));

   #point 1
   L[0,0]=sqrt(A[0,0])
   #point 2
   for j in range(1,n):
       L[j,0] = A[0,j]/L[0,0]

   for i in range(1,n):
       s=0
       for k in range(i):
           s+ = L[i,k] ** 2
       L[i,i] = sqrt(A[i,i]-s)

       for j in range(i+1,n):
           s=0
           for k in range(i):
               s+=L[i,k]*L[j,k]
           L[j,i]=(A[i,j]-s)/L[i,i]

   return L
#tester la fonction
L = decomp(A)
LLT = np.dot(L,np.transpose(L)) #de type float
#print(np.array_equal(A,LLT)) #false int <> float
```

2) Ecrire une fonction **DETER** qui retourne le déterminant d'une matrice **A** d'ordre **n** sans utilisation de la commande **det**.

```
def deter (A):
   """
   Le déterminant d'une matrice triangulaire est égale au produit
   des termes de la diagonale
   """
   L = decomp(A) # la matrice triangulaire inférieure
   d=np.diag(np.dot(L,L)) # diagonale de la matrice triangulaire
   return np.prod(d) # déterminant de A

print(deter (A))
print(np.linalg.det(A))
```

L'inverse d'une matrice carrée **A** d'ordre **n** est donné par la formule suivante :
$\mathbf{A^{-1}=C'/det(A)}$

Notation:
C = matrice des cofacteurs dont les coefficients sont obtenus en utilisant la formule suivante :

$$\mathbf{C_{ij}=(-1)^{i+j}*det(A_{ij})}$$

Avec : $\mathbf{C_{ij}}$: L'élément de **C** à la ligne **i** et colonne **j**
$\mathbf{A_{ij}}$: La matrice **A** privée de la ligne d'indice **i** et de la colonne d'indice **j**
det : Le déterminant de la matrice
C' : transposé de **C**

3) Ecrire une fonction **MINOR** qui supprime la ligne d'indice **i** et la colonne d'indice **j** d'une matrice **A** d'ordre **n**.

```
def minor (A,i,j):
   return np.delete(np.delete(A,j,1),i,0)

#test
A1 = minor (A,0,0)
```

4) Ecrire une fonction **COFACT** qui calcule et retourne la matrice des cofacteurs d'une matrice **A** d'ordre **n**.

```
def cofact (A):
   n= A.shape[0] #taille de la matrice A
   C = copy.deepcopy(A)
   for i in range (n):
       for j in range (n):
           C[i,j] = (-1)**(i+j) * np.linalg.det(minor(A,i,j))
   return C
#test
C = cofact (A)
print(C)
```

Remarque :

On n'écrit pas `C[i,j] = (-1)**(i+j) * deter(minor(A,i,j))` car la fonction `deter()` attend en paramètre une matrice symétrique définie positive qui est différent du résultat de l'appel `minor(A,i,j)`

5) Ecrire une fonction **INVERSE** qui retourne l'inverse d'une matrice **A** d'ordre **n**.

```
def inverse (A):
   return np.transpose(cofact (A))/np.linalg.det(A)
```

```
#test
I = inverse (A) ; print(I)
```