TP 7 : Révisions

Exercice 1 *Suites récurrentes imbriquées.* On modélise l'évolution d'une certaine maladie en répartissant les individus suceptibles d'être infecté en trois groupes : le groupe S (individus sains et non immunisés, suceptibles d'être infectés), le groupe I (individus infectés) et le groupe R (individus guéris suite à l'infection et résistant à la maladie). On note s_n (respectivement i_n, r_n) la proportion d'individus dans le groupe S (respectivement I, R) à l'instant n. On suppose que l'évolution de l'épidémie est modélisée par les relations :

$$\begin{aligned} s_{n+1} &= s_n - 0.3 s_n i_n \\ i_{n+1} &= 0.95 i_n + 0.3 s_n i_n \\ r_{n+1} &= r_n + 0.05 i_n \end{aligned}$$

avec initialement $s_0 = 0.9$, $i_0 = 0.1$ et $r_0 = 0$.

(a) Écrire un programme qui demande d'entrer un entier n et affiche les valeurs s_n, i_n et r_n correspondantes.

(b) Écrire une fonction **`evolution(n)`** qui construit et renvoie les listes $S = [s_0, \dots, s_n]$, $I = [i_0, \dots, i_n]$ et $R = [r_0, \dots, r_n]$.

(c) Représenter graphiquement l'évolution des populations dans chacun des trois groupes. En particulier, on cherchera si les proportions d'individus dans chaque population se stabilisent au bout d'un temps assez long.

Exercice 2 *D'autres suites récurrentes imbriquées.* On considère les suites $(a_n)_{n\geqslant 0}$, $(b_n)_{n\geqslant 0}$ et $(c_n)_{n\geqslant 0}$ définies par $a_0 = b_0 = 1$, $c_0 = 0$ et :

$$\forall k \in \mathbb{N}, \begin{cases} a_{k+1} = -2a_k \\ b_{k+1} = b_k - c_k \\ c_{k+1} = b_k + c_k \end{cases}$$

Écrire une fonction **`suites(n)`** qui calcule et renvoie les valeurs a_n, b_n et c_n à partir de l'entier n donné en paramètre.

Exercice 3 *Recherche d'un élément dans une liste.*

(a) Écrire une fonction **`contient_positif(L)`** qui prend en argument une liste **`L`** et renvoie le booléen **`True`** si cette liste contient un nombre positif et **`False`** dans le cas contraire.

(b) Écrire une fonction **`appartient(x,L)`** qui renvoie le booléen **`True`** si l'élément **`x`** apparait dans **`L`** et **`False`** dans le cas contraire.

Exercice 4 *Algorithme de compression RLE.* La *compression de données* est une opération informatique consistant à transformer une suite de données A en une suite de données B, plus courte que A mais à partir de laquelle il est possible de retrouver les données A. On distingue essentiellement deux types de compressions :

- La compression *sans perte* : la suite de données A peut être restituée intégralement à partir de B ;
- La compression *avec pertes* : la suite B ne permet de retrouver qu'approximativement la suite A de départ.

Les algorithmes de compression avec pertes sont essentiellement utilisés pour le stockage du son, des images et de la vidéo (les formats de données JPEG, MP3, MPEG utilisent ce

mode de compression). On étudie ici un algorithme de compression sans perte appelé *codage par plages* (en anglais *run-length encoding*, acronyme RLE), cet algorithme est utilisé assez souvent pour des compresser des images en noir et blanc (fax) mais aussi en couleurs (formats d'images BPM et PCX). Le principe de l'algorithme est le suivant : on dispose d'une liste A contenant des nombres entiers que l'on souhaite compresser et on va transformer cette liste en une liste B en indiquant les éléments de A ainsi que le nombre de fois où ils apparaissent. Quelques exemples :

- Si $A = [1,1,2,2,2,1,1,3,3,3,4]$, alors $B = [2,1,3,2,2,1,3,3,1,4]$;
- Si $A = [1,2,1,2]$, alors $B = [1,1,1,2,1,1,1,2]$;
- Si $A = [3,3,3,3,3,3,3]$, alors $B = [7,3]$.

(a) Écrire une fonction **`decompresser_rle(B)`** qui construit et renvoie la liste A obtenue à partir de la liste compressée B.

(b) Écrire une fonction **`compresser_rle(A)`** qui construit et renvoie la liste B obtenue en compressant la liste A suivant l'algorithme RLE.

Ne pas oublier de tester ces fonctions sur des exemples pertinents.

Corrections

Ex 1. Pour demander d'entrer une valeur entière :

```
n = int(input('Entrer la valeur de n:'))
```

```
s = 0.9
i = 0.1
r = 0
for k in range(0,n):
    st = s
    it = i
    rt = r
    s = st-0.3*st*it
    i = 0.95*it+0.3*st*it
    r = rt+0.05*it
print(s,i,r)
```

Par exemple pour $n = 4$ on obtient :

```
0.760879992805578 0.2117391993405705 0.027380807853851403
```

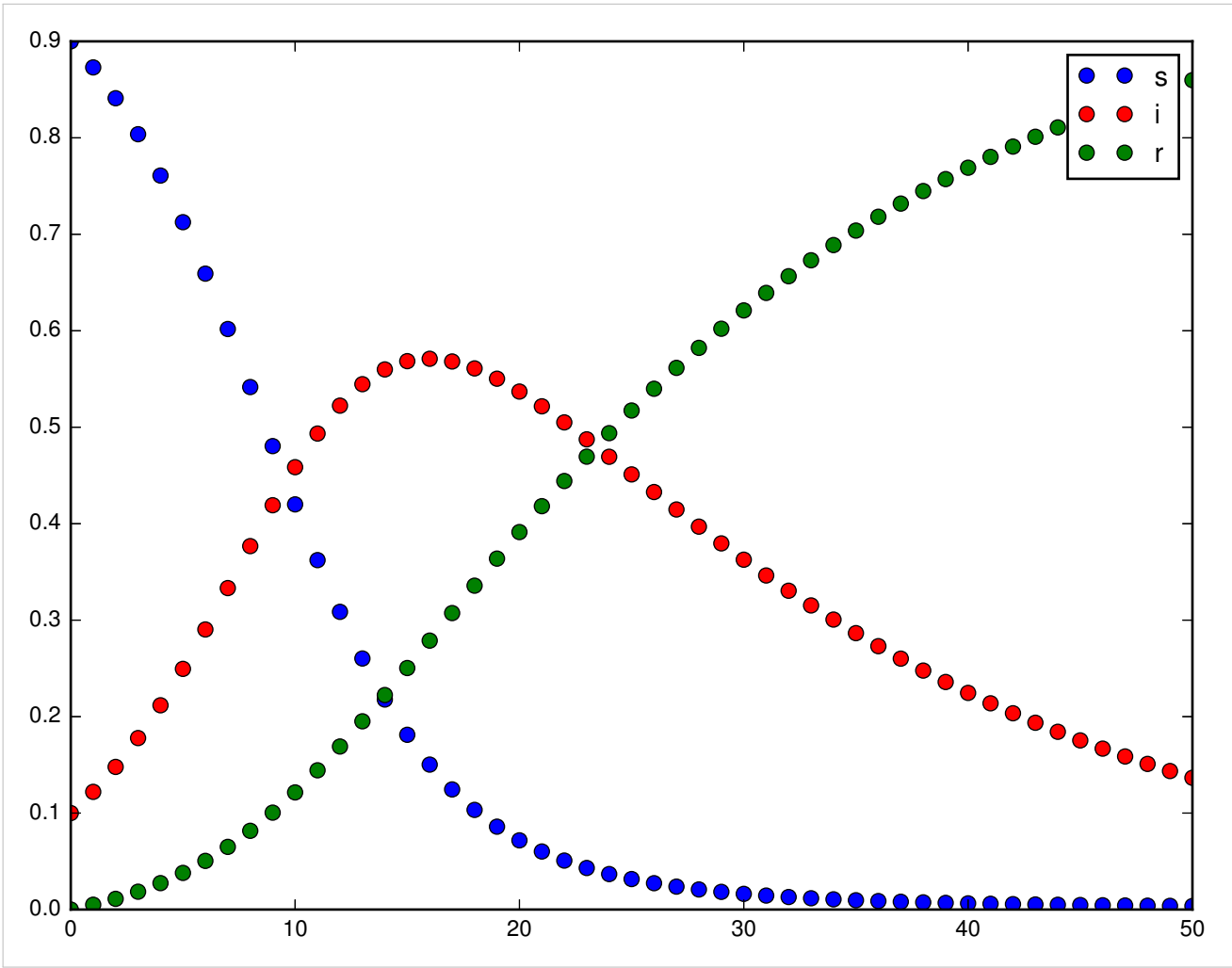
Écriture sous forme d'une fonction produisant les listes :

```
def evolution(n):
    s = 0.9
    i = 0.1
    r = 0
    Ls = [s]
    Li = [i]
    Lr = [r]
    for k in range(0,n):
        st = s
        it = i
        rt = r
        s = st-0.3*st*it
        i = 0.95*it+0.3*st*it
        r = rt+0.05*it
        Ls.append(s)
        Li.append(i)
        Lr.append(r)
    return (Ls,Li,Lr)
```

Représentation graphique :

```
import matplotlib.pyplot as plt

(Ls,Li,Lr) = evolution(50)
plt.plot(Ls,'bo')
plt.plot(Li,'ro')
plt.plot(Lr,'go')
plt.legend(['s','i','r'])
```



Ex 2.

```
def suites(n):
    a = 1
    b = 1
    c = 0
    for k in range(n):
        at = a
        bt = b
        ct = c
        a = -2*at
        b = bt-ct
        c = bt+ct
    return (a,b,c)
```

Ex 3. On va utiliser une variable **positif** de type booléan qui contient au début la valeur **False**. On parcourt ensuite toute la liste et lorsque l'on trouve un élément positif on donne à la variable **contient** la valeur **True**. Après la boucle, on renvoie cette variable qui vaut **True** si on a rencontré un élément positif durant la parcourt de la liste et **False** dans le cas contraire.

```
def contient_positif(L):
    contient = False
    for k in range(0,len(L)):
        if L[k]>=0:
            contient = True
    return contient

print(contient_positif([-2,5,3,-1]))
print(contient_positif([-2,-5,-3,-1]))
```

```
True
False
```

On peut utiliser l'autre manière de parcourir une liste :

```
def contient_positif(L):
    contient = False
    for x in L:
        if x>=0:
            contient = True
    return contient
```

et on peut aussi utiliser le fait qu'un **return** stoppe immédiatement l'exécution de la fonction :

```
def contient_positif(L):
    for x in L:
        if x>=0:
            return True
    return False
```

Sur le même principe :

```
def appartient(x,L):
    for y in L:
        if x==y:
            return True
    return False
```

Ex 4. Dans la décompression, la liste B qui est donnée doit être de la forme $B = [n_0, x_0, n_1, x_1, \ldots, n_{p-1}, x_{p-1}]$ avec $n_0, \ldots, n_{p-1}$ des nombres entiers, strictement positifs par définition de l'algorithme de compression. La liste B contient donc un nombre pair d'éléments, $2p$. La liste A est alors constituée de x_0 apparaissant n_0 fois suivi de x_1 apparaissant n_1 fois, etc. jusqu'à x_{p-1} apparaissant n_{p-1} fois. On peut alors écrire :

```
def decompresser_rle(B):
    p = len(B)//2 # len(B) doit être divisible par 2
    A = []
    for k in range(0,p):
        n = B[2*k]
        x = B[2*k+1]
        # Il faut ajouter x à la liste A n fois
        for j in range(0,n):
            A.append(x)
    return A

print(decompresser_rle([2,1,3,2,2,1,3,3,1,4]))
print(decompresser_rle([1,1,1,2,1,1,1,2]))
print(decompresser_rle([7,3]))
```

```
[1, 1, 2, 2, 2, 1, 1, 3, 3, 3, 4]

[1, 2, 1, 2]

[3, 3, 3, 3, 3, 3, 3]
```

Pour la compression, il faut parcourir la liste A en notant le nombre d'éléments identiques consécutifs rencontrés. Pour cela, on utilise un indice **`i0`** qui représente le début d'une plage de nombre consécutifs identiques ainsi qu'un indice **`i`** pour parcourir la liste A. Lorsque $A[i] = A[i_0]$, on continue à avancer dans la liste. Lorsque $A[i] \neq A[i_0]$ il faut écrire le nombre d'éléments consécutifs rencontrés dans la liste B et repartir pour une nouvelle plage de valeurs consécutives à partir de i.

```
def compresser_rle(A):
    n = len(A)
    B = []
    i0 = 0
    for i in range(0,n):
        if A[i]!=A[i0]:
            B.append(i-i0)
            B.append(A[i0])
            i0 = i
    B.append(n-i0)
    B.append(A[i0])
    return B

print(compresser_rle([1,1,2,2,2,1,1,3,3,3,4]))
print(compresser_rle([1,2,1,2]))
print(compresser_rle([3,3,3,3,3,3,3]))
```

```
[2, 1, 3, 2, 2, 1, 3, 3, 1, 4]
[1, 1, 1, 2, 1, 1, 1, 2]
[7, 3]
```