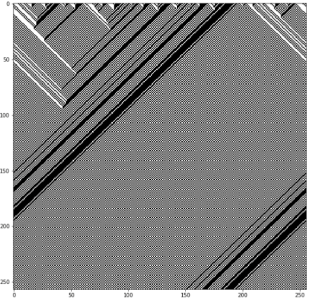
# TP automates cellulaires (correction)



Hervé Wozniak - Université de Montpellier

*Paternité - Pas d'Utilisation Commerciale - Pas de Modification : http://creativecommons.org/licenses/by-nc-nd/4.0/fr/*

13 novembre 2018

v 0.1

# Table des matières

**Introduction** 3

I - **Solutions** 4

1. Exercice n°1 ..... 4
2. Exercice n°2 ..... 5
3. Exercice n°3 ..... 6
4. Exercice n°4 ..... 7
5. Exercice n°5 ..... 7

# Introduction

La difficulté principale du TP sur l'Automate Cellulaire est de comprendre les séquences d'encodage/décodage entre décimaux et représentation binaire, à la base de la notation de Wolfram. C'est lié à l'état des cellules : 0 ou 1. Si une cellule pouvait avoir 4 états, il faudrait envisager des conversions entre décimaux et base 4.

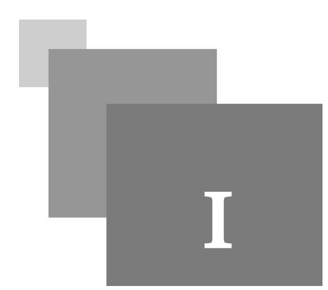
Pour le TP proposé, toute solution est donc basée sur la conversion d'entiers entre la base 10 et la base 2. On se précipite aisément sur la fonction intrinsèque *int()*, alors qu'elle n'est pas introduite en cours, et on se retrouve à devoir gérer des chaînes de caractères. Or, l'utilisation des chaînes de caractères n'est pas le plus efficace pour traiter un objet qui est principalement numérique.

Sachant que l'état d'une cellule vaut 0 ou 1, les conversions dont on a besoin sont les suivantes :

- la combinaison de l'état d'une cellule avec l'état des cellules voisines définit un nombre en binaire. Par exemple, à l'ordre 3, on considère la cellule et ses deux voisins, ce qui donne un nombre à 3 chiffres. 000, 001, 010, 011, 100, 101, 110, 111 sont les $8 = 2^3$ possibilités (à l'ordre $k$ on dispose de $2^k$ possibilités). Ces 8 nombres peuvent s'encoder en décimal est valent trivialement 0, 1, 2, 3, 4, 5, 6, 7. Quand j'ai besoin de connaître l'état d'une cellule et de son voisinage, je n'ai besoin que d'un chiffre en base 10 compris entre 0 et 7, et non d'une succession de 3 chiffres valant 0 ou 1.
- selon la valeur de l'état du triplet de cellules, cet état valant donc entre 0 et 7 en entier décimal, la règle de l'AC me permet de définir le nouvel état de la cellule. La règle est donc composée de 8 possibilités de remplacer la valeur de la cellule centrale par un 0 ou 1. Au lieu de faire 8 tests conditionnels (*if*), on écrit ces 8 possibilités sous forme d'un nombre écrit en binaire de longueurs 7 digits. Au lieu de parler de la règle 01111110, on convertit ce nombre en entier base 10, soit 126.

La programmation de l'AC consiste à faire ces opérations de conversion dans le sens inverse. Partant de la règle énoncée en base 10 (par exemple 126), on la convertit en représentation binaire, ce qui nous donne un nombre de 8 digits fait de 0 et de 1 (01111110). La règle, ainsi conservée sous forme de *list()* de 0 et de 1, permet de définir le nouvel état de la cellule lorsqu'on connaît l'état initial avec voisinage. Cet état initial est calculé comme un nombre valant entre 0 et 7, obtenu par conversion en décimal de la suite des trois 0 et 1. Ce nombre, entre 0 et 7, peut être utilisé comme indice dans la liste 'règle', à condition d'avoir ordonné cette *list()* de façon intelligente. En binaire, les digits les plus à droite représentent les valeurs les plus petites, alors que l'élément d'indice 0 dans une *list()* est formellement à gauche. D'où le renversement de la règle.

# Solutions



## 1. Exercice n°1

*Objectif*

Convertir une règle $\mathcal{R}$ numérotée selon la convention de Wolfram en une liste python (`list`) faite de 0 et de 1 contenant la représentation binaire sur 8 bits, le bit de poids faible étant à gauche afin qu'il corresponde à l'élément [0] de la `list`.

En entrée, la fonction prend le nom de la règle et l'ordre de l'AC (k=3 dans le cas Wolfram, mais la fonction est facilement généralisable pour traiter toutes valeurs de $k$).

```
1 >>> rule_to_binlist(126,order=3)
2 [0, 1, 1, 1, 1, 1, 1, 0]
3 >>> rule_to_binlist(110,order=3)
4 [0, 1, 1, 1, 0, 1, 1, 0]
```

*Exemple*

```
1 def rule_to_binlist(rule_name=126,order=3):
2     # rule_name: nom de la règle dans la notation Wolfram
3     # order: nombre cellules voisines (à 1D) incluant la cellule centrale
4 
5     # on s'assure que rule_name est bien un entier et on initialise la
décomposition en base 2 (binaire)
6     N=int(rule_name)
7 
8     # c'est l'algorithme classique qui remplace l'utilisation de bin()
9     # on l'adapte au besoin d'avoir le bit de poids faible (2**0) à gauche plutôt
qu'à droite
10    # pas besoin de padding sur les poids forts car les 0 sont automatiquement
calculés
11 
12    #liste des différents valeurs de 2**n en décroissant
13    # par défaut (order=3, donc 2**3 valeurs, donc 256 règles) : [128, 64, 32,
16, 8, 4, 2, 1]
14    base=list(2**i for i in range(2**order-1,-1,-1))
15 
16    #on prépare la liste qui contiendra la règle sous forme de 0 et 1 :
17    rule=list()
18 
19    # on parcourt du plus grand 2**(2**order-1) à 2**0
20    for i in range(len(base)):
21        F=N//base[i]    # division euclidienne qui donne 0 ou 1
22        N=N%base[i]     # reste de la division pour l'itération suivante ou 0/1 si
dernier
23        rule=[F]+rule   # insert à gauche car les divisions commencent toujours
pas le poids fort
```

```
24        # on pourrait aussi faire append() puis reverse() de la liste ; le
slicing est plus puissant
25    return rule
```

### *Complément : Solution avec int()*

```
1 def rule_to_binlist(rule_name=126,order=3):
2     # rule_name: nom de la règle dans la notation Wolfram
3     # order: nombre cellules voisines (à 1D)
4 
5     # on s'assure que rule_name est bien un entier
6     rule_name=int(rule_name)
7 
8     # renversement pour poid faible à gauche indice 0
9     # extraction de len(bin(rule_name)) a 1 exclu par pas de -1
10    # on ne veut pas conserver le '0b' renvoyé par bin()
11    # convertion des caractères individuels en entier
12    # mise en list :
13    rule=list(int(i) for i in list(bin(rule_name)[:1:-1]))
14 
15    # padding poid fort à droite
16    # le nb de 0 dépend de la magnitude de rule
17    rule=rule+(2**order-len(rule))*[0]
18    return rule
19 
```

# 2. Exercice n°2

*Objectif*

Il s'agit d'écrire une fonction qui, pour un $k$-uplet d'états de cellules en entrée, renvoie la conversion en valeur entière (`int`).

Autrement dit, pour un automate d'ordre 3, le triplet sous forme de `list` [0,1,1] renvoie la valeur `int` 3 qui sera interprétée comme étant $\alpha_3$ :

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

### *Exemple*

```
1 def state(ntuplet):
2     # valeurs decimales de la base 2 sur la longueur du ntuplet
3     base=list(2**i for i in range(len(ntuplet)-1,-1,-1)) # voir explications
exercice n°1
4 
5     # on fait le produit des 0/1 par leur représentation en 2**n et la somme
6     return sum(ntuplet[i]*base[i] for i in range(len(ntuplet)))
```

### *Complément : Solution avec int()*

```
1 def state(ntuplet):
2     #d'une liste de k-états (ntuplet) retourne l'indice alpha de l'état
```

```
3     # entre 0 et 7 (2^0 a 2^order-1) pour order=3
4     # ressemble à binlist_to_rule sauf qu'il n'y a pas d'inversion des bits
5     # puisqu'il s'agit d'un vrai binaire
6     # ici ntuplet est traité comme une chaine de caractères ; donc on saute les
"[" et les ","
7     return int('0b'+str(ntuplet)[1::3],base=2)
8
```

# 3. Exercice n°3

*Objectif*

On considère une liste d'états de longueur $N$ qui constitue la *condition initiale* de l'AC.

Une fonction (appelée *fenêtre glissante*) est appliquée à cette liste. On l'appelle *fenêtre* car elle retourne une extraction de $k$ valeurs de la liste d'origine centrée sur une cellule donnée. On l'appelle *glissante* car à chaque appel on décale l'extraction d'une cellule vers la droite (indice de `list` croissant).

*Attention*

A l'itération $0$, cette fonction retourne une liste de $k$ valeurs dont la cellule 0 est au centre. Cela implique que $\frac{k-1}{2}$ valeurs se trouvent à gauche de $0$.

Pour réaliser cette condition, on applique des *conditions aux limites périodiques*. On va donc copier les valeurs qui se trouvent tout à droite de la liste.

A l'itération $i$, elle retourne une liste de $k$ valeurs centrée sur la cellule$i$.

A l'approche des dernières itérations, la liste est complétée avec les valeurs à gauche de la liste.

*Complément : Python avancé*

Au lieu d'appeler $N$ fois une fonction qui ne renvoie qu'une seule valeur (fonction se terminant par `return`), on peut faire un usage avantageux d'une fonction génératrice se terminant par l'instruction `yield`, qui, combinée à une boucle dans la fonction, permet de ne l'appeler qu'une seule fois.

S'exercer sur l'exemple ci-dessous avant de l'utiliser pour l'AC.

```
1 def test_yield(niter):
2     for i in range(niter):
3         yield i
4
5 for j in test_yield(10):
6     print(j)
7
```

Appliqué à la fonction demandée, cela peut donner le comportement suivant, où l'on constate que la liste renvoyée contient toutes les fenêtres de triplets possibles :

```
1 >>> print(list(moving_window([1,2,3,4,5,6], wsize=3)))
2 [[6, 1, 2], [1, 2, 3], [2, 3, 4], [3, 4, 5], [4, 5, 6], [5, 6, 1]]
```

*Exemple*

```
def moving_window(arr, wsize=3):
    # arr: list en entree
    # wsize: largeur de la fenetre a faire glisser vers la droite
    #    doit être égale à order (k) ordre de l'AC Wolfram
    # complément à gauche et à droite pour les conditions périodiques
    # arr[-1:] = arr[:dernier]

    arr = arr[-wsize//2 + 1:] + arr + arr[:wsize//2]

    # glissement pas à pas avec la cellule concernée au centre:
    for i in range(len(arr) - wsize + 1):
        # retourne la cellule centrale (ici i+1) et son k-voisinage (ici wsize)
        yield arr[i:i + wsize]
```

# 4. Exercice n°4

*Objectif*

L'AC étant défini comme l'application d'une règle sur un $k$-uplet, il s'agit maintenant de créer une fonction qui réalise les changements d'état.

Pour un $k$-uplet en entrée, la fonction retourne le $k$-uplet modifié par la règle. Puisque les cellules d'un AC changent d'état de façon synchrone, on peut profiter de cette fonction pour parcourir toutes les cellules. Ainsi, on applique la fonction sur la liste complète des cellules. En sortie on récupère la liste mise à jour avec les nouveaux états.

On tirera partie des exercices précédents :

- l'exercice 2 permet de trouver la valeur de $\alpha$ qui viendra remplacer l'ancien état ;
- l'exercice 3 déplace une fenêtre de largeur $k$ le long de la liste des cellules.

```
>>> print(apply_rule([0,0,1,0,0],rule_to_binlist(rule_name=110,order=3),order=3))
[0, 1, 1, 0, 0]
```

*Exemple*

```
def apply_rule(cells, rule, k):

    # w contient une instance de la fenêtre glissante moving_window()
    # state(w) convertit le k-uplet en int
    # rule[] contient le nouvel état pour state(w) qui sert d'indice

    return [rule[state(w)] for w in moving_window(cells, k)]
```

# 5. Exercice n°5

*Objectif*

Toutes les fonctions de base ont été construites. Il faut maintenant prévoir l'avancement dans le temps qui se réalise sous forme d'itération synchronisée pour toutes les cellules.

On écrit donc une fonction qui, à chaque appel, avance d'un pas de temps l'AC.

Autrement dit, à un instant $t + 1$, la fonction applique la règle $\mathcal{R}$ sur l'état des cellules pris à l'instant $t$.

### *Attention*

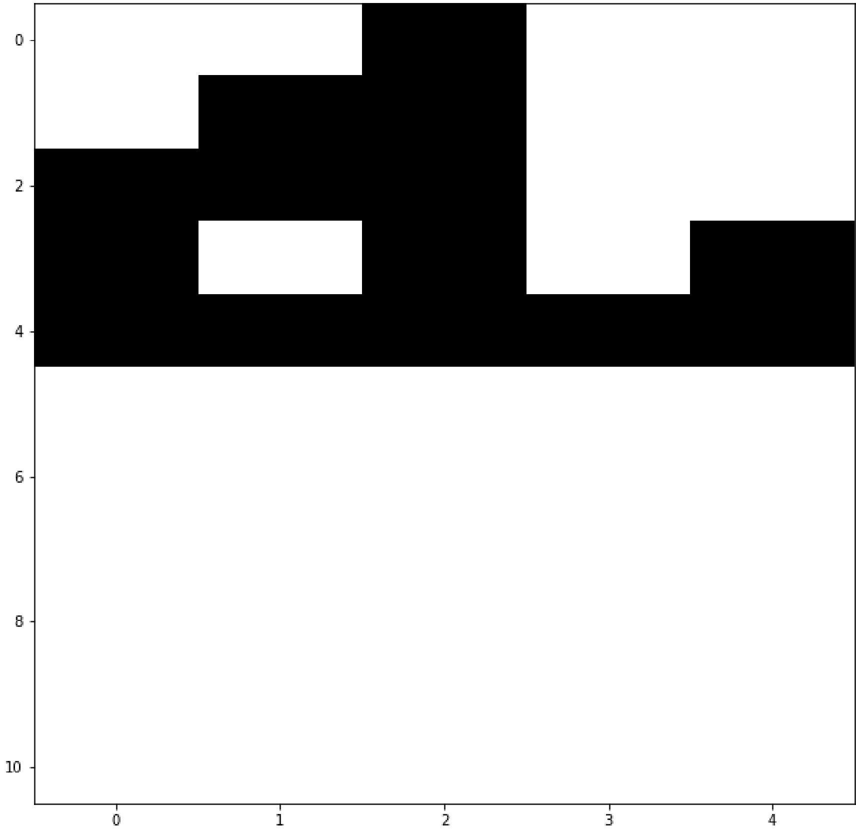
On se déplaçant de gauche à droite, il faut s'assurer que l'état de la (ou des) cellule de gauche soit pris à l'instant $t$ et non $t$+1. A priori ce problème doit avoir été résolu à l'exercice n°4.

Un exemple de résultat attendu ;

```
1 >>> output=ca_iterate(init_cond=[0,0,1,0,0],rule=110,order=3,niter=10)
2 >>> print(list(output))
3 [[0, 0, 1, 0, 0], [0, 1, 1, 0, 0], [1, 1, 1, 0, 0], [1, 0, 1, 0, 1], [1, 1, 1, 1,
  1], [0, 0, 0, 0, 0], [0, 0, 0, 0, 0], [0, 0, 0, 0, 0], [0, 0, 0, 0, 0], [0, 0, 0
  , 0, 0], [0, 0, 0, 0, 0]]
```

La fonction d'itération peut renvoyer une liste de liste (ou un objet *generator* produit par l'instruction `yield`) qui se trace alors comme suit :

```
1 >>> import numpy as np
2 >>> tab=np.array(list(output))
3 >>> import matplotlib.pyplot as plt
4 >>> plt.figure(figsize=(10,10))
5 >>> plt.imshow(tab,origin="upper",cmap="Greys",aspect='auto')
```




### *Exemple*

```
1 def ca_iterate(init_cond=[0,0,1,0,0],rule=126,order=3,niter=10):
2     state_cell=list(init_cond)
3     # retourne l'état initial de toutes les cellules, donc t=0
4     yield state_cell
5
6     # boucle sur le temps
7     for i in range(niter):
8         # on lit de droite à gauche, donc state_cell contient l'état précédent
  lors de l'appel à apply_rule
9         state_cell=apply_rule(state_cell,rule_to_binlist2(rule,order),order)
10        yield state_cell
11
```

```
1 def ca(rule=126,largeur=511,niter=256,order=3,init='centre'):
2     if init=='centre':
3         ca_init=[0]*largeur
4         ca_init[largeur//2]=1   # centré
5     else :
6         import random
7         ca_init=[random.getrandbits(1) for i in range(largeur)]
8
9     output=ca_iterate(ca_init,rule,order,niter)
```

```
10
11  import numpy as np
12  tab=np.array(list(output))
13  import matplotlib.pyplot as plt
14  plt.figure(figsize=(12,12))
15  plt.imshow(tab,origin="upper",cmap="Greys",aspect='auto')
16  plt.savefig("ca"+str(rule)+init+".png")
```