Chapitre 14

Introduction aux automates cellulaires

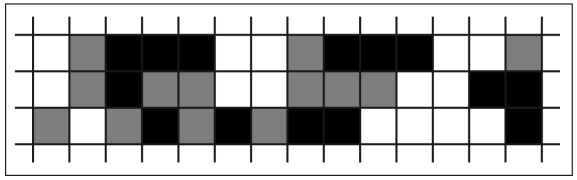
14.1 Automates cellulaires

14.1.1 Définition d'un Automate cellulaire

• C'est un modèle simple et riche d'automate : système évoluant de manière (pseudo-)déterministe au cours du temps.

• Un automate cellulaire est constitué de :

- Une grille régulière de cellules (1, 2 ou plus de dimensions)
- Un nombre fini d'états. Chaque cellule se trouve au temps t dans un état.
- L'état de la cellule au temps $t+1$ ne dépend que de son état et des états de ses cellules voisines au temps précédent t (et éventuellement à un aléatoire : non-déterministe).



- L'espace est discret, un sous-ensemble du réseau $\mathbb{Z}^d$ où $d = 1, 2, \ldots$ est la dimension.
- Le temps est discret : $[[0; T]] \subset \mathbb{N}$.
- Au temps 0 la grille contient un "motif" initial, donné par l'état de chacune de ses cellules.

• Inventés par le mathématicien génial John von-Neumann pour créer un "système auto-réplicatif".

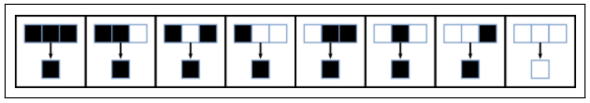
• Des règles simples permettent d'aboutir à des évolutions très complexes. Etudiés en Mathématiques car ils donnent des exemples intéressants de systèmes dynamiques discrets.

• Ils sont étudiés en informatique théorique pour leurs propriétés remarquables, et parce qu'ils constituent un modèle de calcul : modèle théorique d'ordinateur.

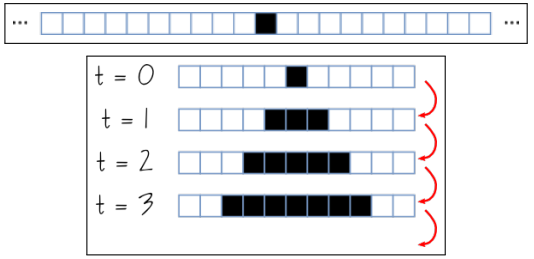
• Ils sont aussi amplement utilisés pour modéliser simplement l'évolution des systèmes physiques, biologiques, des systèmes complexes (percolation, diffusion évolution des feux de forêt, chutes de tas de sable, évolution du trafic routier, évolution d'une population, ...).

14.1.2 Exemples unidimensionnels ordonnés

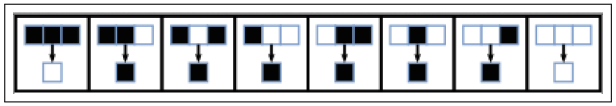
• Appliquer la règle suivante (règle 254) :



en partant d'une seule case noire :



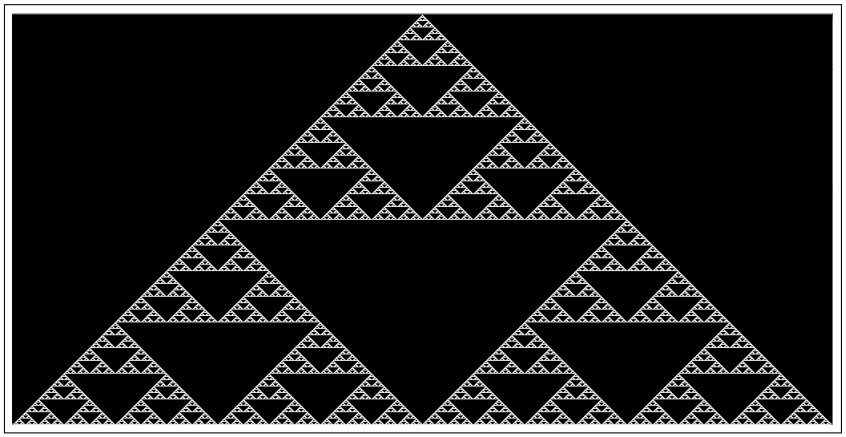
• Appliquer la règle suivante (règle 126) :



(seule différence, une case noire entourée de 2 cases noires devient blanche)

en partant d'une seule case noire :

```
import numpy as np
from PIL import Image
T = 512    # Nombre d'itérations
N = 2*T     # Nombre de cases
M = np.zeros((T,N))    # Tableau : ligne t = état des cases au
temps t
M[0,N//2] = 1        # Initialement : 1 seule case noire
for temps in range(1,T):
    for index in range(1,N-1):
    voisinage = M[temps-1,index-1] + M[temps-1,index] +
M[temps-1,index+1]
    if voisinage == 0 or voisinage == 3:
        M[temps,index] = 0
    else:
        M[temps,index] = 1
img2 = Image.new('1',(N,T))    # Nouvelle image en mode 1 : N&B
img2.putdata(np.reshape(M,T*N))    # Remplissage des pixels
img2.show()
```

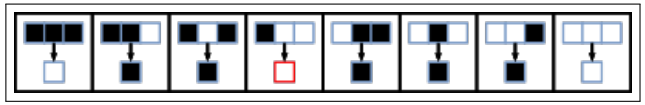


Ici on a inversé blanc = 1 et noir = 0, pour une meilleure visibilité.

On obtient le **triangle de Sierpinski.**

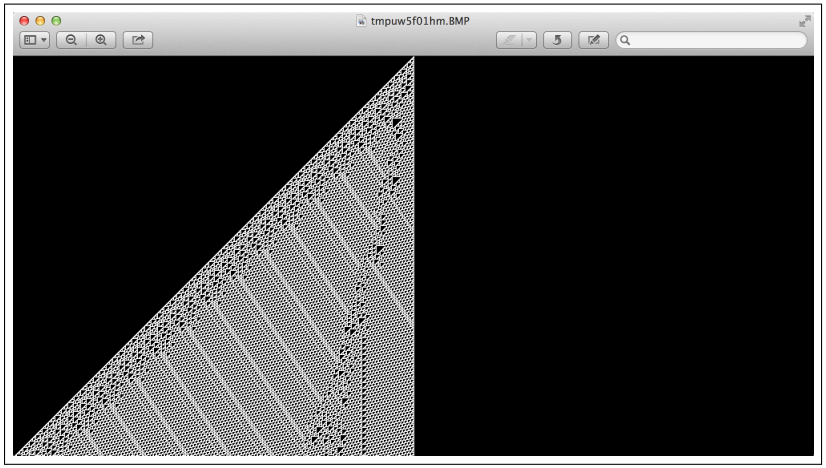
14.1.3 Exemple unidimensionnel émergent

- Appliquer la règle suivante 110, légère modification de la précédente :



en partant d'une seule case noire :

```
import numpy as np
from PIL import Image


T = 500    # Nombre d'itérations
N = 2*T     # Nombre de cases
M = np.zeros((T,N))    # Tableau : ligne t = état des cases au
temps t
M[0,N//2] = 1        # Intialement : 1 seule case noire
for temps in range(1,T):
    for index in range(1,N-1):
    voisinage = M[temps-1,index-1] + M[temps-1,index] +
M[temps-1,index+1]
    if voisinage == 0 or voisinage == 3:
        M[temps,index] = 0
    elif voisinage == 1 and M[temps-1,index-1] = 1:
        M[temps,index] = 0
    else:
        M[temps,index] = 1
img2 = Image.new('1',(N,T))
img2.putdata( np.reshape(M,T*N))
img2.show()
```



Les cases de droite demeurent nulles. A gauche au milieu d'un comportement régulier émerge un comportement chaotique, difficile à décrire.

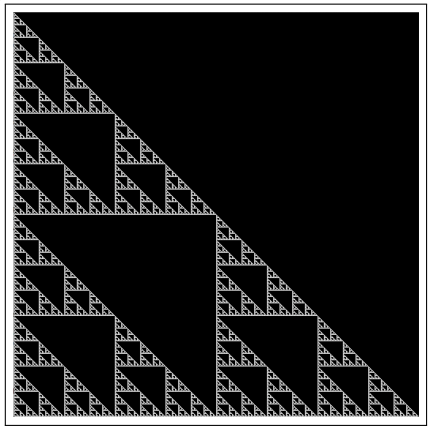
Cook a démontré (2002) que cet automate est Turing universel : il peut effectuer tout calcul faisable sur ordinateur...

14.1.4 Triangle de Sierpinski et Triangle de Pascal

• Dans le triangle de Pascal (des coefficients binomiaux), remplacer chaque coefficient par son reste dans la division euclidienne par 2 (0 pour pair, 1 pour impair) :

```
import numpy as np
from scipy.misc import comb     # Import de : comb(n,k) = (k
parmi n)

T = N = 1024
M = np.zeros((T,N))
for t in range(T):
    for index in range(N):
        if index <= t:
            M[t,index] = comb(t,index, exact = True) % 2
img = Image.new('1',(N,T))
img.putdata( np.reshape(M,T*N))
img.show()
```

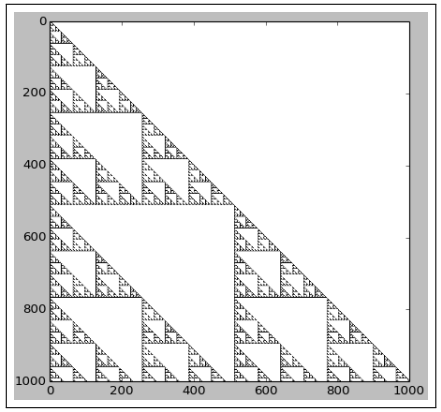
L'option `exact = True` dans `comb()` assure que le coefficient binomial retourné à une valeur exacte (autrement ce n'est qu'une valeur approchée donnée par un float).



En noir les coefficients binomiaux pairs, en blanc les coefficients impairs.

•Ou encore, à l'aide de la fonction `imshow()`, du module `pyplot` :

```
import matplotlib
import matplotlib.pyplot as plt
plt.imshow(M, cmap=matplotlib.cm.gray_r)    # Affichage en
niveau de gris
```



14.2 Exemples bidimensionnels

14.2.1 Le jeu de la vie

• **Le jeu de la vie** est le plus célèbre des automates cellulaires.

Il fût inventé en 1970 par le Mathématicien John Conway (Université de Cambridge), pour trouver une solution plus simple au problème originel de John von Neumann : construire une "machine" auto-réplicante.

• Son principe est simple :

- Automate cellulaire à 2 dimensions.
- Chaque cellule a deux états possibles : inactive (morte) ou active (vivante).
- Règles de transformations générationnelles :
  - Toute cellule entourée de 3 cellules vivantes devient active à la génération suivante.
  - Toute cellule entourée de 2 cellules vivantes garde son état à la génération suivante.
  - Toute cellule entourée de < 2 ou > 3 cellules vivantes meurt (d'isolement ou d'étouffement) à la génération suivante.

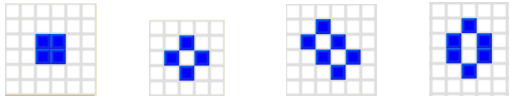
- En résumé :
  - Si 3 voisins vivants : naissance.
  - Si 2 voisins vivants : survivance.
  - Sinon : mort.

- Il possède des propriétés remarquables d'universalité intrinsèque : toute configuration s'obtenant d'un automate cellulaire 2-dimensionnel peut s'obtenir grâce au jeu de la vie. En particulier, en théorie il peut effectuer le calcul de tout algorithme (Universel $\Longrightarrow$ Turing universel).

L'essayer en ligne : `http://www.dcode.fr/jeu-de-la-vie`

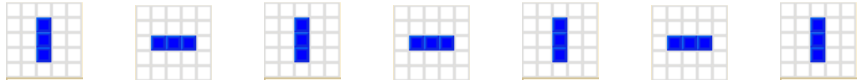
14.2.2 Quelques motifs remarquables

- Formes stables : (demeurent identiques au fil des générations)



- Oscillateurs (formes périodiques) :

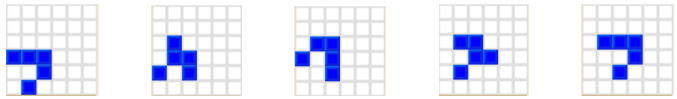
Le clignotant :



Le phare :

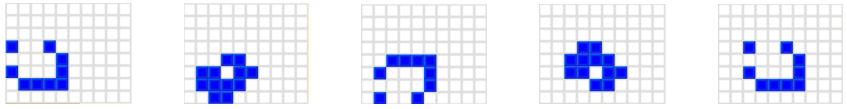
Quelques exemples de "mobiles" :

- Le planeur :



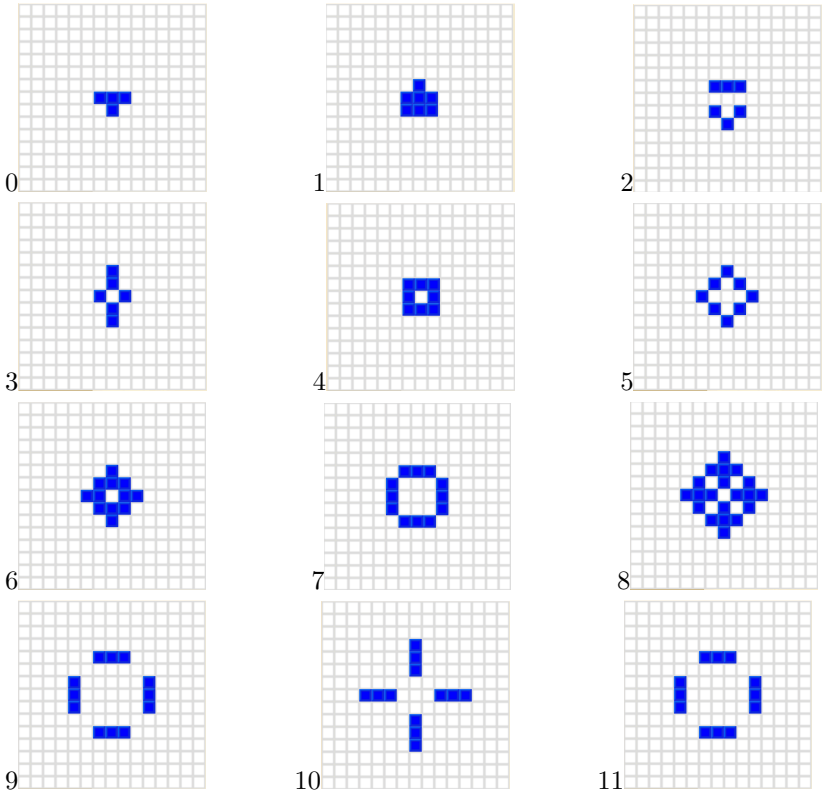
se déplace d'un cran en diagonale en 5 itérations.

- Le navire :



se déplace d'un cran sur la droite en 5 itérations.

- Mathusalem (se stabilise après quelques générations) :



14.2.3 Le jeu de la vie : programmation

- Le réseau sera représenté par un tableau (on utilisera un tableau bidimensionnel `numpy`). L'état initial (motif) sera un tableau prédéfini passé en paramètre.

Il

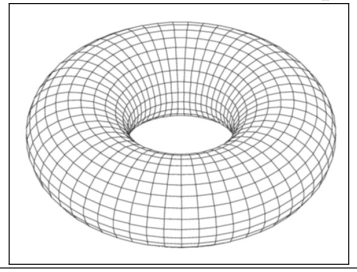
- A chaque génération, en remplit le tableau de la nouvelle génération à l'aide du tableau de la génération précédente. La boucle principale du programme est une boucle for qui parcourt les générations.

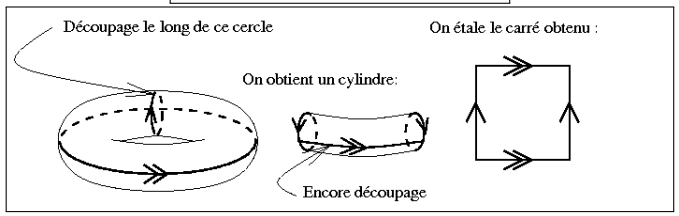
- Le remplissage de la génération suivantes se fait élément par élément à l'aide de 2 boucles for imbriquées où 2 indices parcourent les numéros de lignes et de colonnes.

Il

- Pour chaque élément on compte le nombre de cellules voisines vivantes. Puis on applique la règle pour déterminer l'état de la cellule à la prochaine génération.

• **Au bord de la grille : on identifie bord droit avec bord gauche, et bord haut avec bord bas.**
Les indices du tableau donnant la grille seront considérés modulo la largeur ou modulo la hauteur. Cela revient topologiquement à considérer que **les cellules sont disposées sur un tore** :






```
# Animation obtenue en traçant successivement les générations
import numpy as np
import matplotlib
import matplotlib.pyplot as plt

def jeuVie(R, Temps, dt = 1.e-3):
    # R : état (grille) initial(e)
    # Temps : nombre de générations
    # dt : Intervalle de temps entre deux affichages. Par
défaut dt = 0.001
    dimY = len(R)    # Nombre de lignes de la grille
    dimX = len(R[0])    # Nombre de colonnes de la grille
    # Tracé au temps 0
    plt.figure("Le jeu de la vie")
    plt.clf()
    img = plt.imshow(R,interpolation = 'nearest', /
                     cmap=matplotlib.cm.binary)
    plt.title('Temps 0')
    plt.draw()
    plt.pause(dt)
```

L'affichage produit une image en N&B (avec 0 :blanc, 1 : Noir).

```
    # Animation :
    for t in range(Temps):
        # Génération suivante
        P = np.zeros((dimY,dimX))    # génération suivante
        population = 0          # Compteur de population
        for i in range(dimY):
            for j in range(dimX):
                ip, jp = (i+1)%dimY, (j+1)%dimX # Tracé
                voisins = R[i-1,j-1] + R[i-1,j] + R[i-1,jp]\
                        + R[i,j-1] + R[i,jp]+ R[ip,j-1]\
                        + R[ip,j] + R[ip,jp]
                if voisins == 3:
                    P[i,j] = 1     #naissance
                    population += 1
                elif voisins != 2:
                    P[i,j] = 0     #mort
                else:
                    P[i,j] = R[i,j]     #survivance
                    population += P[i,j]
        R = P          # Tracé de l'état au temps t
        # pour la fluidité, appeler 1 seul imshow() :
        img.set_data(R)
        titre='Temps '+str(t+1)+' : Population = ' /
                                  +str(int(population))
        plt.title(titre)
        plt.draw()
        plt.show()
        plt.pause(dt)
    # Fin de JeuVie()
```

On écrit aussi des fonctions pour définir des formes initiales :

• Le planeur : 

```
def planeur(n):
    M = np.zeros((n,n))
    M[n-1,2], M[n-2,3] = 1,1
    M[n-3,1], M[n-3,2], M[n-3,3] = 1,1,1
    return M
```

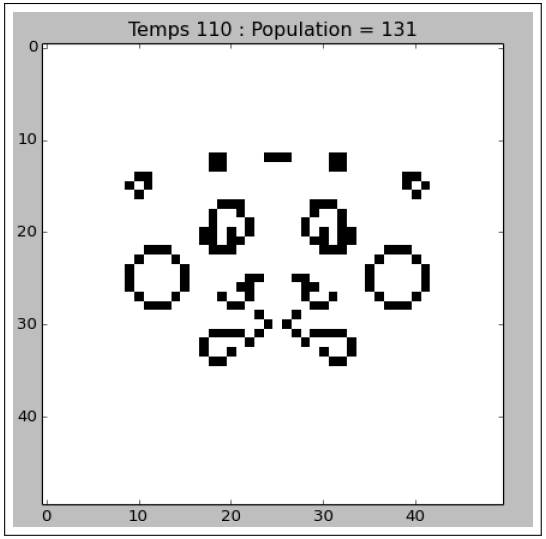
• Le navire : 

```
def navire(n):
    M = np.zeros((n,n))
    y = n//2
    M[y-3,1], M[y-3,4] = 1, 1
    M[y-2,5] = 1
    M[y-1,1], M[y-1,5] = 1,1
    M[y,2], M[y,3], M[y,4], M[y,5] = 1, 1, 1, 1
    return M
```

• Le clown (au bout de 110 itérations) : 

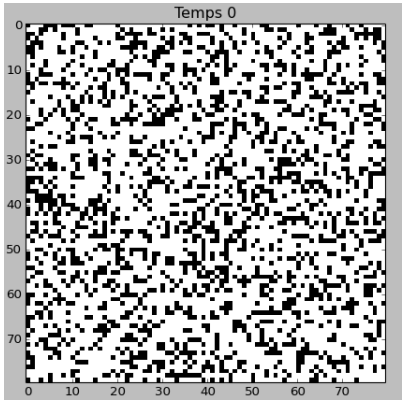
```
def clown(n):
    M = np.zeros((n,n))
    m = n//2
    M[m-3,m-1],M[m-3,m],M[m-3,m+1] = 1,1,1
    M[m-2,m-1], M[m-2,m+1] = 1,1
    M[m-1,m-1], M[m-1,m+1] = 1, 1
    return M
```

Le clown  devient après 110 itérations :

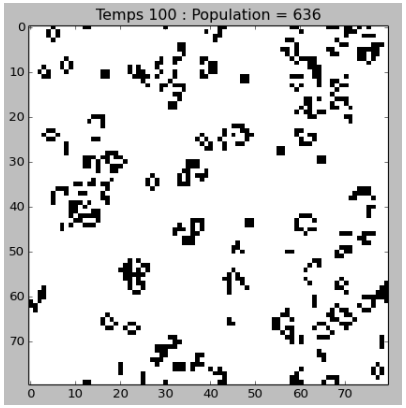


• Distribution aléatoire (de densité donnée) :

```
from random import random

def alea(n, p=0.3):
    M=np.zeros((n,n))
    for i in range(n):
        for j in range(n):
            if random() < p:
                M[i,j]=1
    return M
```



`alea(80)`



Après 100 générations.

14.2.4 Le compteur de Fredkin

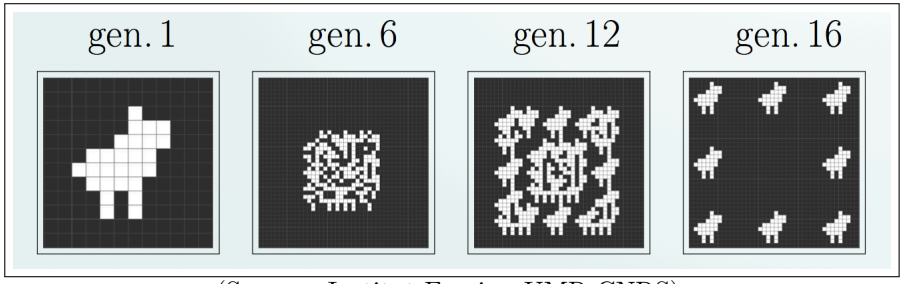
• Au contraire du jeu de la vie, il est basé sur une règle simple qui provoque un résultat très ordonné (aucun phénomène chaotique ou émergent).

• Sa règle est très simple :

Bidimensionnel ; chaque cellule à deux états possibles : active ou inactive.

- Si le voisinage d'une cellule (ses 8 cellules adjacentes) contient un nombre impair de cellules actives, alors la cellule sera active à la prochaine génération.
- Sinon elle sera inactive.

• Le résultat est assez étonnant : il duplique périodiquement tout motif initial :

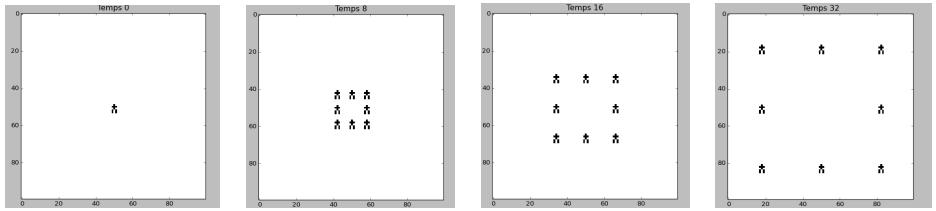


(Source : Institut Fourier, UMR CNRS).

```
def fredkin(R, Temps, dt = 1.e-3):
    dimY, dimX = len(R), len(R[0])
    img = plt.imshow(R,interpolation = 'nearest', /
                     cmap=matplotlib.cm.binary)
    plt.draw()
    plt.pause(dt)
    for t in range(Temps):
        P = np.zeros((dimY,dimX))
        for i in range(dimY):
            for j in range(dimX):
                ip, jp = (i+1) % dimY, (j+1) % dimX
                voisins = R[i-1,j-1] + R[i-1,j] + R[i-1,jp]\
                        + R[i,j-1] + R[i,jp]+ R[ip,j-1]\
                        + R[ip,j] + R[ip,jp]
                if voisins % 2 == 1:
                    P[i,j] = 1 #active
                else:
                    P[i,j] = 0 #inactive
        R = P
        img.set_data(R)
        titre='Temps '+str(t+1)
        plt.title(titre)
        plt.draw()
        plt.pause(dt)
```

• On part d'un motif arbitraire :  que l'on passera en paramètre :

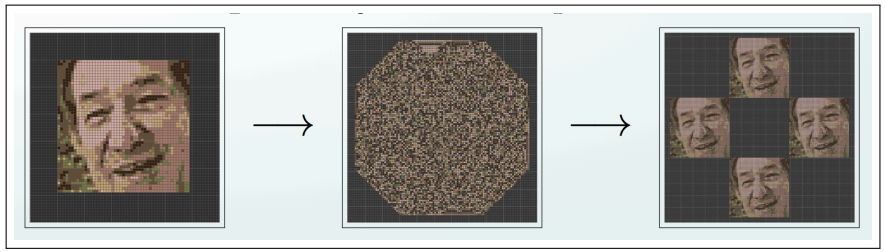
```
def motif(n):
    M = np.zeros((n,n))
    a = n//2
    M[a-1,a], M[a+1,a] = 1,1
    M[a,a-1], M[a,a], M[a,a+1] = 1,1, 1
    M[a+1,a] = 1
    M[a+2,a-1], M[a+2,a+1] = 1,1
    M[a+3,a-1], M[a+3,a+1] = 1,1
    return M
```



Initialement 8 générations 16 générations 32 générations

• Le compteur de Fredkin se généralise à un automate à $p \geqslant 2$ états dès que p est premier, en prenant pour règle :

Règle de Fredkin : Chaque cellule prend pour état la somme modulo p de ses 8 voisines.

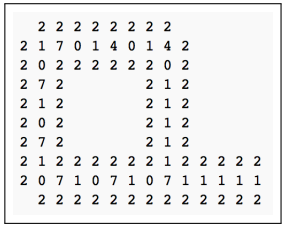


Ici $p = 7$, après 7 itérations. (Source : Institut Fourier, UMR CNRS.)

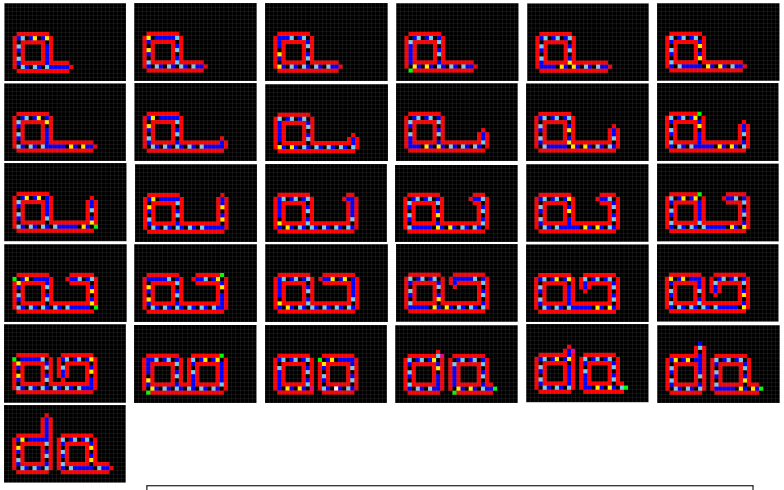
14.2.5 Boucle de Langton

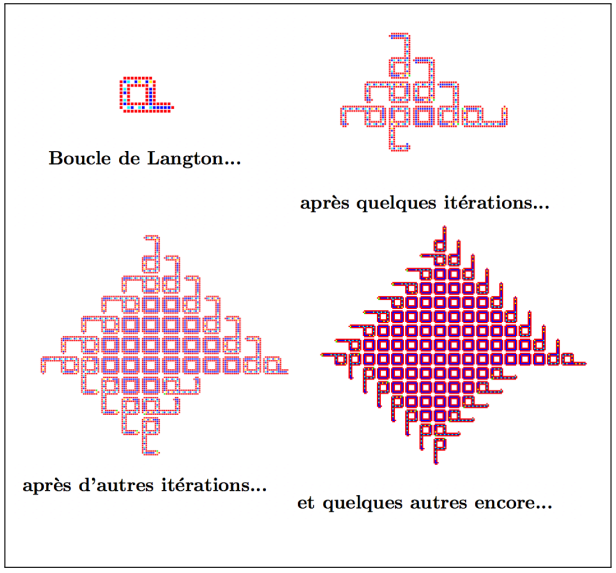
• Un A.C. **auto-réplicant** à 8 états et 86 cellules soumis à 219 règles (Langton, 1984).

• Le motif initial qui s'auto-réplique est constitué d'une membrane (état 2) entourant un brin d'"ADN" central qui code le processus d'auto-réplication, notamment la mort de la cellule (qui devient une boucle fermée inerte).



• Un A.C. **auto-réplicant** à 8 états et 86 cellules soumis à 219 règles (Langton, 1984).





Source : Poster, Institut Fourier (Université Joseph Fourier, Grenoble).

14.3 Application en modélisation

14.3.1 Automate cellulaire et simulation

• Les automates cellulaires sont largement utilisés pour la simulation de systèmes complexes.

• Il s'agit d'un modèle simplifié (les agents sont soumis à des règles simples) qui en général comporte une part d'aléa, soit dans la configuration initiale soit dans les règles d'évolution.

Ils sont employés notamment pour la simulation de :

1. trafic routier,
2. phénomènes de percolation, notamment l'évolution de feux de forêts.
3. Diffusion particulaire.
4. avalanches d'un tas de grains de sables (Sujet 0 Mines-Ponts).
5. Désintégration de particules radioactives.
6. etc...

Un système physique complexe régi par des équations différentielles peut être discrétisé et soumis à une modélisation par un A.C. (qui sera pertinente ou non). (Un schéma d'Euler, ou autre permet la discrétisation d'un équation différentielle et la définition d'un modèle par A.C.).

14.3.2 Percolation : Simulation d'un feu de forêt

• On utilise une grille bidimensionnelle carrée. Au départ chaque cellule est soit vide, soit arborée.

Notre motif initial (zone forestière) sera construit aléatoirement donnés le nombre de cases en largeur, et la densité p de zones arborées.

• Les règles d'évolution sont :

> Si une cellule arborée a au moins une cellule voisine en feu, elle passe à l'état "en feu".
>
> Une cellule en feu passe au temps suivant à l'état "en cendre".

Les cellules peuvent prendre 4 états : vide, arborée (vivace), en feu, en cendre.

• Forêt aléatoire densité taille et donnée, et son tracé :

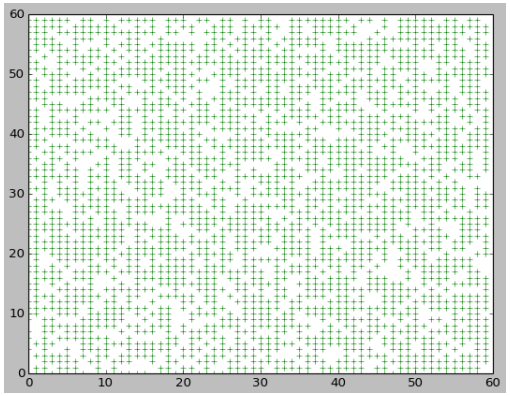
```
from random import random

def foret(n = 60, p = 0.7):
    M = np.zeros((n,n))
    for i in range(n):
        for j in range(n):
            if random() < p:
                M[i,j] = 1
    return M
```

0 sera l'état d'une cellule vide, 1 sera l'état d'une cellule arborée vivace.

• Nous allons compléter une fonction d'affichage. Une cellule arborée vivace sera marquée d'une croix verte :

```
def affichage(T):
    n = len(T)
    X1 = []
    Y1 = []
    for i in range(n):
        for j in range(n):
            if T[i,j] == 1:
                X1.append(j)
                Y1.append(i)
    plt.plot(X1,Y1,'g+')
    plt.show()
```

```
>>> T = foret()
>>> affichage(T)
```



• On prendra pour états :

0 : vide ; 1 : vivace ; 2 : en feu ; 3 : en cendres

• On modifie la fonction d'affichage de sorte que :

vide : rien

vivace : croix verte

en feu : croix rouge

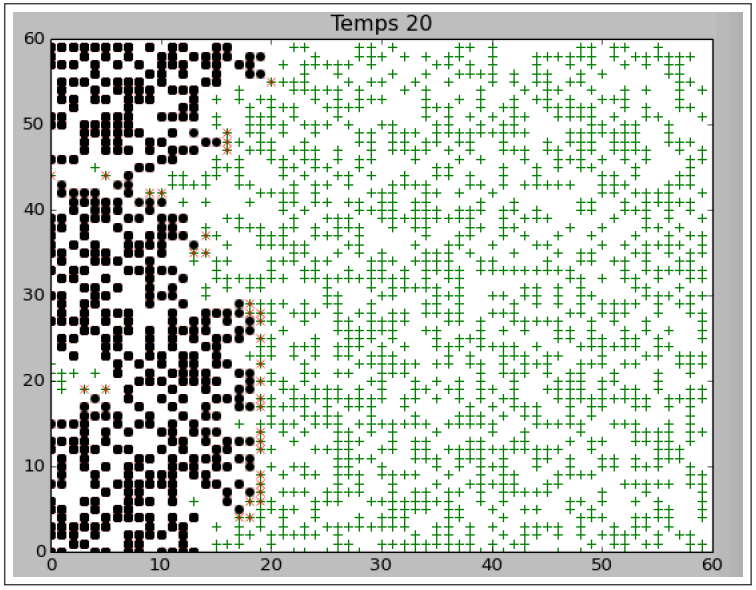
en cendre : rond noir

```
def affichage(T):
    n = len(T)
    X1, X2, X3 = [], [], []    # Listes des abscisses pour
états 1, 2, 3
    Y1, Y2, Y3 = [], [], []    # Listes des ordonnées pour
états 1, 2, 3
    for i in range(n):
        for j in range(n):
        if T[i,j] == 1:    # Si état == 1 (vivace)
            X1.append(j)
            Y1.append(i)
        elif T[i,j] == 2:    # Si état = 2 (en feu)
            X2.append(j)
            Y2.append(i)
        elif T[i,j] == 3:    # Si état == 3 (en cendres)
            X3.append(j)
            Y3.append(i)
    plt.figure(1)
    plt.plot(X1,Y1,'g+')    # croix verte
    plt.plot(X2,Y2,'rx')    # croix rouge
    plt.plot(X3,Y3,'ko')    # point noir
    plt.show()
```

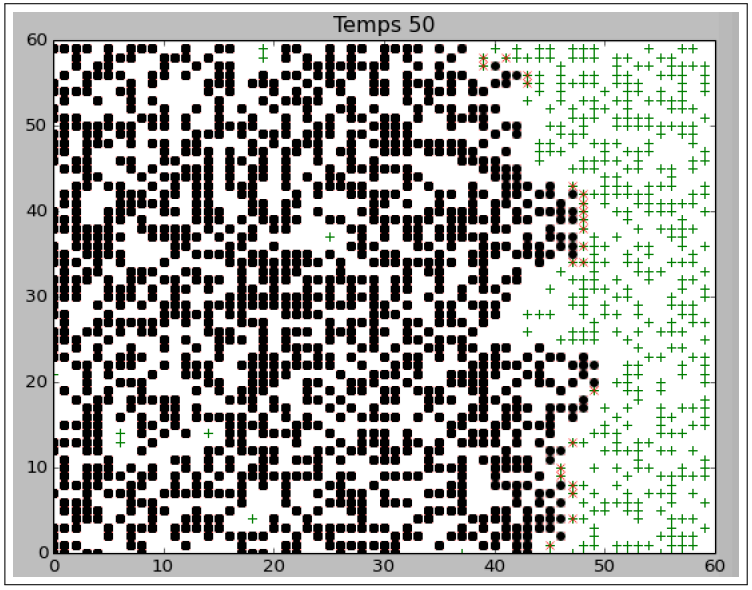
• Pour simplifier on prend 3 départs de feu, régulièrement espacés sur le bord gauche.

```
def feuForet(F, T = 100, dt = 1e-3):
    n = len(F)
    F[0,0], F[n//2,0], F[n-1,0] = 2, 2, 2    # Départs de feu
    affichage(F)
    plt.pause(dt)
```

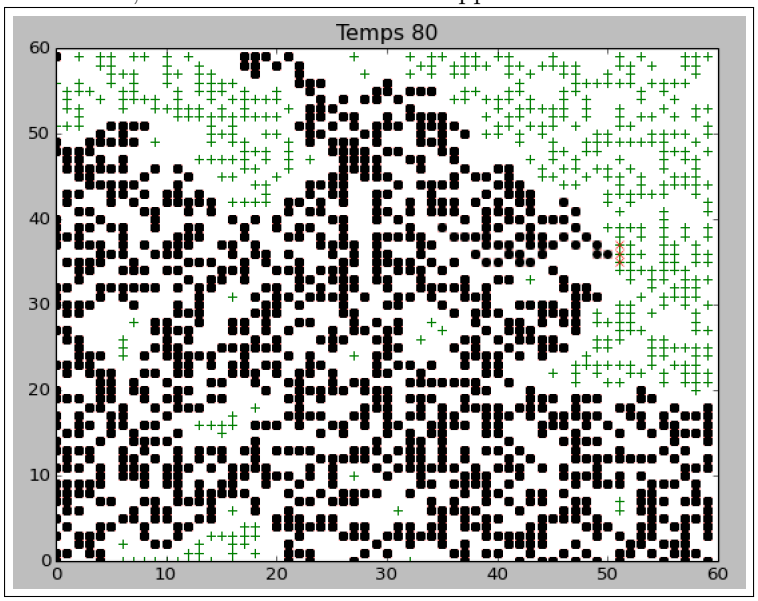
```
for t in range(T):
    F1 = np.zeros((n,n))
    for i in range(n):
        for j in range(n):
            if F[i,j] == 2: #si en feu
                F1[i,j] = 3 #passe en cendre
            elif F[i,j] != 1: #sinon si non vivace
                F1[i,j] = F[i,j] # inchangé
            else: # si vivace
                F1[i,j] = 1
                for ii in range(max(0,i-1),/
                                        min(n-1,i+1)+1):
                    for jj in range(max(0,j-1), /
                                    min(n-1,j+1)+1):
                        if F[ii,jj] == 2: #voisin en feu
                            F1[i,j] = 2 #alors prend feu
    F = F1
    titre = 'Temps ' + str(t+1)
    plt.title(titre)
    affichage(F)
    plt.pause(dt)
```



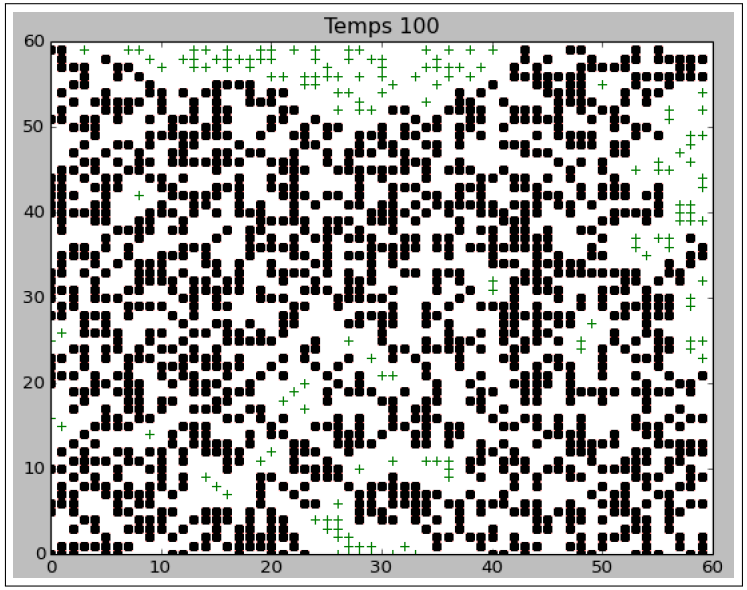
Densité = 0.5, T = 20



Densité = 0.5, T = 50. Peu de zones échappent aux flammes.



Densité = 0.45, T = 80. Certaines zones échappent aux flammes. Percolation : le feu a traversé la zone.



Densité = 0.45, T = 100. Certaines zones ont échappé aux flammes. Percolation : le feu a traversé la zone.

14.3.3 Diffusion de particules

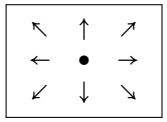
• On considère un bassin parrallélépipédique de longueur et largeur L, et de profondeur 1, dans lequel, à l'instant 0, N particules sont disposées au centre : on modélise leur évolution au cours du temps dans un milieu homogène fluide.

Programmation :

- Le bassin est modélisé par une grille bidimensionnelle carrée de $L \times L$ cellules. L'état de chaque cellule au temps t est le nombre de particules qu'elle contient.
- Chaque particule est soumise à un mouvement brownien : simulé par une marche aléatoire sur le réseau Z^2.
- Conditions au bord : dans notre approche, une particule atteignant le bord sort du bassin.

• On considère un automate cellulaire bidimensionnel sur une grille de taille $L \times L$, ou chaque cellule prend $N+1$ états possibles (son nombre de particules).

Au temps t chaque particule de façon équiprobable se déplace aléatoirement sur l'un des ses 8 cellules adjacentes ou reste à sa position.



(chaque déplacement (ou pas) a même probabilité 1/9.)

• Importation des modules :

```
import numpy as np
import matplotlib
import matplotlib.pyplot as plt
from random import randint
```

• Le module `random` est celui permettant de générer des nombres pseudo-aléatoires :

`randrange(a,b,k)`	Choisit un entier aléatoirement dans `range(a,b,k)`
`randint(a,b)`	Choisit un entier aléatoirement dans $[[a,b]]$
`choice(List)`	Choisit un entier aléatoirement dans la liste `List`
`random()`	Choisit un float aléatoirement dans $[0,1[$
`uniform(a,b)`	Choisit un float aléatoirement dans $[a,b[$

• Nous utiliserons `randint(-1,1)` pour tirer au sort les déplacements horizontaux et verticaux de chaque particule :

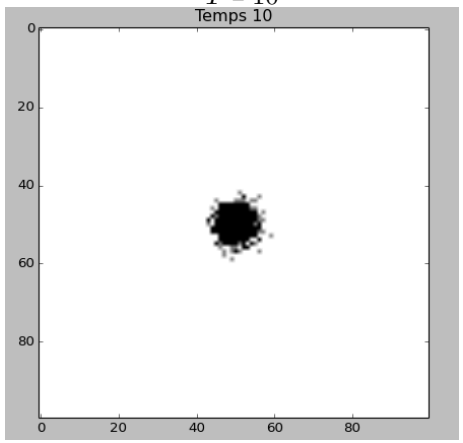
Si au temps t une particule est à la position `[i,j]`, au temps $t+1$ elle sera en :

`[i + randint(-1,1), j + randint(-1,1)]`.

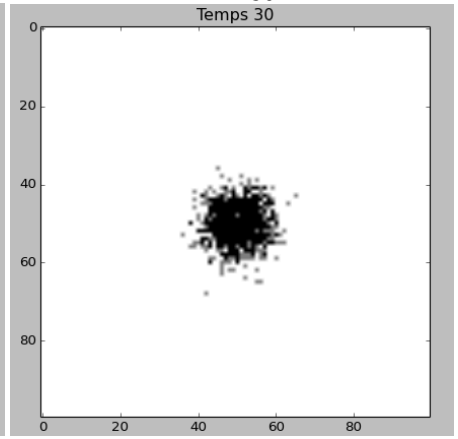
```
def diffusion(T= 100, N = 1000, L = 100, dt = 1e-6):
    M = np.zeros((L,L))
    M[L//2,L//2] = N    # N particules au centre
    # Tracé au temps 0
    plt.figure("Diffusion")
    plt.clf()
```

```
# Tracé en niveau de gris. vmax fixe la valeur du noir.
img = plt.imshow(M, vmax = N/500,/
                        cmap = matplotlib.cm.gray_r)
plt.title('Temps 0')
plt.draw()
plt.pause(dt)
for t in range(T):
    N = np.zeros((L,L))    # Etat au temps t+1
    for i in range(L):
        for j in range(L):
            Nbre = int(M[i,j]) # Nbre particules en (i,j)
            for k in range(Nbre):    # Pour chacune :
                di = randint(-1,1)
                dj = randint(-1,1)
                ii, jj = i+di, j+dj
                if 0 <= ii < L and 0<= jj < L:
                    N[ii,jj] += 1
    M = N
    # Tracé
    #pour la fluidité, appeler 1 seul imshow()
    img.set_data(M)
    titre = 'Temps ' + str(t+1)
    plt.title(titre)
    plt.draw()
    plt.pause(dt)
```

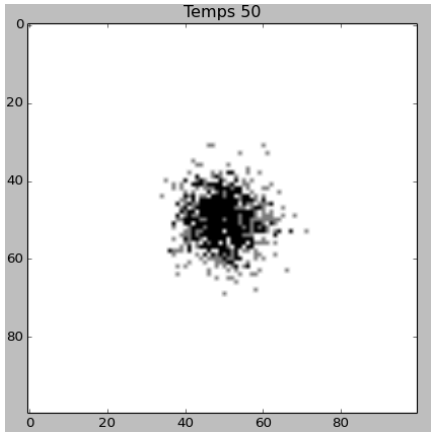
$T = 10$



$T = 30$



$T = 50$



$T = 100$