

## BIO2045 – Séance 2

## Les tâches et les rayures

**Contenu**

Concepts principaux .....	1
Indexation dans les matrices .....	1
Iteration .....	1
Nombres (pseudo)-aléatoires .....	2
Un automate cellulaire pour la pigmentation .....	2
Règles biologiques .....	2
État initial .....	2
Mise à jour de l'activation des cellules .....	2
Résultat final .....	2

**Concepts principaux**

## Indexation dans les matrices

## Iteration

Dans la séance précédente, nous avons utilisé une boucle **for**, qui permettait de répéter un processus plusieurs fois. Dans cette séance, nous allons formaliser ce concept, qui est fondamental pour le reste du cours.

Une boucle **for** est une structure qui s'écrit en général de la manière suivante:

```
for ELEMENT in COLLECTION
  instructions
end
```

La variable **ELEMENT** n'existe pas en dehors de la boucle. C'est une nuance importante: elle est créée par la boucle, et détruite quand la boucle est terminée. Une boucle **for** va simplement prendre

chaque valeur de **COLLECTION**, les stocker dans **ELEMENT**, et on pourra donc appliquer des opérations de manière itérative.

Par exemple, si on veut multiplier par deux tous les éléments du vecteur **[1, 2, 3, 4]**, et afficher le résultat sur une nouvelle ligne avec **println**, on peut utiliser une boucle **for**:

```
for x in [1, 2, 3, 4]
    println(2x)
end
```

```
2
4
6
8
```

## Nombres (pseudo)-aléatoires

```
import Random
Random.seed!(2045)
```

```
Random.TaskLocalRNG()
```

## Un automate cellulaire pour la pigmentation

Règles biologiques

État initial

Mise à jour de l'activation des cellules

Résultat final

```
using CairoMakie
```

Seed

```
"""
    etat_initial(rows, cols, p_activation)

Initialise une grille avec des cellules activées aléatoirement.
```

Arguments:

- `rows::Int`: Nombre de lignes dans la grille
- `cols::Int`: Nombre de colonnes dans la grille
- `p\_activation::Float64`: Probabilité d'activation initiale

Retourne:

- `Array{Bool, 2}`: Grille initialisée

"""

```
function etat_initial(rows, cols, p_activation)
    lattice = zeros{Bool, 2}(rows, cols)
    for row in 1:rows
        for col in 1:cols
            lattice[row, col] = rand() < p_activation
        end
    end
    return lattice
end
```

"""

voisins\_valides(lattice, row, col, rayon)

Retourne les voisins valides d'une cellule dans un certain rayon.

Arguments:

- `lattice::Array{Bool, 2}`: Grille de cellules
- `row::Int`: Ligne de la cellule
- `col::Int`: Colonne de la cellule
- `rayon::Int`: Rayon de recherche des voisins

Retourne:

- `Array{Bool, 2}`: Sous-grille des voisins valides

"""

```
function voisins_valides(lattice, row, col, rayon)
    d_lignes = max(row - rayon, 1)
    f_lignes = min(row + rayon, size(lattice, 1))
    d_colonnes = max(col - rayon, 1)
    f_colonnes = min(col + rayon, size(lattice, 2))
    return lattice[d_lignes:f_lignes, d_colonnes:f_colonnes]
end
```

"""

nombre\_voisins(lattice, row, col, rayon)

Calcule le nombre de voisins d'une cellule dans un certain rayon.

Arguments:

- `lattice::Array{Bool, 2}`: Grille de cellules
- `row::Int`: Ligne de la cellule
- `col::Int`: Colonne de la cellule
- `rayon::Int`: Rayon de recherche des voisins

Retourne:

- `Int`: Nombre de voisins

"""

```
function nombre_voisins(lattice, row, col, rayon)
    voisinage = voisins_valides(lattice, row, col, rayon)
```

```

        n_voisins = count(voisinage)
        return n_voisins
    end

    """
        nouvel_etat(Na, Ni, wa, wi)

    Détermine le nouvel état d'une cellule en fonction du nombre de
    voisins activés et inhibés.

    Arguments:
    - `Na::Int`: Nombre de voisins activés
    - `Ni::Int`: Nombre de voisins inhibés
    - `wa::Float64`: Poids de l'activation
    - `wi::Float64`: Poids de l'inhibition

    Retourne:
    - `Bool`: Nouvel état de la cellule (true pour activé, false
    pour désactivé)
    """
    function nouvel_etat(Na, Ni, wa, wi)
        etat = wa * Na > wi * Ni
        return etat
    end

    """
        afficher_matrice(matrice)

    Affiche une matrice de cellules, où les cellules activées sont
    représentées par '■' et les cellules désactivées par un espace.

    Arguments:
    - `matrice::Array{Bool, 2}`: Matrice de cellules à afficher

    Retourne:
    - Rien
    """
    function afficher_matrice(matrice)
        for i in 1:size(matrice, 1)
            for j in 1:size(matrice, 2)
                if matrice[i, j] == 1
                    print("■")
                else
                    print(" ")
                end
            end
            println()
        end
    end
end

```

```
Main.var"##289".afficher_matrice
```

## Variables

```
wa = 1.0 # Poids de l'activation
wi = 0.12 # Poids de l'inhibition
Ra = 2 # Rayon d'activation
Ri = 9 # Rayon d'inhibition

lignes = 95 # Nombre de lignes dans la grille
colonnes = 65 # Nombre de colonnes dans la grille

p_activation = 0.05 # Probabilité d'activation initiale
temps = 100 # Nombre de générations à simuler
```

100

## Initialisation de la grille

```
lattice = etat_initial(lignes, colonnes, p_activation)
```

[illegible]

[illegible]

[illegible]

[illegible]



[illegible]

[illegible]

Pour chaque génération

```

for gen in 1:temps
    # Grille au temps suivant
    temps_suivant = zeros(Bool, lignes, colonnes)
    # Pour chaque cellule
    for row in 1:lignes
        for col in 1:colonnes
            # Calcul du nombre de voisins activés et inhibés
            activation = nombre_voisins(lattice, row, col, Ra)
            inhibition = nombre_voisins(lattice, row, col, Ri)
            # Détermination du nouvel état de la cellule
            temps_suivant[row, col] = nouvel_etat(activation,
inhibition, wa, wi)
        end
    end
    for i in 1:lignes
        for j in 1:colonnes
            lattice[i, j] = temps_suivant[i, j]
        end
    end
end

end

# Visualisation de type heatmap
heatmap(
    # On passe d'abord l'objet a visualiser
    lattice,
    # Puis on fixe les deux couleurs à blanc et noir
    # pour resp. `false` et `true`
    colormap=[:white, :black],
    # On spécifie que les cellules du heatmap
    # sont des carrés
    axis=(; aspect=DataAspect()),
    # Et on fixe enfin un plus grand nombre de pixels pour avoir
    # une meilleure résolution

```

```

    figure=(; figure_padding=0)
)

# On termine enfin cette figure en retirant les axes et les
# graduations,
# puis en affichant la figure finale
hidespines!(current_axis())
hidedecorations!(current_axis())
current_figure()

```

