

BIO2045 – Séance 2

Les tâches et les rayures

Contenu

Concepts principaux	1
Indexation basique dans les matrices	1
Indexation avancée dans les matrices	3
Iteration	4
Iteration avancée dans les matrices	4
Nombres (pseudo)-aléatoires	7
Fonctions (moins que le minimum nécessaire!)	7
Un automate cellulaire pour la pigmentation	8
État initial	8
Règles biologiques	9
Mise à jour de l'activation des cellules	10
Résultat final	10
Références	13

Concepts principaux

Indexation basique dans les matrices

Dans la séance précédente, nous avons vu comment créer des matrices, et comment lire le contenu à une ligne et colonne particulière. On peut, en pratique, faire beaucoup plus avec des matrices.

Prenons l'exemple de la matrice suivante, avec trois lignes et cinq colonnes, qui contient des nombres aléatoires entiers entre 1 et 5:

```
V = rand(1:5, 3, 5)
```

```
3x5 Matrix{Int64}:
 5  3  4  4  5
 3  2  4  5  4
 4  5  3  5  1
```

On peut accéder à la première ligne de cette matrice avec

```
v[1, :]
```

```
5-element Vector{Int64}:
 5
 3
 4
 4
 5
```

et à sa deuxième colonne avec

```
v[:, 2]
```

```
3-element Vector{Int64}:
 3
 2
 5
```

On peut aussi prendre les deux premières lignes, et les trois dernières colonnes, avec

```
v[begin:(begin+1), (end-2):end]
```

```
2×3 Matrix{Int64}:
 4  4  5
 4  5  4
```

Ce qui est la même chose que

```
v[1:2, 3:5]
```

```
2×3 Matrix{Int64}:
 4  4  5
 4  5  4
```

mais sans avoir besoin d'avoir les coordonnées exactes de la dernière colonne.

Indexation avancée dans les matrices

Les matrices ont toutes un système de coordonnées, qui sont soit les coordonnées Cartésiennes:

```
collect(CartesianIndices(V))
```

```
3×5 Matrix{CartesianIndex{2}}:
 CartesianIndex(1, 1) CartesianIndex(1, 2) CartesianIndex(1,
 3) CartesianIndex(1, 4) CartesianIndex(1, 5)
 CartesianIndex(2, 1) CartesianIndex(2, 2) CartesianIndex(2,
 3) CartesianIndex(2, 4) CartesianIndex(2, 5)
 CartesianIndex(3, 1) CartesianIndex(3, 2) CartesianIndex(3,
 3) CartesianIndex(3, 4) CartesianIndex(3, 5)
```

soit les coordonnées linéaires:

```
collect(LinearIndices(V))
```

```
3×5 Matrix{Int64}:
 1  4  7 10 13
 2  5  8 11 14
 3  6  9 12 15
```

Notez que les coordonnées linéaires suivent les colonnes: Julia est un langage *column-major*, qui va stocker les colonnes ensemble dans la mémoire. Si on veut améliorer la performance de nos simulations, opérer sur les colonnes sera souvent beaucoup plus rapide que d'opérer sur les lignes.

Une caractéristique importante des indices est qu'ils sont *relatifs*. Par exemple, si on veut exprimer la position qui est "la cellule à gauche de la position 3, 4", on peut l'écrire

```
CartesianIndex(3, 4) + CartesianIndex(-1, 0)
```

```
CartesianIndex(2, 4)
```

La position `CartesianIndex(-1, 0)` signifie: une colonne avant, sur la même ligne. Nous allons *beaucoup* utiliser cette propriété pour nous déplacer rapidement dans des 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
```

Iteration avancée dans les matrices

On peut traverser des matrices de façon beaucoup plus efficace en combinant les boucles `for` et les techniques d'indexation. Pour rappel, dans cette section, nous utilisons la matrice suivante:

```
V = rand(1:9, 3, 4)
```

```
3x4 Matrix{Int64}:
 6  9  2  6
 1  1  1  6
 9  9  9  9
```

Par exemple, on peut prendre chaque élément d'une matrice sans devoir spécifier les lignes et les colonnes:

```
for v in V
    println(v)
end
```

```
6
1
9
9
1
9
2
1
9
6
6
9
```

Remarquez que l'ordre des éléments suit ici le **LinearIndex**. On peut aussi aller chercher directement les indices des matrices:

```
for i in eachindex(V)
    println(i)
end
```

```
1
2
3
4
5
6
7
8
9
10
11
12
```

Mais les indices sont eux-même retournés sous forme de matrice. On peut donc itérer sur les indices Cartésiens:

```
for ci in CartesianIndices(V)
    println(ci)
end
```

```

CartesianIndex(1, 1)
CartesianIndex(2, 1)
CartesianIndex(3, 1)
CartesianIndex(1, 2)
CartesianIndex(2, 2)
CartesianIndex(3, 2)
CartesianIndex(1, 3)
CartesianIndex(2, 3)
CartesianIndex(3, 3)
CartesianIndex(1, 4)
CartesianIndex(2, 4)
CartesianIndex(3, 4)

```

Cette structure est particulièrement utile, parce que nous aurons souvent besoin de faire des tâches comme: pour chaque cellule, prendre la cellule du dessus, et si cette cellule est dans la matrice, effectuer une opération sur sa valeur.

```

for position in CartesianIndices(V)
    dessous = position + CartesianIndex(0, -1)
    if dessous in CartesianIndices(V)
        println(dessous)
    end
end

```

```

CartesianIndex(1, 1)
CartesianIndex(2, 1)
CartesianIndex(3, 1)
CartesianIndex(1, 2)
CartesianIndex(2, 2)
CartesianIndex(3, 2)
CartesianIndex(1, 3)
CartesianIndex(2, 3)
CartesianIndex(3, 3)

```

On utilise ici la structure `if un truc in plusieurs trucs`, qui renvoie `true` si l'élément `un truc` fait partie de la collection `plusieurs trucs`.

On peut enfin itérer d'une manière qui nous renvoie à la fois la position et la valeur:

```

for (position, valeur) in enumerate(V)
    println("La position $position contient la valeur $valeur")
end

```

```

La position 1 contient la valeur 6
La position 2 contient la valeur 1

```

```

La position 3 contient la valeur 9
La position 4 contient la valeur 9
La position 5 contient la valeur 1
La position 6 contient la valeur 9
La position 7 contient la valeur 2
La position 8 contient la valeur 1
La position 9 contient la valeur 9
La position 10 contient la valeur 6
La position 11 contient la valeur 6
La position 12 contient la valeur 9

```

Nombres (pseudo)-aléatoires

```

import Random
Random.seed!(2045)

```

```
Random.TaskLocalRNG()
```

Fonctions (moins que le minimum nécessaire!)

```

function operation(entree1, entree2)
    resultat = entree1 + entree2
    return resultat
end

```

```
operation (generic function with 1 method)
```

```
operation(1, 2)
```

```
3
```

```
operation(3.0, 2.5)
```

```
5.5
```

Au cours de la session, nous allons *considérablement* complexifier les tâches que l'on peut faire en déclarant des fonctions, en introduisant notamment des valeurs par défaut, des mot-clés, puis enfin des restrictions sur le type des entrées. Pour cette séance, cette compréhension de base est suffisante.

Un automate cellulaire pour la pigmentation

Avec cette simulation, nous voulons observer la pigmentation d'un tissu en utilisant une série de règles simples qui vont approximer un modèle dit de réaction/diffusion. Ces modèles ont été introduits par Alan Turing dans les années 1950 [1].

Le modèle d'origine utilise des équations différentielles partielles pour modéliser la diffusion, mais on peut approximer les mêmes mécanismes avec un automate cellulaire. Nous allons d'abord définir le problème et son état initial, puis introduire les différents règles.

Ce modèle représente un tissu (la peau ou le pelage d'un animal) comme un espace en deux dimensions, dans lequel chaque position (cellule dans une lattice) est soit pigmentée (**true**), soit non-pigmentée (**false**).

État initial

Le tissu a une dimension qui reste fixe pendant toute la simulation. Notez qu'ici on définit deux variables sur la même ligne. C'est un raccourci d'écriture qui n'est pas nécessaire.

```
lignes, colonnes = 205, 155
```

```
(205, 155)
```

On définit ensuite une probabilité que les cellules soient initialement pigmentées. Puisque c'est une probabilité, ce nombre devrait être entre 0 et 1.

```
p_activation = 0.01
```

```
0.01
```

Pour l'état initial, on va devoir parcourir une grille de taille **lignes, colonnes**, et pour chaque cellule, lui donner la valeur qui correspond à la pigmentation (**true**) avec une probabilité **p_activation**.


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

```
etat_initial (generic function with 1 method)
```

Cette fonction utilise `rand()`, qui par défaut génère un nombre aléatoire entre 0 et 1, avec une distribution uniforme.

Avec ces informations, on peut maintenant créer notre lattice:

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

On n'affiche pas cette lattice, qui peut être très grande.

Ici, on choisit d'appeler cet objet `lattice`, puisque c'est ce qu'il représente. Mais on peut donner un nom plus explicite à cet objet, comme par exemple `pelage`, ou encore 🐎. Julia accepte la majorité des [symboles unicode](#). Il se peut que votre police de caractère ne les affiche pas tous — celles qui ont le plus de support sont [Iosevka](#) (ma préférée!), [JuliaMono](#), et dans une moindre mesure, [Noto Sans Mono](#). Elles sont toutes gratuites.

Règles biologiques

Dans notre simulation du modèle de réaction/diffusion, une cellule va s'activer si le signal qui encourage son activation est plus grand que le signal qui encourage sa désactivation. Ces deux signaux se calculent de la même façon: le nombre de voisins actifs, multiplié par le poids du signal d'activation.

Autrement dit, une cellule se pigmente si $w_a N_a > w_i N_i$, avec w_a et w_i les poids de l'activation et de l'inhibition, et N_a et N_i le nombre de cellules voisines qui sont activées et inhibées.

Ce modèle représente une situation dans laquelle une cellule est activée en réponse à la diffusion de deux substances: les cellules activées diffusent une substance activatrice, et les cellules inhibées diffusent une substance inhibitrice. Les poids w_a et w_i mesurent

l'affinité des cellules pour ces substances, et on peut modifier le rayon de diffusion des substances en calculant le nombre de voisins dans un voisinage toujours plus grand.

Mise à jour de l'activation des cellules

Résultat final

```
using CairoMakie
```

Seed

```
"""
    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
```

```

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"##465".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
```

9

stuff i guess

```
temps = 100 # Nombre de générations à simuler
```

100

Initialisation de la grille

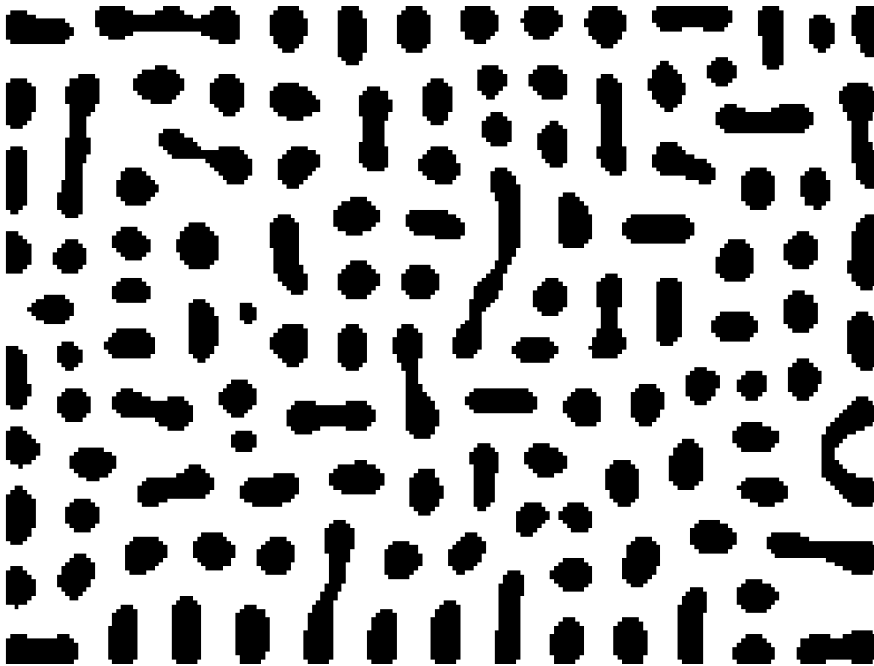
Pour chaque génération

```
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

# Visualisation de type heatmap
heatmap(
    # On passe d'abord l'objet à 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()
```



Références

- [1] A. M. Turing, The Chemical Basis of Morphogenesis, Philosophical Transactions of the Royal Society of London. B, Biological Sciences **237**, 37 (1952).