BIO2045 – Séance 2

Les tâches et les rayures

Contenu

Concepts principaux

Nombres (pseudo)-aléatoires

Quand on génère des nombres aléatoires, avec les fonctions comme `rand`, les résultats sont différents à chaque fois. Pour faciliter la comparaison des résultats, il est utile d'utiliser un `seed` , qui représente le point de départ de l'algoritme des nombres (pseudo)-aléatoires.

Cette initialisation se fait de la façon suivante:

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

```
Random.TaskLocalRNG()
```

Les nombres qui sont générés après cette commande seront toujours les mêmes, mais vont respecter les propriétés des nombres aléatoires. La bonne pratique est d'utiliser cette commande après avoir chargé les packages, et avant le code.

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)
```

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

On peut accéder à la première ligne de cette matrice avec

```
V[1, :]
```

```
5-element Vector{Int64}:
 2
 2
 4
 3
 3
```

et à sa deuxième colonne avec

```
V[:, 2]
```

```
3-element Vector{Int64}:
 2
 4
 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  3  3
 4  2  2
```

Ce qui est la même chose que

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

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

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

Indexation avancée dans les matrices

Les matrices ont toutes un système de coordonées, qui sont soit les coordonées 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 coordonées 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 coordonées 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 avions 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)
```

```
3×4 Matrix{Int64}:
 2  5  8  7
 4  7  6  3
 3  2  5  5
```

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
```

```
2
4
3
5
7
2
8
6
5
7
3
5
```

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

Fonctions (moins que le minimum nécessaire!)

Lors de la dernière séance, nous avions mis des instructions dans une boucle `for`. Pour rendre le code plus lisible, il est souvent pertinent de regrouper des opérations similaires dans des fonctions. Dans les séances qui viennent, nous allons introduire des façons plus complexes (et flexibles!) de définir des fonctions. Pour le moment nous allons utiliser la syntaxe suivante:

```
fonction nom(arg1, arg2, arg3)
  [operations sur arg1, arg2, arg3]
  return resultat
end
```

Par example, on peut définir une fonction qui additione ses deux arguments avec:

```
function addition(entree1, entree2)
    resultat = entree1 + entree2
    return resultat
end
```

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

```
addition(1, 2)
```

```
3
```

```
addition(3.0, 2.5)
```

```
5.5
```

Comme avec les boucles, les variables qui sont déclarées dans la fonction n'existent *que* dans la fonction.

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 même 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'appeller cet object `lattice`, puisque c'est ce qu'il représente. Mais on peut donner un nom plus explicite à cet object, 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 (utilisée dans ces notes de cours), et dans une moindre mesure, Noto Sans Mono et JetBrains 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 à la fois une substance activatrice et 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.

Identification des voisins

Nous allons devoir calculer les voisins qui sont actifs très souvents, et donc c'est une bonne occasion de créer une fonction pour localiser ces voisins. Nous allons supposer que nous cherchons des voisins qui sont dans un cercle d'un diamètre donné.

Un point est dans un cercle si la distance Euclidienne entre ce point et le centre du cercle est inférieure ou égale au rayon du cercle. Nous allons pouvoir identifier les points via leurs `CartesianIndex`. Par exemple, le point $(1, 1)$ est à une distance ≈ 1.41 du centre du plan:

```
sqrt(sum(Tuple(CartesianIndex(0, 0) - CartesianIndex(1, 1)) .^
2))
```

```
1.4142135623730951
```

NB: l'appel à `Tuple` est nécessaire ici. C'est comme ça. Tout n'a pas toujours une explication satisfaisante.

```
function point_dans_cercle(centre, point, rayon)
    dᵢⱼ = sqrt(sum(Tuple(centre - point) .^ 2))
    return dᵢⱼ <= rayon
end
```

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

Cette fonction va donc nous permettre, pour chaque point de la matrice, de trouver ses voisins. Par exemple, si on prend la position `[10, 15]` et qu'on cherche les points dans un rayon de 2 cellules, on obtient:

```
for cellule in CartesianIndices(lattice)
    if point_dans_cercle(CartesianIndex(3, 5), cellule, 2)
        println(cellule)
    end
end
```

```
CartesianIndex(3, 3)
CartesianIndex(2, 4)
CartesianIndex(3, 4)
CartesianIndex(4, 4)
CartesianIndex(1, 5)
CartesianIndex(2, 5)
CartesianIndex(3, 5)
CartesianIndex(4, 5)
CartesianIndex(5, 5)
CartesianIndex(2, 6)
CartesianIndex(3, 6)
CartesianIndex(4, 6)
CartesianIndex(3, 7)
```

Mais en pratique, on veut faire une tâche un peu plus compliquée: pour chaque cellule, on souhaite compter ses voisins actifs, dans deux rayons possiblement différents. On va donc commencer par faire une boucle sur chaque cellule:

```
for cellule in eachindex(lattice)
  1) identifier tous les voisins
  2) identifier ceux qui sont actifs
end
```

Il existe une fonction, `findall`, qui permet d'identifier toutes les positions d'une collection qui correspondent à un critère donné. Si on donne uniquement une collection comme argument, `findall` renvoie les positions pour lesquelle cette collection a la valeur `true`. On peut donc économiser du temps, en limitant notre recherche aux cellules actives:

```
centre = CartesianIndex(10, 12)
for cellule_active in findall(lattice)
    if point_dans_cercle(centre, cellule_active, 3)
        println(cellule_active)
    end
end
```

On peut donc maintenant calculer le nombre de voisins:

```
function nombre_de_voisins(cellule, cellules_actives,
rayon_activation, rayon_inhibition)
    na = 0
    ni = 0
    for voisin in cellules_actives
        if point_dans_cercle(cellule, voisin, rayon_activation)
            na += 1
        end
        if point_dans_cercle(cellule, voisin, rayon_inhibition)
            ni += 1
        end
    end
    return (na, ni)
end
```

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

Cette approche du problème est assez lente, mais notre objectif pour cette séance est de décomposer le problème au maximum. Dans la pratique, on peut employer plusieurs méthodes pour (i) identifier une région carrée de la matrice dans laquelle les cellules pertinentes sont contenues, et (ii) identifier lequel des deux rayons est le plus grand pour ne vérifier le second rayon que si c'est pertinent. Ces aéméliorations rendraient la fonction beaucoup plus rapide, mais aussi plus longue.

```
nombre_de_voisins(CartesianIndex(10, 12), findall(lattice), 12,
15)
```

```
(4, 5)
```

Mise à jour de l'activation des cellules

Pour changer l'activation des cellules, il faut calculer le nouvel état de chaque cellule, sans interférer avec l'état actuel. Nous allons donc devoir créer une nouvelle lattice pour la génération suivante:

```
prochaine_lattice = zeros(Bool, size(lattice));
```

Remarquez qu'on utilise `size(lattice)` pour que la lattice aie la même taille.

On définit maintenant les paramètres de la simulation:

```
wa = 0.25   # Poids de l'activation
wi = 1.0  # Poids de l'inhibition
Ra = 3.5  # Rayon d'activation
Ri = 6  # Rayon d'inhibition
```

```
6
```

Puis on simule la première génération:

```
cellules_actives = findall(lattice)

for cellule in CartesianIndices(lattice)
    na, ni = nombre_de_voisins(cellule, cellules_actives, Ra,
Ri)
    prochaine_lattice[cellule] = wa * na > wi * ni
end
```

Simulation

Il faut maintenant répéter cette opération plusieurs fois:

```
temps = 5  # Nombre de générations à simuler
```

```
5
```

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

```
for generation in 1:temps
    cellules_actives = findall(lattice)
    prochaine_lattice = zeros(Bool, size(lattice))
    for cellule in CartesianIndices(lattice)
        ni, na = nombre_de_voisins(cellule, cellules_actives,
Ra, Ri)
        prochaine_lattice[cellule] = wa * na > wi * ni
    end
    for cellule in CartesianIndices(lattice)
        lattice[cellule] = prochaine_lattice[cellule]
    end
end
```

```
┌ Warning: Assignment to `cellules_actives` in soft scope is
ambiguous because a global variable by the same name exists:
`cellules_actives` will be treated as a new local. Disambiguate
by using `local cellules_actives` to suppress this warning or
`global cellules_actives` to assign to the existing global
variable.
└ @ ~/Documents/SimulerLeVivant/output/02_Salamandre.md:2
┌ Warning: Assignment to `prochaine_lattice` in soft scope is
ambiguous because a global variable by the same name exists:
`prochaine_lattice` will be treated as a new local. Disambiguate
by using `local prochaine_lattice` to suppress this warning or
`global prochaine_lattice` to assign to the existing global
variable.
└ @ ~/Documents/SimulerLeVivant/output/02_Salamandre.md:3
```

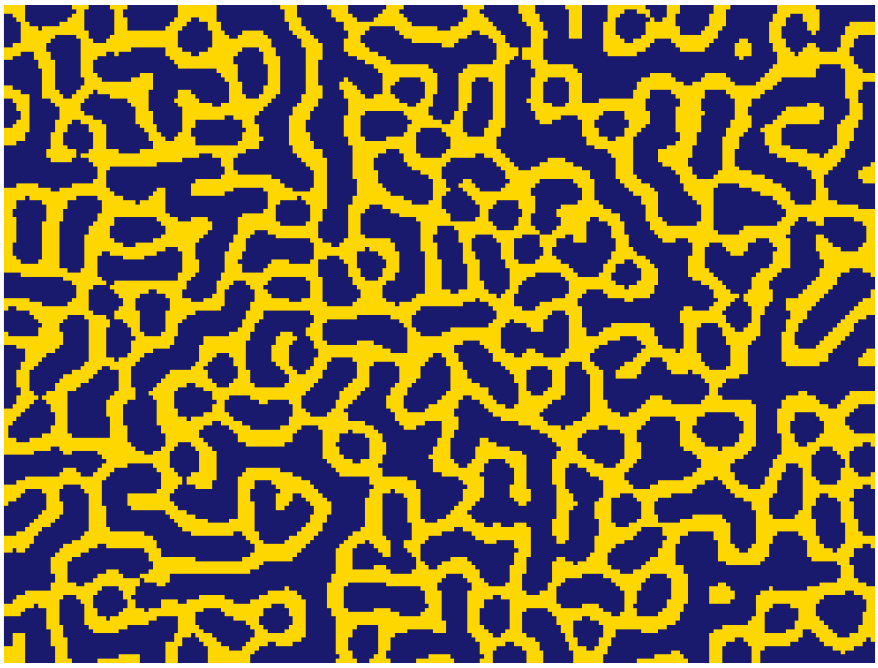
On peut enfin visualiser le résultat:

```
using CairoMakie
```

```
heatmap(
    # On passe d'abord l'objet a visualiser
    lattice,
    # Puis on fixe les deux couleurs
    # pour resp. `false` et `true`
    colormap=[:midnightblue, :gold],
    colorrange = (0, 1),
    # On spécifie que les cellules du heatmap
    # sont des carrés
    axis=(; aspect=DataAspect()),
    # Et on fixe enfin un plus grand nombre de pixles 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()
```



Est-ce que ce résultat ressemble à des motifs qu'on peut observer dans la nature? Que pensez-vous que changer le rayon, ou la force de l'activation ferait sur la coloration de cette surface? Est-ce que ça pourrait avoir un effet sur la forme des motifs?

Bibliographie

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