BIO2045 – Séance 3

Génétique des populations

Contenu

Concepts principaux

Fonctions anonymes

Dans la séance précédente, nous avons commencé a écrire des fonctions. Dans certaines situations, il est nécessaire d'utiliser des fonctions, mais puisqu'elles sont très simples et à usage unique, on dispose d'une syntaxe simplifiée, `argument -> operation`:

```
x -> 2*x + 1
```

```
#2 (generic function with 1 method)
```

Le symbole `->` est simplement `-` puis `>`.

Ces fonctions peuvent s'utiliser comme des fonctions régulières:

```
(x -> 2*x + 1)(3)
```

```
7
```

Mapping et slicing

On peut automatiser certaines opérations *via* la fonction `map`:

```
import Random
Random.seed!(2045)
x = rand(1:5, 10)
```

```
10-element Vector{Int64}:
 2
 5
 1
 2
 4
 5
 4
 4
 4
 3
```

```
map(v -> sqrt(v + 1), x)
```

```
10-element Vector{Float64}:
 1.7320508075688772
 2.449489742783178
 1.4142135623730951
 1.7320508075688772
 2.23606797749979
 2.449489742783178
 2.23606797749979
 2.23606797749979
 2.23606797749979
 2.0
```

Cette syntaxe est équivalente à l'utilisation d'une boucle `for`: on applique la fonction anonyme `x -> sqrt(x + 1)` à chaque élément du vecteur.

On peut aussi appliquer la fonction `map` sur certaines dimensions d'un objet, *via* `mapslices`. Par exemple:

```
V = rand(1:10, 3, 4)
```

```
3×4 Matrix{Int64}:
 4  3  4  8
 5  7  3  3
 5  3  5  9
```

```
mapslices(x -> minimum(x) % 2 == 0, V, dims=1)
```

```
1×4 Matrix{Bool}:
 1  0  0  0
```

Cette fonction va appliquer la fonction `minimum(x) % 2 == 0`, qui renvoie `true` si la plus petite valeur de la colonne est un nombre pair, et `false` sinon, pour chaque *colonne* de la matrice.

On peut faire la même chose pour les lignes:

```
mapslices(x -> minimum(x) % 2 == 0, V, dims=2)
```

```
3×1 Matrix{Bool}:
 0
 0
 0
```

Deux choses sont importantes ici:

D'abord, la dimension 1 correspond aux *colonnes*, et pas aux *lignes*. C'est parce que Julia stocke les colonnes en premier dans la mémoire, et non les lignes.

Enfin, l'objet est retourné sous forme de *matrice*, parce que l'objet donné en agument est une matrice.

On peut éliminer les dimensions qui ne sont plus nécessaires pour obtenir un vecteur:

```
v = mapslices(x -> minimum(x) % 2 == 0, V, dims=2)
dropdims(v, dims=2)
```

```
3-element Vector{Bool}:
 0
 0
 0
```

Notez qu'on peut utiliser `map` et `mapslices` avec le nom d'une fonction qui accepte un unique argument:

```
mapslices(sum, V, dims=1)
```

```
1×4 Matrix{Int64}:
 14  13  12  20
```

Filtres

Dans plusieurs situations, on va devoir identifier des éléments d'une collection qui répondent à certains critères. On peut effectuer ce travail avec `filter`, qui ne renvoie que les éléments pour lesquels la fonction donnée en argument vaut `true`.

```
x = rand(1:5, 10)
```

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

On peut par exemple choisir uniquement les éléments compris entre 4 et 6 avec:

```
filter(v -> 4 <= v <= 6, x)
```

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

Les fonctions `map` et `filter` se combinent très bien. On peut par exemple créer un vecteur qui contient les racines carrées des nombres pairs contenus dans `x`:

```
map(sqrt, filter(v -> v % 2 == 0, x))
```

```
3-element Vector{Float64}:
 2.0
 2.0
 1.4142135623730951
```

Arrays à plusieurs dimensions

On peut créer des tableaux (`Array`) avec plus de deux dimensions. Par exemple, cet objet est un “cube” avec quatre lignes, trois colonnes, et deux “tranches” en profondeur:

```
T = rand(1:9, (4, 3, 2))
```

```
4×3×2 Array{Int64, 3}:
[:, :, 1] =
 4  6  8
 6  4  6
 2  6  7
 9  9  8

[:, :, 2] =
 6  2  1
 5  6  1
 2  9  2
 7  2  5
```

On peut par exemple vérifier si la valeur minimum de chaque ligne est comprise entre 2 et 4 avec:

```
mapslices(x -> 2 <= minimum(x) <= 4, T, dims=2)
```

```
4×1×2 Array{Bool, 3}:
[:, :, 1] =
 1
 1
 1
 0

[:, :, 2] =
 0
 0
 1
 1
```

Simulation: maintien du polymorphisme

Nous allons simuler une population de cellules avec un génome très simple, composé de trois gènes: R (rouge), V (vert), et B (bleu). Ces gènes ont deux allèles: allumé ou éteint. La combinaison de ces trois gènes détermine la couleur de la cellule. Par exemple, une cellule avec le génome `101` (R et B sont actifs) aura la couleur suivante:

```
using CairoMakie
CairoMakie.activate!(px_per_unit=2.0)
CairoMakie.Colors.RGB(1, 0, 1)
```

À chaque génération, les cellules se reproduisent: chaque position dans l'espace va choisir deux parents proches, au hasard, avec une distance maximale que l'on peut fixer. Le descendant de ces deux parents portera un mélange des génômes des deux parents. On suppose que les trois gènes vont se transmettre de manière indépendante.

Après la reproduction, chaque position du génome des descendants peut subir une mutation aléatoire, avec un taux faible.

Pour sélectionner des parents au hasard, nous aurons besoin de la fonction `StatsBase.sample`:

```
import StatsBase
```

L'environnement des cellules a une seule dimension, et on veut visualiser le changement de phénotype au cours du temps. On suppose que la taille de la population est constante: les positions sur la ligne qui représente la population sont toutes toujours occupées.

Choix des paramètres initiaux

```
cells = 100
generations = 501
mutation = 1e-4
parents_distance = 3
```

```
3
```

État initial

```
lattice = zeros(Bool, (cells, generations, 3))
for i in Base.OneTo(cells)
    lattice[i,1,:] = rand(Bool, 3)
end
```

Première génération

```
for generation in 2:generations
    for i in Base.OneTo(cells)
        parents_possibles = filter(p -> 1 <= p <= cells, (i-
parents_distance):(i+parents_distance))
        parents = StatsBase.sample(parents_possibles, 2,
replace=false)
        for gene in 1:3
            lattice[i,generation,gene] =
lattice[rand(parents),generation-1,gene]
            if rand() <= mutation
                lattice[i,generation,gene] = !
lattice[i,generation,gene]
            end
        end
    end
end
```

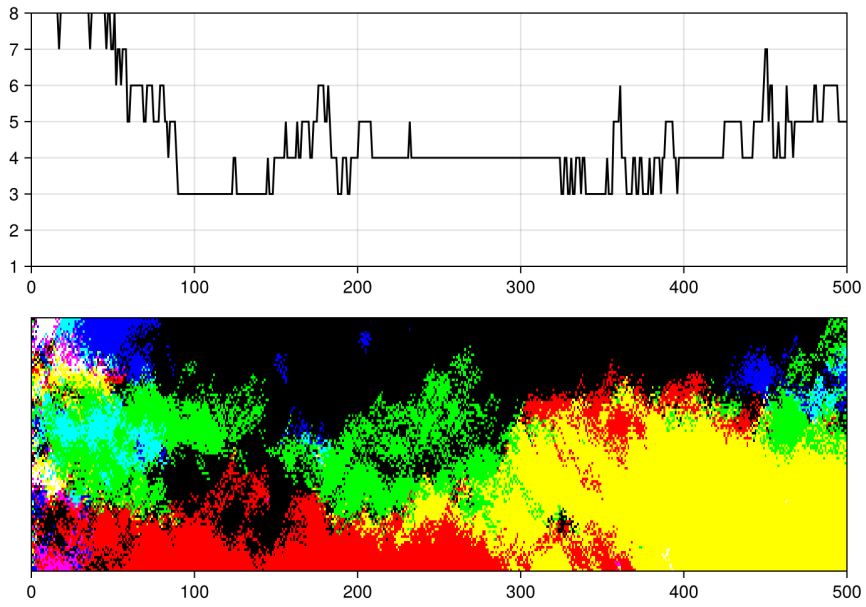
Fonction pour les couleurs

```
colormap = dropdims(mapslices(x -> CairoMakie.Colors.RGB(x...),
lattice, dims=3), dims=3);
```

Heatmap et diversité

```
f = Figure(; size=(700, 500))
ax = Axis(f[2,1])
heatmap!(ax, 0:(generations-1), 1:cells, permutedims(colormap))
hideydecorations!(ax)
plax = Axis(f[1,1])
lines!(plax, 0:(generations-1), vec(mapslices(x ->
```

```
length(unique(x)), colormap, dims=1)), color=:black)
ylims!(plax, 1, 8)
xlims!(plax, 0, generations-1)
xlims!(ax, 0, generations-1)
f
```



Suggestions pour le premier devoir

Paysage circulaire

Sélection génétique

Déséquilibre de liaison

Traits quantitatifs