BIO2045 – Séance 1

La pigmentation de *Conus textile*

Contenu

Concepts principaux

Valeurs Booléennes

Les valeurs Booléennes représentent les états "vrai" et "faux", et sont particulièrement importantes pour nous: elles vont intervenir dans la majorité des sénces. De manière générale, beaucoup de décisions que nous allons prendre seront *in fine* des questions dont la réponse est "oui" ou "non", et les valeurs Booléennes sont appropriés dans ce contexte.

La première valeur est "vrai":

```
true
```

```
true
```

et la seconde est “faux”:

```
false
```

```
false
```

Nous pouvons combiner ces valeurs via différentes opérations.

Opérations sur les valeurs Booléennes

Les valeurs Booléennes ont leurs propres opérations. Ces opérations ont la propriété de prendre comme entrée une où plusieurs valeurs Booléenes, et de retourner une réponse Booléenne.

La première est le `or`, qui renvoie vrai *ssi* au moins une de ses entrées est vrai. Elle est représentée par la barre verticale:

```
true | true
```

```
true
```

```
true | false
```

```
true
```

```
false | false
```

```
false
```

La seconde opération importante est `and`, qui renvoie vrai *ssi* ses deux entrées sont vraies. Elle est représentée par le signe `&`:

```
true & true
```

```
true
```

```
true & false
```

```
false
```

```
false & false
```

```
false
```

On peut aussi prendre la *négation* d'une valeur Booléenne avec l'opérateur `not`, qui est en général représenté par `!`, mais parfois aussi par `~`:

```
!true
```

```
false
```

```
!false
```

```
true
```

Le dernier opérateur Booléen est le `xor` ("où exclusif"), qui renvoie vrai uniquement *ssi* l'opération `or` appliquée à des deux entrées revnoie vrai *et* que l'opération `and` renvoie faux. Il est representé par le signe `⊻`, qui s'écrit `\xor<Tab>`

```
true ⊻ false
```

```
true
```

```
false ⊻ false
```

```
false
```

```
true ⊻ true
```

```
false
```

Ces opérateurs peuvent être utilisés pour prendre des décisions complexes. Par exemple, `⊻` est défini, pour deux entrées x_1 et x_2, comme `(x₁ | x₂) & (!(x₁ & x₂))`.

Vecteurs et matrices

Une des tâches les plus courantes que nous devrons réaliser est de stocker de l'information dans des structures avec plusieurs dimensions. Autant que possible, nous essaierons de connaître les dimensions de ces objets avant de les créer.

Un objet à une seule dimension est un vecteur, et on peut en créer un avec la commande

```
zeros(5)
```

```
5-element Vector{Float64}:
 0.0
 0.0
 0.0
 0.0
 0.0
```

qui se lit "un vecteur de cinq positions initialement rempli de zéros". Par défaut, ce vecteur pourra stocker des *nombres* (nous reviendrons sur la définition d'un nombre plus tard), mais on peut créer un vecteur qui contient des valeurs Booléennes:

```
zeros(Bool, 3)
```

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

NB: même si le résultat est affiché avec des `0` et des `1`, il s'agit bien de valeurs Booléennes; pour gagner de la place, `true` est en général remplacé par `1` et `false` par `0`.

On peut aussi créer des objets avec plus d'une dimension, comme des matrices (deux dimensions), des tenseurs (trois dimensions), etc.. Par exemple, cette commande crée une matrice initialement rempli de valeurs Booléennes aléatoires, avec 3 lignes et 2 colonnes:

```
rand(Bool, 3, 2)
```

```
3×2 Matrix{Bool}:
 1  1
 1  1
 1  0
```

NB: regardez la documentation des fonctions `rand` et `ones`.

Au cours de la session, nous allons identifier des façons différentes de naviguer dans ces objets. Pour le moment, nous allons nous contenter de trouver et de modifier le contenu de ces objets en utilisant les coordonées.

Pour un vecteur, on peut extraire l'information en utilisant le numéro de la position. Notez que les vecteurs dans Julia sont des colonnes (pour faciliter les opérations d'algèbre linéaire):

```
V = zeros(Bool, 3)
V[1]
```

```
false
```

On peut modifier la deuxième position de ce vecteur:

```
V[2] = true
V
```

```
3-element Vector{Bool}:
 0
 1
 0
```

Il existe aussi les raccourcies `begin` (premier élément) et `end` (dernier élément):

```
V[begin] = true
V
```

```
3-element Vector{Bool}:
 1
```

```
1
0
```

Pour une matrice, l'indexation se fait exactement de la même manière, mais on utilise les coordonées sous la forme ligne, colonne:

```
M = zeros(Bool, 2, 3)
M[1, 2] = true
M[2, 3] = true
M
```

```
2×3 Matrix{Bool}:
 0  1  0
 0  0  1
```

Installer des packages

Les *packages* contiennent des fonctionnalités qui ne sont pas présentes dans le langage par défaut. Certains de ces packages (comme `Statistics` et `Random`) font partie de la bibliothèque standard (*standard library*) du langage, mais d'autres doivent être installés pour les utiliser.

Les projets

Julia stocke ses packages d'une manière qui diffère de plusieurs autres langages de programmation. L'information sur la liste et les des packages utilisée est stockée localement dans un fichier `Project.toml`, et les versions complètes de toutes les dépendences sont stockées dans un fichier `Manifest.toml`. Ces deux fichiers sont en général créés automatiquement.

Pour cette raison, il est *indispensable* de créer un environnement, qui sera à la racine du projet. Dans le cadre de ce cours, vous pouvez créer *un seul* environnement pour l'ensemble des séances.

```
import Pkg
Pkg.activate(".")
```

La documentation du gestionnaire de packages est disponible en ligne: https://pkgdocs.julialang.org/v1/ – il est important de la consulter.

Il faut activer cet environnement avec la même syntaxe avant d'éxécuter du code. Dans VSCode, cette activation se fait automatiquement.

NB: on peut aussi activer le mode `pkg` avec la touche `]` - la documentation du package manager explique comment.

Installer un package

On peut installer un package avec la commande `add`. Par exemple, cette commande va installer le package `CairoMakie`, que nous allons utiliser pour la visualisation.

```
Pkg.add("CairoMakie")
```

La documentation de Makie est disponible en ligne: https://docs.makie.org/stable/

L'étape d'installation des packages peut être *très* longue, parce que Julia va les compiler avant leur utilisation. Une fois que les packages sont compilés, leur chargement est beaucoup plus rapide.

Charger un package

Il faut importer explicitement les packages pour pouvoir les utiliser:

```
using CairoMakie
```

Simulation: *Conus textile*

Nous allons simuler la pigmentation de la coquille de *Conus textile* en utilisant un automate cellulaire très-simple, qui est en général appelé *Rule 30*.

Rule 30 est un automate cellulaire qui fonctionne sur une seule dimension. À chaque génération, une cellule q change d'état selon l'état de ses deux voisins p et r, et de son propre état.

Pour trouver l'état de la cellule au temps suivant, on s'intéresse à la séquence de valeurs Booléennes qui représente p, q, et r. Si le triplet p, q, r vaut 100, 011, 010, ou 001, la cellule est active. Sinon, la cellule devient inactive.

NB: les transitions dans cette famille d'automates cellulaires sont toujours listées dans le même ordre: 111, 110, 101, 100, 011, 010, 00, et 000. La séquence des états qui correspond, 00011110, est la représentation binaire du nombre 30, ce qui donne son nom à la rêgle.

Rule 30 peut se représenter de manière beaucoup plus simple, avec la formule suivante: `p ⊻ (q | r)`. On peut donc simuler l'évolution de cette rêgle dans le temps avec une expression beaucoup plus simple.

NB: vous pouvez aussi essayer de ré-écrire ce code en utilisant la séquen de *p*, *q*, et *r*, ce qui permettra de remplaer *Rule 30* par *Rule 90* (01011010), qui donne un résultat similaire.

Définir les paramètres de la simulation

Pour effectuer la simulation, nous allons commencer par choisir un nombre de cellules: cette quantité est une *variable*, `n_cellules`, dont la *valeur* est fixée avant le départ de la simulation:

```
n_cellules = 191
```

```
191
```

Nous allons ensuite identifier le nombre de générations qu'il faut simuler. Puisque notre modèle simule de la croissance, qui va être symétrique, on veut arrêter la simulation à la génération pour laquelle toutes les cellules auront été activées:

```
n_generations = Int((n_cellules - 1) / 2)
```

```
95
```

Quand on assigne cette variable, on oblige le nombre qu'elle contient à être un nombre entier (`Int`). Nous reviendrons plus tard sur les différents types de nombres.

Une fois les deux paramètres connus, nous pouvons créer un objet qui va représenter la coquille du coquillage. Spécifiquement, nous allons stocker l'état des cellules dans une matrice, qui aura une ligne par cellule, et une colonne par génération de division cellulaire:

```
shell = zeros(Bool, n_cellules, n_generations);
```

Comme on sait que l'état de nos cellules est représenté par une variable Booléenne (pigmenté est `true`, non-pigmenté est `false`), on a spécifié que cet objet contenait des variables Booléennes.

Notez qu'on a ajouté le `;` à la fin de la ligne: cela permet de ne pas afficher le résultat de l'opération, ce qui est en général une bonne pratique si les objets sont très grands.

Conditions initiales

On doit maintenant définir l'état de la première génération. Nous allons partir avec une seule cellule, qui est initialement pigmentée (`true`), et qui se situe à la position du milieu:

```
milieu_index = div(n_cellules, 2) + 1
```

```
96
```

On utilise ici la fonction `div` – sa documentation est accessible avec `?div`.

Un fois que la position initiale est identifiée, on peut modifier l'état de cette cellule. Puisque notre coquille est représentée par une matrice, il faut indiquer la ligne (la cellule) et la colonne (la génération):

```
shell[milieu_index, 1] = true;
```

Effectuer la simulation

On va maintenant effectuer la simulation. Pour cette simulation, nous allons utiliser une boucle avec `for`. Nous passerons plus de temps pendant les prochaines séances sur les différents types de boucles.

```
# On commence a la génération 2, parce que la génération
# 1 est la génération initiale!
for generation in 2:n_generations

    # On ne met à jour que les cellules 2 jusqu'à n-1, pour
    # éviter les effets de bord.
```

```
    for cellule in 2:n_cellules-1

        # La cellule p est la cellule à gauche, et on veut
        # son état dans la génération précédente
        p = shell[cellule-1, generation-1]

        # Même logique pour q et r
        q = shell[cellule, generation-1]
        r = shell[cellule+1, generation-1]

        # Règle de transition pour la Rule 30 des automates
        # cellulaires : p xor (q or r)
        shell[cellule, generation] = p ⊻ (q || r)
    end

end
```

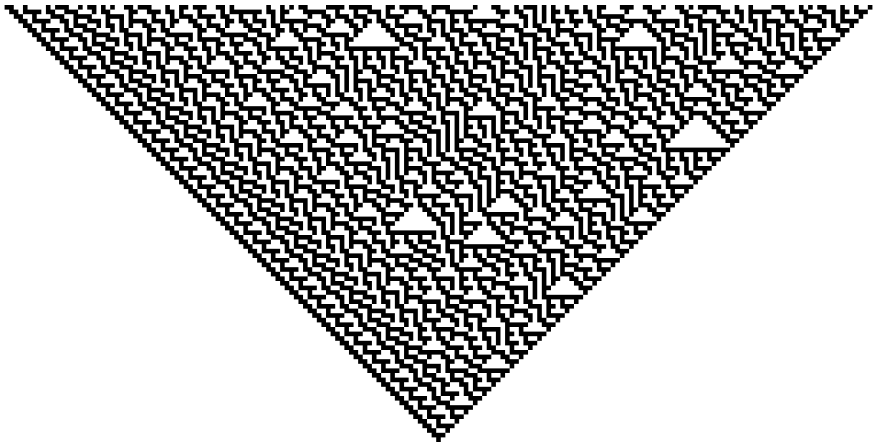
La simulation est maintenant termineé. Notez qu'ici rien n'est affiché, parce que nous n'avons pas explicitement demandé de voir un objet.

Afficher l'état final de la simulation

On va utiliser le package `CairoMakie` pour visualiser la simulation.

```
# Visualisation de type heatmap
heatmap(
    # On passe d'abord l'objet a visualiser
    shell,
    # 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 pixles pour avoir
    # une meilleure résolution
    figure=(; size=(3n_cellules, 3n_generations),
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()
```



Comparez cette simulation a des images de *Conus textile*. En imaginant cette surface enroulée autour d'un axe, est-ce que notre simulation a produit une bonne approximation de la pigmentation de la coquille?