

## BIO2045 – Séance 1

La pigmentation de *Conus textile***Contenu**

Concepts principaux .....	1
Valeurs Booléennes .....	1
Opérations sur les valeurs Booléennes .....	2
Vecteurs et matrices .....	4
Installer des packages .....	6
Les projets .....	6
Installer un package .....	7
Charger un package .....	7
Simulation: <i>Conus textile</i> .....	7
Définir les paramètres de la simulation .....	8
Effectuer la simulation .....	8
Afficher l'état final de la simulation .....	9

**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 séquences. 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 appropriées 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 ou plusieurs valeurs Booléennes, et de retourner une réponse Booléenne.

La première est le **or**, qui renvoie vrai si 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 si ses deux entrées sont vraies. Elle est représentée par le signe **&**:

```
true & true
```

```
true
```

```
true & false
```

false

false &amp; 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** ("ou exclusif"), qui renvoie vrai uniquement ssi l'opération **or** appliquée à des deux entrées renvoie vrai *et* que l'opération **and** renvoie faux. Il est représenté 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 `x1` et `x2`, 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 *nombre*s (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  0
 0  1
 0  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 coordonnées.

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 raccourcis `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 coordonnées 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 dépendances 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'exécuter 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/>

## 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, 001, et 000. La séquence des états qui correspond, 00011110, est la représentation binaire du nombre 30, ce qui donne son nom à la règle.

*Rule 30* peut se représenter de manière beaucoup plus simple, avec la formule suivante:  $p \oplus (q \mid r)$ . On peut donc simuler l'évolution de cette règle 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

```
n_cellules = 191
```

```
191
```

Définir le nombre de générations comme  $(n\_cellules - 1) / 2$  et convertir en entier

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

```
95
```

## TODO

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

Trouver l'index de la cellule au milieu et mettre sa valeur à true

```
milieu_index = div(n_cellules, 2) + 1
shell[milieu_index, 1] = true
```

```
true
```

## Effectuer la simulation

```
for generation in 2:n_generations
    for cellule in 2:n_cellules-1
        p = shell[cellule-1, generation-1]
        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

```

## Afficher l'état final de la simulation

```

heatmap(
    shell,
    colormap=[:white, :black],
    axis=(; aspect=DataAspect()),
    figure=(; size=(3n_cellules, 3n_generations),
    figure_padding=0)
)
hidespines!(current_axis())
hidedecorations!(current_axis())
current_figure()

```

