Initiation aux systèmes multi-agents 1

Présentation de Netlogo

Netlogo est un programme permettant de faire du Système Multi Agents (SMA). Un agent (une tortue en Netlogo) interagit avec les autres agents, et aussi son environnement (formé de patches en Netlogo). Les agents peuvent être de différents types, ainsi que les patchs. On peut par exemple ainsi :

- modéliser le fonctionnement des fourmilières (agents : la reine, les ouvriers, les guerriers, patchs : les différentes parties de la fourmilière, la nourriture, les brindilles,..). Les fourmis ou les abeilles, dans le monde réel, font preuve d'intelligence collective : chaque individu est loin d'être « intelligent », mais la communauté a néanmoins des comportements complexes et très adaptés. En informatique, on parle d'intelligence artificielle distribuée (AID).
- programmer des jeux vidéos comme les Sims (agents : les sims, patch : leur maison, objets dans la maison, etc…). on voit ici un type supplémentaire d'interaction ; c'est l'interaction avec « l'observateur », c'est-à-dire le joueur.
- Modéliser des phénomènes complexes, comme par exemple l'évolution de la couleur d'un insecte pour se camoufler dans son environnement.

Les SMA sont très bien représentés au cinéma dans Matrix, avec les robots insectoïdes du monde « réel ».

1. Première utilisation de Netlogo.

Lancer le programme. Si le raccourci ne fonctionne pas, on peut le lancer en ligne : http://ccl.northwestern.edu/netlogo/

a. Le programme.

Il comporte 3 onglets

- Interface : là où on voit le programme s'exécuter. Il comporte un curseur, qui règle la vitesse d'exécution du programme. Le bouton « settings » donne les propriétés de la carte des patchs (ne pas la modifier maintenant, et à éviter plus tard). L'interface comporte aussi un champ de saisie des commandes *Command Center.*
- Info : permet de préciser ce que le programme fait.
- Code : comme son nom l'indique, c'est là où l'on écrit le programme.

b. Les commandes.

Pour se familiariser avec le langage de programmation, on va d'abord rentrer les instructions/commandes une par une dans le centre de commandes.

Vous avez tout en bas un choix possible entre observer, turtles, patches et link. Laissez (ou choisissez) « observer ».

Tapez :

```
create-turtles 5 [ setxy random-xcor random-ycor]
```

Choisissez « turtles », tapez `set size 2` puis `set color red` puis `set shape ''wolf''` (toutes les formes et les couleurs disponibles peuvent se trouver dans le menu tools en farfouillant)

Choisissez « observer », tapez `ask turtle 1 [set color blue]`. Que constatez-vous ?

Puis tapez `ask turtles [set color green]`. Que constatez-vous ?

Puis tapez `ask turtles [set color random 140]`. Que constatez-vous ?

Choisissez « patch », tapez `set pcolor green`

Choisissez « turtles », tapez `fd 1` puis `rt 45` puis `fd 1` puis `lt 45` puis `fd 1`. Que fait cette suite d'instructions ?

Choisissez « turtles », tapez `pen-down` puis `fd 1`.

Tracez des carrés.

Choisissez « observer », tapez `clear-all`

Cours de Frédéric Mandon sous licence Creative Commons BY NC SA, http://creativecommons.org/licenses/by-nc-sa/3.0/fr/.

c. Les curseurs.
Rajoutez un curseur (choisir « slider » en haut à gauche, dans le menu déroulant, puis cliquer sur le bouton Add). Appelez la variable globale associée `nombre`.
En vous inspirant de ce que vous avez déjà fait, tracez des carrés avec nombre tortues. La syntaxe pour la création est `create-turtles nombre [ setxy random-xcor random-ycor]`

2. Premier programme.

Ouvrir le programme « simple_aleatoire.nlogo », dans le dossier Documents>Devoirs>Mandon. Je vous conseille de l'enregistrer dans un dossier de travail « Documents>Netlogo », ainsi que tous les programmes que vous créerez.
Le code est ci-dessous, vous pouvez le voir également dans l'onglet `code`.
Lancez-le, testez son fonctionnement.
Structure du langage :
Voici le code :

```
to setup
  ;; Il faut peut-etre mettre la commande
  ;; __clear-all-and-reset-ticks
  ;; au début de la procedure, et enlever
  ;; clear-all ainsi que reset-ticks

  clear-all
  set-default-shape turtles "bug"

  ;; place les tortues de maniere aleatoire
  create-turtles nombre [
    set color red
    setxy random-xcor random-ycor
    set size 2
  ]
  reset-ticks
end

to agiter
  rt random 50    ;; angle aleatoire entre 0 et 50 degres
  lt random 50
end

to go
  ask turtles[
    agiter
    fd 1 ]
  tick        ;; fait un tick d'horloge, finit la boucle et revient au debut
                ;;du "go"
end
```

Examinons le programme « `simple_aleatoire.nlogo` » dans l'onglet Code ou ci-dessus.
Les procédures commencent par `to` et finissent par `end`.
Il y a deux procédures principales, le `setup` et le `go` (qui équivalent au `setup` et au `loop` pour Arduino).
On construit d'autres procédures pour les actions des tortues et les modifications des patchs. Ici il y a une procédure « `agiter` », qui fait se mouvoir les tortues.
Dans l'onglet `Interface`, il y a également deux boutons en plus : `setup` et `go`. Avec un clic droit (`Edit…`), regardez les propriétés desdits boutons.
En se basant sur cet exemple, créer le programme « `carrés` », qui dessine des carrés automatiquement. Conservez « `indent automatically` » coché, cela rendra votre code plus lisible. Vous pouvez vérifier la syntaxe du code avec le bouton « `check` »..
Faites ensuite un programme « `triangles équilatéraux` ».
Comme pour le thème Arduino, vous me rendrez le programme fait sous le nom : `carres_votreNom.nlogo`. De préférence dans le dossier Devoirs>Mandon.

Cours de Frédéric Mandon sous licence Creative Commons BY NC SA, http://creativecommons.org/licenses/by-nc-sa/3.0/fr/.

3. **Les variables.**

Vous allez modifier le programme qui trace des carrés pour tracer maintenant des spirales.
Pour cela, il va falloir créer une variable qui donne la longueur du côté de la spirale. En effet cette longueur varie au cours du temps
Les variables sont déclarées au début du code, sous la syntaxe : `turtles-own [longueur]`. Ici on a créé pour chaque tortue une variable « `longueur` ».
Puis, au bon endroit dans le code, pour augmenter la longueur de 1 on tape :

- `ask turtles [set longueur longueur + 1]` si la procédure est une procédure « `observer` »
- `set longueur longueur + 1` si la procédure est une procédure « `turtles` »

Une fois que votre programme est fonctionnel, vous pouvez l'améliorer en faisant des spirales finies.
Pour cela, il faut décocher la case « `forever` » dans le bouton « `go` », et utiliser un « `repeat` ».
Par exemple `repeat 10 [truc]` répètera 10 fois les instructions « `truc` ».
Modifiez votre programme afin qu'il fasse des spirales finies.
Le programme pourra faire les spirales une seule fois, ou bien une fois les spirales faites, effacer et recommencer.
Rendez le programme fait sous le nom : `spirales_finies_votreNom.nlogo`. De préférence dans le dossier Devoirs>Mandon.

4. **Les patches.**

Contrairement aux tortues, les patches ne sont pas mobiles : ce sont les carrés qui constituent le « monde »..
Mais comme les tortues, ils ont des propriétés qui peuvent évoluer.
On peut modifier la taille du monde dans settings, ainsi que la taille des patches. Pour les exercices que l'on va faire, il vaut mieux créer des patches plus petits (taille 5 par exemple), avec un monde de taille 50.

a. Quelques commandes.

Dans le command center, taper :

- soit pour l'observateur `ask patches [set pcolor random 140]`
- soit pour les patches `set pcolor random 140`

Créer un curseur avec comme variable « densité » (ne pas mettre d'accent comme toujours en informatique).
On va créer des patches de couleur soit noire soit blanche, avec une densité fixée par le curseur.
La commande est, dans le command center avec patches :

```
ifelse random-float 100.0 < densite [set pcolor black][set pcolor white]
```

Explications :
`ifelse` est une instruction si... sinon. Si une certaine condition est réalisée, on fait quelque chose, sinon autre chose.
La condition est ici `random-float 100.0 < densite`.
`Random-float 100,0` tire un nombre a virgule au hasard, entre 0 et 100 (en fait entre 0 et 99,99999...)
On teste ensuite si ce nombre est plus petit ou pas que la densité demandée :

- Si le nombre est plus petit que la densité, on met la couleur à noir ;
- Sinon on la met à blanc.

On remarque que les instructions à exécuter sont entre crochets
Si on veut exécuter cette commande en tant qu'observateur, on rajoute :

```
ask patches [ifelse random-float 100.0 < densite [set pcolor white][set pcolor black]]
```

Comment avoir trois couleurs au hasard ?
Dans l'onglet code, taper `patches-own [couleur]` : chaque patch aura sa variable `couleur`.
Dans le command center, soit patches soit observer (mettre ou non « `ask patches` ») :

Cours de Frédéric Mandon sous licence Creative Commons BY NC SA, http://creativecommons.org/licenses/by-nc-sa/3.0/fr/.

- `set couleur random 3 ;; met la couleur à 0, 1 ou 2 (3 exclu)`
- `ifelse couleur = 0 [set pcolor white][ifelse couleur = 1 [set pcolor black][set pcolor pink]]`

Expliquer le fonctionnement de la dernière instruction

Un objet (tortue ou patch) peut aussi observer ce qu'il y a autour de lui. Nous allons avoir besoin de compter, l'instruction dans le command center « observateur » est :

```
count patches with [pcolor = white]
```

Pour compter autour d'un patch (ou d'une tortue), l'observateur peut demander :

```
ask patch 0 0 [show count neighbors with [pcolor = white]]
```

Ici on a compté le nombre de cellules blanches dans les huit cellules autour de l'origine.

b. Le jeu de la vie

Présentation et travail « sur papier »

Le nom est trompeur, ce n'est pas un jeu ; on dit que c'est un jeu « à zéro joueur », puisqu'il joue tout seul ! C'est ce que l'on appelle un automate cellulaire, c'est à dire un processus qui évolue de lui-même, par étapes. Chaque étape correspond à un tick d'horloge. Les cellules sont sur une grille régulière et possèdent un nombre fini d'états, entre lesquels elles évoluent. Les règles d'évolution sont très simples en général, mais donnent des comportements très complexes et souvent imprévisibles.

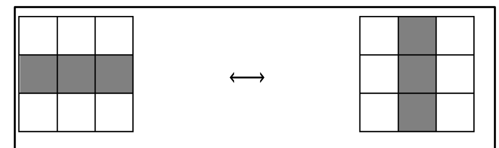
Dans le jeu de la vie, les cellules sont représentées par les patches. Elles sont soient mortes soient vivantes (vivantes en noir par exemple, mortes en blanc).

L'état d'une cellule à l'étape suivante est entièrement déterminé par l'état des 8 cellules qui l'entoure :

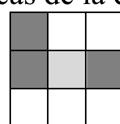
- une cellule morte possédant exactement trois voisines vivantes devient vivante (elle naît) ;
- une cellule vivante possédant deux ou trois voisines vivantes reste vivante, sinon elle meurt.

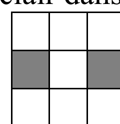
Exemples :

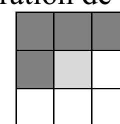
- Ainsi, les configurations ci-contre passent de l'une à l'autre lors d'étapes successives



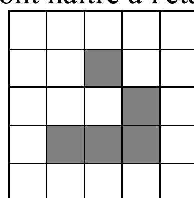
- On peut également formuler cette évolution ainsi :
    - Si une cellule a exactement trois voisines vivantes, elle est vivante à l'étape suivante. C'est le cas de la cellule gris clair dans la configuration de gauche. (1)
    - Si une cellule a exactement deux voisines vivantes, elle reste dans son état actuel à l'étape suivante.
      Dans le cas de la configuration du milieu, la cellule située entre les deux cellules vivantes reste morte à l'étape suivante. (2)
    - Si une cellule a strictement moins de deux ou strictement plus de trois voisines vivantes, elle est morte à l'étape suivante.
      C'est le cas de la cellule gris clair dans la configuration de droite (3).

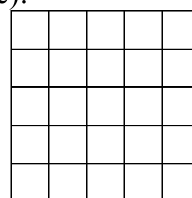

(1)


(2)


(3)

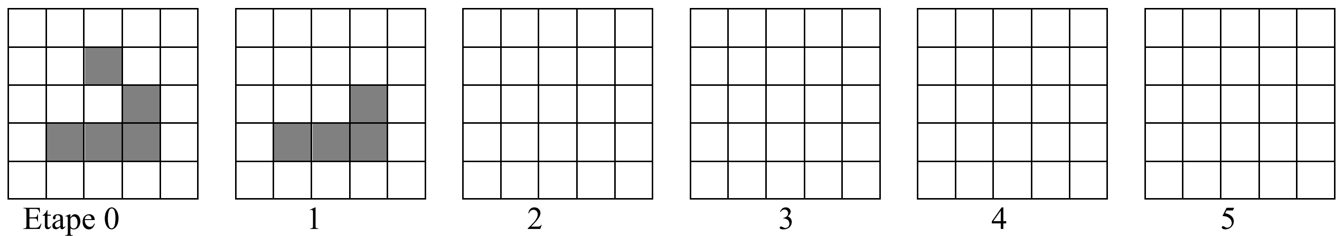
À la main, donner l'étape suivante de la configuration ci dessous (on peut marquer d'un point vert toutes les cellules qui vont naître à l'étape suivante):





Cours de Frédéric Mandon sous licence Creative Commons BY NC SA, http://creativecommons.org/licenses/by-nc-sa/3.0/fr/.

Programmation
Comme dans les programmes avec les tortues, on va avoir un bouton setup et un bouton go.
`Setup` : facile
Pour le setup, reprendre la création de cellules vivantes/mortes avec le curseur de densité
`Go` : plus difficile
Pour bien comprendre le fonctionnement du « `go` », reprenons le papier et le crayon, ainsi que l'exemple, mais en le faisant tourner avec un algorithme faux :



Imaginez que vous modifiiez immédiatement les cellules une par une, dès que vous avez observé les cellules environnantes, et non pas toutes en même temps : à la première étape, la cellule en haut meurt puisqu'elle n'a qu'une seule voisine vivante. Compléter les étapes suivantes et comparez avec le résultat attendu.

Pour éviter cette erreur, il va donc falloir dans un premier temps examiner toutes les cellules pour prévoir leur état à l'étape suivante, et ensuite seulement modifier leur état.
Cela nécessite trois choses :

- de garder en mémoire l'état vivant ou mort d'une cellule
- de garder en mémoire le nombre de voisins vivants d'une cellule
- de d'abord parcourir toutes les cellules pour compter les voisins vivants, pour seulement ensuite modifier toutes les cellules d'un coup

Pour garder en mémoire une donnée, on crée une variable comme on l'a fait pour le programme du dessin des spirales.
Pour les variables on écrira au début du programme :

```
patches-own [
vivant? ;; pour indiquer si la cellule est vivante ou non
nombreVoisinsVivants ;; pour compter le nombre de voisins vivants...
]
```

La variable `vivant?` vaut `true` ou `false`.
La variable `nombreVoisinsVivants` est un nombre entier (compris entre 0 et 8 ici)

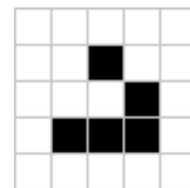
Dans le « `go` », il y aura les deux parties :

```
to go
   ask patches [compter-voisins]
   ask patches [mettre-a jour]
   tick
end
```

On crée deux procédures supplémentaires
`to compter-voisins...end`
Pour changer la valeur d'une variable, on utilise `set` (voir les spirales), qu'on va ici mélanger avec `count`. Ecrire l'instruction demandée dans cette procédure, qui compte le nombre de voisins vivants et met à jour la variable `nombreVoisinsVivants`.
`to mettre-a-jour... end`
Ici on va changer la valeur de `vivant?`, ainsi que la couleur du patch
On rappelle qu'une cellule devient vivante si elle a exactement 3 voisines vivantes, et que sinon elle meurt si elle a tous sauf 2 voisines vivantes. Si nécessaire, le symbole différent s'écrit `!=`.

Programmer le jeu de la vie, et tester son fonctionnement. Que se passe-t-il souvent ?

Cours de Frédéric Mandon sous licence Creative Commons BY NC SA, http://creativecommons.org/licenses/by-nc-sa/3.0/fr/.

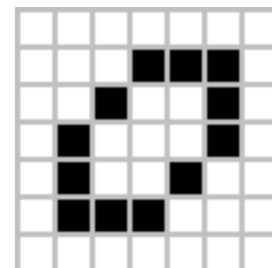
Un bouton supplémentaire.
Rajouter un deuxième bouton de setup : le setup-glider
On va créer une nouvelle procédure associée : *`to setup-glider... end`*

Dans cette procédure, mettre tous les patches à blanc (morts), puis tracer la figure du glider (ci-contre). Je vous rappelle que l'on peut demander à un patch spécifique une action, comme ceci :

- `ask patch 0 1 [set pcolor black]`

Tester le fonctionnement du jeu de la vie avec le glider (et après regarder la traduction !)
Rendez le programme sous le nom `jeu_de_la_vie_votreNom.nlogo`

c. Variantes
Vous pouvez programmer facilement une ou plusieurs de ces variantes à partir du jeu de la vie originel, seules les conditions de changement d'état vivant/mort sont à modifier. Rendez les programmes que vous faites.

i. *Day & Night*
Une cellule morte y naît à l'étape suivante si elle est entourée de 3, 6, 7 ou 8 voisines vivantes, une cellule vivante survit à l'étape suivante si elle est entourée de 3, 4, 6, 7 ou 8 cellules vivantes.

ii. *High Life*

Une cellule morte y naît à l'étape suivante si elle est entourée de 3 ou 6 voisines vivantes, une cellule vivante survit à l'étape suivante si elle est entourée de deux ou trois cellules vivantes. Essayer la figure ci-contre en démarrage, elle donne un résultat intéressant.

d. Les automates cellulaires à plusieurs états.
Programmez au moins « le cerveau de Brian », sous le nom `cerveau_Brian_votreNom.nlogo` , à rendre.

i. *le cerveau de Brian est en feu…*
Les cellules ont trois états : en feu (blanche), réfractaire (rouge), et morte (noires).
Les cellules en feu (blanches) deviennent toujours réfractaires (rouges) à l'étape d'après.
Les cellules réfractaires (rouges) deviennent toujours mortes (noires) à l'étape d'après.
Une nouvelle cellule prend feu (blanche) à l'étape suivante quand elle a exactement deux voisines en feu.

ii. *Immigration*
Immigration fonctionne exactement de la même façon que le jeu de la vie, à ceci près qu'il possède trois états, dont deux « vivants ». Une cellule morte y naît à l'étape suivante si elle est entourée de 3 voisines, une cellule vivante survit à l'étape suivante si elle est entourée de 2 ou 3 cellules vivantes.

iii. *QuadLife*
QuadLife fonctionne exactement de la même façon que le jeu de la vie, à ceci près qu'il possède cinq états, dont quatre « vivants ». Une cellule morte y naît à l'étape suivante si elle est entourée de 3 voisines, une cellule vivante survit à l'étape suivante si elle est entourée de 2 ou 3 cellules vivantes.
Lorsqu'une cellule naît, si toutes les cellules qui lui ont donné naissance se trouvent dans des états différents, la nouvelle cellule prend l'état restant. Dans le cas contraire, elle prend l'état de la majorité des trois cellules.

Cours de Frédéric Mandon sous licence Creative Commons BY NC SA, http://creativecommons.org/licenses/by-nc-sa/3.0/fr/.

Initiation aux systèmes multi-agents 2

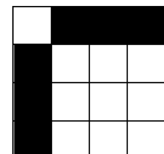
1. Reprise du jeu de la vie.

a. Corrigé

Ci-dessous le code, présent aussi dans votre dossier « devoirs>mandon ». N'oubliez pas de rajouter les boutons « setup », « go », ainsi que le curseur densité.

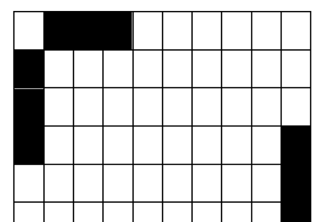
```
patches-own [
  vivant?          ;; etat de la cellule vivante (true) ou morte (false)
  nbVoisinsVivants  ;; compte le nombre de cellules voisines vivantes
]

to setup
  clear-all
  ask patches
    [ ifelse random-float 100.0 < densite
      [ cellule_vivante ]
      [ cellule_morte ] ]
  reset-ticks
end

to cellule_vivante
  set vivant? true
  set pcolor black
end

to cellule_morte
  set vivant? false
  set pcolor white
end

to go
  ;; on compte le nombre de voisins vivants pour toutes les cellules
  ask patches
    [compter_voisins]

  ;; puis on modifie l'etat de la cellule
  ask patches
    [ mettre_a_jour ]
  tick
end

to compter_voisins
  set nbVoisinsVivants count neighbors with [vivant?]
end

to mettre_a_jour
  ifelse nbVoisinsVivants = 3
      [ cellule_vivante ]
      [ if nbVoisinsVivants != 2
        [ cellule_morte ] ]
end
```

b. High Life.



Programmez la variante « High Life » à partir du corrigé précédent

Une cellule morte y naît à l'étape suivante si elle est entourée de 3 ou 6 voisines vivantes, une cellule vivante survit à l'étape suivante si elle est entourée de deux ou trois cellules vivantes. Vous pouvez inverser la condition précédente (c'est-à-dire répondre à la question : quand est-ce qu'une cellule meurt ?), pour que la programmation soit plus facile.



Essayer les deux figures ci-contre en démarrage, elles donnent un résultat intéressant. La commande pour exécuter des instructions pour un patch donné par ses coordonnées est : `ask patch` *`coord_x coord_y`* `[…]`

Cours de Frédéric Mandon sous licence Creative Commons BY NC SA, http://creativecommons.org/licenses/by-nc-sa/3.0/fr/.

c. Le cerveau de Brian.

Programmez cette variante. Les cellules ont trois états, plutôt que vrai/faux ou mettra un nombre 0/1/2 ou 1/2/3.
Les cellules ont trois états : en feu (blanche), réfractaire (rouge), et morte (noires).
Les cellules en feu (blanches) deviennent toujours réfractaires (rouges) à l'étape d'après.
Les cellules réfractaires (rouges) deviennent toujours mortes (noires) à l'étape d'après.
Une nouvelle cellule prend feu (blanche) à l'étape suivante quand elle a exactement deux voisines en feu.

2. Interactions turtles/patches.

a. La fourmi de Langton.

Cet automate cellulaire, qui a des règles très simples, permet de mettre en évidence un comportement « émergent », c'est-à-dire un comportement complexe, imprévisible à partir des règles initiales.
La fourmi est une « turtle » (dans l'optique netlogo), qui si déplace sur les patches du « monde ».
Initialement, tous les patches sont soit noirs soit blancs.
La fourmi peut se déplacer à gauche, à droite, en haut ou en bas d'une case à chaque fois selon les règles suivantes :

- Si la fourmi est sur une case noire, elle tourne de 90° vers la gauche, change la couleur de la case en blanc et avance d'une case.
- Si la fourmi est sur une case blanche, elle tourne de 90° vers la droite, change la couleur de la case en noir et avance d'une case.

b. Indications pour la programmation.

Dans Settings (taille du monde dans l'interface), faire un monde assez grand (coordonnées maximales 120, taille du patch 2 par exemple).
Dans le `setup` :

- Mettre tous les patches d'une seule couleur.
- Créer une tortue ni blanche ni noire, de taille 6 au moins.
- Dans le create-turtle, forcer la direction avec « `face one-of neighbors4` ».

Dans le `go` :

- On rappelle les instructions `fd` (pour avancer), `rt` et `lt` (pour tourner).
- Également `set pcolor`, `ifelse`…

c. Complément.

Faire un bouton « alenvers », qui permet de remonter dans le temps et de retrouver la position initiale.
On peut aussi observer ce qui se passe avec un monde non homogène au départ (les patches étant crées comme dans le jeu de la vie, soit noirs, soit blancs, suivant un curseur de donnant la densité).

Cours de Frédéric Mandon sous licence Creative Commons BY NC SA, http://creativecommons.org/licenses/by-nc-sa/3.0/fr/.

d. L'herbe, les moutons, voire les loups.

On va chercher à modéliser l'évolution d'une population animale en fonction des ressources (voire la coévolution de deux populations). Pour cet exercice final, utilisez les programmes précédents pour vous guider : je ne vous donne que les nouvelles instructions de codage, à vous de réutiliser les anciennes.

Créer un monde (settings dans interface) de coordonnées maximales 25 et 25, taille du patch 9.

Créer plusieurs curseurs, les valeurs initiales sont proposées entre parenthèses :

- densité de l'herbe (50 %) ;
- temps nécessaire pour la repousse de l'herbe (30)
- nombre initial de moutons (100) ;
- taux de reproduction des moutons (4%)
- gain d'énergie (des moutons quand ils mangent de l'herbe, 4).

Dans le code, les tortues auront une variable « énergie », et les patchs une variable « compteur de temps ». Les patchs seront marrons sans herbe et vert avec.

Les moutons seront blancs, de taille 1,5, avec une énergie aléatoire (par exemple `set energie random (2 * gain_mouton_herbe)`).

Dans le « go », on écrira plusieurs fonctions :

- déplacement aléatoire des moutons (comme le tout premier exemple de ce cours) ;
- une fonction « manger_herbe » qui diminue l'énergie de 1 à chaque déplacement s'il n'y a pas d'herbe sur le patch, sinon augmente l'énergie du mouton suivant le gain, remet le compteur de temps du patch à 0, et la couleur à marron.
- Une fonction « reproduction ». Si un nombre aléatoire entre 0 et 100 est plus petit que le pourcentage de reproduction, alors la tortue/mouton se reproduit dans un patch adjacent. La syntaxe est `hatch 1 [ rt random-float 360 fd 1 ]`. L'énergie du parent est divisée par deux. Vous remarquerez que la reproduction se fait ici par parthénogénèse (le mouton se divise en 2 !).
- Une fonction « mort » : si l'énergie est négative, le mouton meurt. Syntaxe : `if energy < 0 [ die ]`
- Une fonction « herbe_pousse » : si la couleur du patch est marron, on augmente le compteur de temps, et si ce compteur est supérieur ou égal au temps nécessaire à la repousse, on met le patch en vert.

Enfin, dans l'interface, on peut tracer automatiquement les courbes qui montrent l'évolution des populations. Ca se fait avec « Add », on choisit « plot ». On ajoutera deux « crayons » (pen), l'un pour les moutons, l'autre pour l'herbe (divisée par 4 pour tenir sur le schéma). Syntaxes dans « pen update command » : `plot count sheep` et `plot grass / 4`. Faire varier les paramètres et observer les différents scénarios d'évolution.

Pour ceux qui sont très rapides, on peut rajouter les loups (qui vont manger les moutons).

Dans le code il y aura deux types de tortues, on le précise en écrivant au tout début :

```
breed [ moutons mouton ]          ; il y a le pluriel et le singulier comme turtle et turtles
breed [ loup loups ]
```

Plutôt que `create-turle`, on écrira `create-moutons` et `create-loups`.

Il y aura des curseurs similaires à ceux des moutons (nombre de loups 50, gain d'énergie 20, taux de reproduction des loups 5 %), ainsi que les fonctions de déplacement, de reproduction et de mort.

Enfin, la procédure lorsque les loups mangent des moutons est celle-ci (à compléter) :

```
to manger_moutons
  let prey one-of moutons-here  ;le loup essaie de manger un mouton
  if prey != nobody             ; le loup a attrapé un mouton ? Si oui
    [ ask prey […]              ; il le tue
      …]                        ; et son énergie augmente
end
```

On complètera également les courbes de population.

Cours de Frédéric Mandon sous licence Creative Commons BY NC SA, http://creativecommons.org/licenses/by-nc-sa/3.0/fr/.