université PARIS-SACLAY

# Projet de programmation

**Cours de programmation impérative**

*—Licence MPI L1 S2 - Info 121—*

## Système prédateurs / Proies

Le but de ce projet est de simuler le comportement d'un écosystème comportant deux espèces d'animaux : une espèce prédatrice (les renards) et une espèce proie (les lapins). D'un points de vue mathématique, la dynamique d'un tel système biologique a été modélisée indépendamment par Alfred James Lotka en 1925 et Vito Volterra en 1926 (voir `https://fr.wikipedia.org/wiki/Équations_de_prédation_de_Lotka-Volterra`). Les résultats observés montrent que l'effectif des deux populations suit un comportement oscillant avec un retard des prédateurs sur les proies :

- S'il y a beaucoup de proies, les prédateurs sont bien nourris et se reproduisent jusqu'à ce qu'ils aient mangé trop de proies. Ils commencent alors à mourir de faim.
- Quand le nombre de prédateurs devient faible, les proies peuvent se reproduirent à nouveau, relançant le cycle.

La modélisation mathématique de Lotka-Volterra utilise un couple d'équations différentielles non-linéaires du premier ordre.
Dans notre cas, plutôt que de modéliser le comportement mathématique des fonctions décrivant les effectifs des deux populations, nous allons modéliser de manière ultra simplifiée le comportement individuel de chaque animal. En fonction des paramètres (taux de reproduction, taux de prédation), nous devrions pouvoir observer des comportement analogues.

---

Nous allons modéliser de manière simplifiée le comportement individuel de chaque animal.

Les animaux vont se déplacer sur une grille, par exemple de 20 lignes et 20 colonnes. Une case de la grille peut être vide, ou contenir un renard ou un lapin. Une case ne peut pas contenir en même temps deux animaux. On initialise la grille de façon aléatoire avec 20% de lapins et 7% de renards.

A chaque tour, tous les animaux se déplacent dans une case voisine, en suivant les règles décrites dans les sections suivantes. Il y a donc 8 cases possibles au maximum pour le déplacement d'un animal situé au milieu de la grille. On suppose que les bords de la grille sont des murs infranchissables. L'ordre des déplacements se fera de manière arbitraire (par exemple ligne par ligne ?) ou mieux aléatoirement.

Un déplacement, peut de plus, engendrer une reproduction – le nouvel animal apparaît alors sur la case anciennement occupée – ou la mort d'un animal. Pour simplifier, nous supposerons que les animaux se reproduisent seuls (sans rencontre avec un animal du sexe opposé) et instantanément. Si un renard rencontre un lapin, il le mange et le lapin disparaît. Si un renard ne mange pas pendant trop longtemps, il meurt de faim. Pour simplifier les règles, les lapins mangent de l'herbe et ne meurent jamais de faim. On suppose également que les animaux ne meurent pas de vieillesse.

# 1 Règles

Les déplacements se feront en deux temps : on déplace d'abord tous les lapins, puis ensuite tous les renards.

## 1.1 Comportement des lapins

Les lapins se comportent de la manière suivante :

- Le lapin choisit aléatoirement une case vide voisine et se déplace dessus ;

— Si le nombre de cases voisines vides avant son déplacement était au moins égal à la constante `MinFreeBirthLapin`, le lapin se reproduit avec une probabilité égale à `ProbReproLapin`. Dans ce cas, un nouveau lapin apparaît à l'ancienne position du parent.

## 1.2 Comportement des renards

Un renard possède comme attribut un niveau de nourriture : plus ce niveau est élevé mieux le renard est nourri. À la naissance du renard, ce niveau est égal à `FoodInit`, lorsqu'il mange un Lapin, il augmente de `FoodLapin`, et il diminue de une unité à chaque début de tour. On suppose que lorsque le niveau est égal à zéro, le renard meurt de faim. De plus, le niveau de nourriture ne peut pas dépasser un certain seuil fixé à `MaxFood`. Finalement, pour se reproduire, un renard doit avoir suffisamment mangé, son niveau de nourriture doit être supérieur ou égal à `FoodReprod`.

Le déplacement d'un renard s'effectue ainsi de la manière suivante :
— Le niveau de nourriture diminue de une unité, s'il atteint 0 le renard meurt et disparaît.
— Si l'une des cases voisines contient un lapin, le renard s'y déplace, et mange le lapin. Si plusieurs cases voisines contiennent un lapin, le renard en choisit une au hasard.
— Sinon, le renard choisit aléatoirement une case vide et s'y déplace.
— S'il a suffisamment mangé (niveau de nourriture supérieur à `FoodReprod`), il se reproduit avec une probabilité de `ProbBirthRenard`, et un nouveau renard apparaît à l'ancienne position du parent.

## 1.3 Simulation

La simulation consiste essentiellement à initialiser la grille en plaçant les renards et les lapins, puis à itérer un certain nombre de fois le déplacement de chacun. Il est nécessaire de mettre plus de lapins que de renards à l'initialisation pour que les lapins survivent assez longtemps.

### Initialisation de la grille

Chaque case est initialisée de manière aléatoire, en plaçant un lapin, un renard ou rien. Tous les animaux possèdent les mêmes paramètres de comportement. Il est intéressant de voir comment le comportement global évolue en fonction de ces paramètres. Les réglages ci-dessous induisent un comportement périodique qui correspond à la théorie.

Pour les renards :
— `FoodInit` = 5 ;
— `FoodLapin` = 5 ;
— `FoodReprod` = 8 ;
— `MaxFood` = 10 ;
— `ProbBirthRenard` = 0.05 ;
Pour les lapins :
— `ProbBirthLapin` = 0.30 ;
— `MinFreeBirthLapin` = 4 ;

Note : en fonction du hasard, il se peut que tous les renards meurent ou qu'ils aient mangé tous les lapins. Dans ce cas le cycle s'arrête. On remarque que plus le monde est grand, moins il y a de chance pour que cela se produise.

### Déplacement des animaux

Les déplacements sont organisés de la manière suivante : on recense les lapins présents dans la grille et on les déplace (et fait éventuellement se reproduire) dans un ordre fixe ou mieux aléatoire, puis on fait de même avec les renards.

## 2 Affichage

Dans un premier temps pour le test, on pourra faire un affichage texte avec, par exemple, `'L'` pour représenter un lapin, `'R'` pour les renards, et `0` pour une case vide.

Pour aller plus loin, on pourra utiliser un affichage graphique, qui permet d'avoir des couleurs. On représente chaque case de la grille par un pixel coloré. La couleur dépend du contenu de la case, par exemple

— les renards en rouge
— les lapins en bleu
— les cases vides en blanc

Pour l'affichage graphique, on propose d'implémenter une solution non-interactive (mais si vous avez le temps de faire une solution interactive, par exemple avec la bibliothèque `SFML`, n'hésitez pas). On va générer une suite d'images au format «ppm» dont les noms seront de la forme `img000.ppm`, `img001.ppm`, `img002.ppm` .... représentant chaque étape de la simulation. On va ensuite utiliser soit `convert` (sous Linux et Mac), soit le logiciel *beneton movie* pour les rassembler en un fichier gif animé :

— Sous Linux ou Mac, les fichiers ppm générés peuvent être rassemblés dans un fichier gif animé par la commande

```
convert -scale 300 -delay 10 img*.ppm movie.gif
```

Le paramètre 10 après delay peut être changé pour varier la vitesse d'animation. Le fichier gif peut ensuite être visionné avec Firefox. L'outil `convert` peut être installé sous Ubuntu avec :

```
sudo apt-get install imagemagick
```

— Sous Windows, on peut utiliser le logiciel *beneton movie* qui est téléchargeable gratuitement à l'adresse

`https://beneton-movie-gif.fileplanet.com/`

Concernant la bibliothèque SFML, vous trouverez des informations de démarrage dans votre cours de premier semestre :

`https://nicolas.thiery.name/Enseignement/Info111/Assignments/Semaine10/index.html#exercice-premiers-graphiques-avec-sfml`

Pour compiler avec la SFML, il faut rajouter les trois options

```
-lsfml-system -lsfml-window -lsfml-graphics
```

au moment de l'édition des liens (variable `LDFLAGS` du `Makefile` proposé en cours).

### 2.1 Pour aller plus loin

Pour avoir une simulation plus réaliste, on pourra faire les améliorations suivantes :

1. **Déplacement des animaux dans un ordre aléatoire** : dans un premier temps pour les déplacements des animaux, on parcourt la structure recensant toute la population et on déplace les animaux dans l'ordre où on les rencontre. Une première amélioration consiste à d'abord recenser tous les animaux à déplacer, puis à les déplacer dans un ordre aléatoire.
2. **Tenir compte de l'age des animaux** : un animal trop vieux meurt de vieillesse.
3. **Reproduction sexuée** : un animal ne peut se reproduire que si il rencontre un autre animal de la même espèce. Dans une version avancée, on peut de plus tenir compte du sexe.

Attention ! Avant de faire une amélioration, sauvegardez une version qui marche pour pouvoir la présenter à la soutenance, dans le cas où vous n'arrivez pas à faire tourner l'amélioration.

# 3 Implémentation de la simulation

Afin de vous aider à réaliser le projet, nous vous donnons une analyse de la suite d'actions à mettre en œuvre pour implémenter la simulation.

**Àprès l'écriture de chaque fonction, il faut vérifier soigneusement qu'elle fonctionne bien**. Soit quand c'est possible avec une autre fonction de tests que l'on **conservera**, soit par un affichage. Dans ce second cas, on essayera également de conserver dans une procédure, le code qui permet de reproduire ce test, sans le lancer automatiquement.

1. Spécifier les **types abstraits** suivants :
   — `Coord` pour contenir les coordonnées d'un point de la grille ;
   — `Ensemble` pour contenir un ensemble d'entiers ; Ce type pourra servir notamment à regrouper les identifiants d'un ensemble d'animaux.
   — `Animal` pour coder un animal ;
   — `Population` pour coder la population des animaux ; Chaque animal est repéré dans la population par un identifiant qui permet de le retrouver.
   — `Grille` pour contenir l'état de la grille à un moment donné.
   — `Jeu` pour contenir l'état du jeu à un moment donné.
2. Décider des types concrets associés.
   *Aide 1 :* Un animal doit être associé à ses coordonnées dans la grille. Pour cela, utiliser le type concret `coord` défini dans le TP.
   *Aide 2 :* Une case de la grille peut soit être vide soit contenir un animal. On identifiera un animal par son identifiant dans la population et on utilisera `Vide = -1` pour indiquer qu'une case est vide.
   *Aide 3 :* Un animal peut être soit un lapin soit un renard. On pourra déclarer un type énuméré.
   *Aide 4 :* Une paire de coordonnées $(x, y)$ dans une grille carrée de côté $n$ peut être associée à l'entier $xn + y$. On pourra écrire des fonctions et/ou des constructeurs adaptés pour faire la traduction lorsque cela est utile. Ainsi les ensembles d'entiers `Ens` pourront aussi représenter des ensembles de coordonnées.
3. **Initialiser une grille** $20 \times 20$ avec le contenu de chaque case choisi aléatoirement.
4. Afficher le contenu de cette grille dans le terminal pour vérifier que l'initialisation a correctement été effectuée.
5. Écrire une fonction `voisinsVide` qui renvoie l'ensemble des coordonnées des cases voisines vides d'une case donnée. Même chose pour `voisinsLapin` qui renvoie les cases contenant un lapin. Dans l'une comme dans l'autre, utiliser la correspondance entre coordonnées et entiers.
6. Écrire une fonction qui déplace aléatoirement les animaux (sans reproduction ni mort). Vérifier que tout fonctionne bien, en particulier qu'aucun animal ne disparaît ou n'apparaît.
7. Rajouter les règles où les renards cherchent à manger les lapins ; Ces derniers ne devraient alors pas survivre très longtemps ;
8. **Comportement des animaux** : Écrire des fonctions qui décident selon l'espèce, l'état de l'animal et le résultat d'un tirage aléatoire, si d'une part, l'animal survit et d'autre part, s'il se reproduit. Pour ce dernier cas, et dans le cas des lapins, il faut aussi que la fonction reçoive en paramètre le nombre de cases voisines libres (voir les règles de déplacement des lapins).
9. Coder le comportement complet des renards et des lapins. Afficher à chaque instant le nombre de renards et de lapins.

# 4 Organisation et évaluation du projet

Deux séances de TP (l'une dirigée, l'autre libre) porteront sur le projet. N'hésitez pas à poser des questions à vos encadrants de TP pour régler les problèmes que vous ne parvenez pas à surmonter. Il est indispensable de commencer à travailler sur le projet dès maintenant.

L'évaluation du projet se fera au travers d'une soutenance orale. L'organisation de la soutenance est la suivante :

- vous vous installez sur une machine, chargez votre programme et vérifiez rapidement que tout fonctionne comme la dernière fois que vous l'avez testé ;
- les deux membres du binôme ont 5 mn pour présenter ensemble le résultat de leur travail (organisation du code, visualisation des résultats, discussion sur les problèmes rencontrés...) ;
- ensuite l'examinateur a 5 mn pour juger de votre maîtrise de la programmation. À partir de ce moment, vous serez considérés individuellement, et un effort important sera fait pour mettre en avant clairement votre implication dans le projet. Un étudiant ne peut prétendre être noté sur un code qu'il ne réussit pas à expliquer.
- Un étudiant absent à la soutenance aura zéro sur vingt.

Le principal critère d'évaluation est le nombre d'actions fonctionnant correctement. Nous prendrons également en compte :

- Qualité de présentation...
- Types abstraits (tous les attributs sont privés) : pertinence, choix alternatifs, utilisation dans le code...
- Qualité du code : modularité, lisibilité...
- Originalité : gestion de problèmes rencontrés, affichage d'informations intermédiaires...

**Attention !** Les critères ci-dessus ne seront pas évalués en regardant seulement le code, mais également par la manière dont vous répondez aux questions lors de la soutenance. Il est tout à fait autorisé de se faire aider, mais il faut avoir *écrit soit même, compris et être capable d'expliquer le code que l'on présente.* La note des deux membres du binôme est individuelle et dépend de leurs réponses aux questions et de leur implication dans le projet.

# 5 Annexe

Un fichier `ppm` (voir `https://fr.wikipedia.org/wiki/Portable_pixmap`) contient une image décrite par la suite de caractères suivante :

1. Un "magic number" identifiant le type.
   Pour un fichier ppm il s'agit des deux caractères "P3".
2. Des espaces (blancs, TABs, CRs, LFs).
3. Une largeur L, formatée avec des caractères ASCII en décimal.
4. Des espaces.
5. Une hauteur H, également en ASCII en décimal.
6. Des espaces.
7. La valeur de couleur maximale, en ASCII en décimal. Comprise entre 1 et 65536.
8. Un retour à la ligne ou d'autres caractères espace.
9. Un tableau de H rangées, allant de haut en bas, chaque rangée contient L pixels de gauche à droite. Chaque pixel est un triplet des composantes RGB (c'est-à-dire rouge, vert, bleu) de la couleur, dans cet ordre. Tous les nombres doivent être précédés et suivis d'un espace. Les lignes ne doivent pas dépasser 70 caractères.

Voici l'exemple d'un fichier img000.ppm contenant une image de 16 pixels (4 par 4), représentant une diagonale rouge sur fond vert (les composantes RGB de la couleur rouge sont (255,0,0) et celles de la couleur verte sont (0,255,0)).

```
P3
4 4
255
255 0 0    0 255 0    0 255 0    0 255 0
0 255 0    255 0 0    0 255 0    0 255 0
0 255 0    0 255 0    255 0 0    0 255 0
0 255 0    0 255 0    0 255 0    255 0 0
```

Ce fichier ppm a été généré par le programme suivant :

```
#include <iostream>     // pour cout
#include <iomanip>      // pour setfill, setw
#include <sstream>      // pour ostringstream
#include <fstream>      // pour ofstream
#include <string>

using namespace std;
// variable globale permettant de creer des noms de fichiers differents
int compteurFichier = 0;
// action dessinant un damier
void dessinerDamier(){
  int i,j;
  int r,g,b;
  ostringstream filename;
  // creation d'un nouveau nom de fichier de la forme img347.ppm
  filename << "img" << setfill('0') << setw(3) << compteurFichier << ".ppm";
  compteurFichier++;
  cout << "Ecriture dans le fichier : " << filename.str() << endl;
  // ouverture du fichier
  ofstream fic(filename.str(), ios::out | ios::trunc);
  // ecriture de l'entete
  fic << "P3" << endl
      << 4 << " " << 4 << " " << endl
      << 255 << " " << endl;
  // ecriture des pixels
  for (i = 0; i < 4; i++){
      for (j = 0; j < 4; j++){
        // calcul de la couleur
        if (i == j) { r = 255; g = 0; b = 0; }
        else { r = 0; g = 255; b = 0; }
        // ecriture de la couleur dans le fichier
        fic << r << " " << g << " " << b << "    ";
      }
    // fin de ligne dans l'image
    fic << endl;
  }
  // fermeture du fichier
  fic.close();
}
int main (){
  dessinerDamier();
  return 0;
}
```