Compte rendu Simulation Croissance de lapins

BELLEC Louison, BOUVARD Alexandre

Table of contents

Paramètres initiaux

Pas temporel

Pour avoir une simulation plus réaliste, nous avons choisi un pas temporel de `1 mois`.

Adulte

Les adultes sont les lapins qui peuvent se reproduire. Tous les lapins deviennent mature entre `5 et 8 mois`. Nous le determinons aléatoirement au moment de la creation du lapin.

Mortalité

Nous avons choisis de suivre un taux de mortalité global qui ne détaille pas les causes de la mort. Pour les enfants, le taux de mortalité commence a `65%` puis

passe à 40% une fois arrivés à l'âge adulte. Enfin, les lapins qui atteignent l'âge de 10 ans se voient augmenter leur taux de mortalité de 10% au début de chaque année jusqu'à leurs 16 ans où il atteindra 100% et où ils mourront forcément.

Naissance

Nous avons estimé que chaque mois un mâle pouvait féconder au maximum 15 femelles. Du coté des femelles, elles ont autant de chances de faire 5, 6 et 7 portées par ans, soit 30% et respectivement 5% de faire 4 et 8 portées dans l'année. A chaque portée, elles ont une probabilitée uniforme de faire entre 3 et 6 lapereaux. Le temps de gestation d'une femelle est de 1 mois.

Contraintes techniques

Augmentation exponentielle

A chaque mois passé le nombre d'individu augmente exponentiellement. Par conséquent on arrive très vite à 1 million d'individus. L'étape suivante va donc également multiplier ces 1 millions d'individu et le nombre va très vite être trop élevé pour que le temps d'éxecution reste convenable et que l'ont puisse avoir plusieurs résultat à exploiter.
Par conséquent nous faisons tourner notre simulation sur un nombre limité d'année: entre 7 et 10.

Details algorithmique

A chaque fois que nous avons un évenement avec une certaine probabilité, nous générons un nombre entre 0 et 1. Pour le générer, nous utilisons la fonction `uniform_real_distribution` en utilisant le générateur aléatoire de Mersenne Twister qui existe directement dans la librarie standard c++.

```
/**
 * @brief Genere un nombre aléatoire entre deux nombres.
 * Utilises le generateur de mersenne twister
 *
 * @param a float
 * @param b float
 * @return float
 */
float Generator::randomBetween(float a, float b) {
    std::uniform_real_distribution<double> dis(a, b);

    return dis(Generator::get());
```

```
}

/**
 * @brief The generateur mersenne twister

 * @return
 */
std::mt19937& Generator::get() {
    static std::mt19937 generator(SEED);
    return generator;
}
```

Ensuite, si le nombre est inférieur a notre nombre random, l'événement arrive, sinon il n'arrive pas.

Pour simuler le fait d'avoir entre 4 et 8 portées par an par femelle, nous procédons de la maniere suivante:
Avec un histogramme discret, nous materialisons les probabilités d'avoir 4 et 8 portées (`5%`) et 5, 6, 7 portées (`30%`).
Pour cela, nous avons la fonction de répartition qui est donc la somme des probabilitées. Nous générons le nombre et nous regardons à quel évènement il correspond (par exemple : 0.20 correspond à 5 portées).
Ensuite, nous générons un calendrier de naissances par individu à partir du nombre de portée à avoir dans l'année en les repartissant aléatoirement. Pour optimiser ces choix de mois, on créé un tableau d'indexes de 1 à 12 correspondants aux mois puis on tire à chaque fois un nombre aléatoire entre 0 et 1. On ajoute dans le calendrier que ce mois aura une gestation puis on remplace sa valeur dans le tableau des mois par la dernière du tableau en question. Cela nous permet de faire exactement `nbPortées` tirages et pas plus.

Complexite Algorithmique

Notre algorithme principal est en O(n*m) avec n le nombre de mois et m le nombre d'individus dans la population. Sachant que m augmente a chaque étape.

Compilation

Nous utilisons un projet CLion, et nous compilons avec CMake. Pour utilisons le flag `-DCMAKE_BUILD_TYPE=Release` pour utiliser les optimisation de compilation de CMake et accélerer l'execution.
Nous avons aussi fait des essais et fait tourner la simulation avec g++ et le flag `-O2`.
Nos resultats utilisé sont d'ailleurs issus d'éxecution avec g++.

Visualisation des résultats

Nous avons choisi de ne pas effectuer de visualisation de la simulation en direct pour des raisons de performances d'une part et de temps d'autre part.
En ce qui concerne les résultats, nous recuperons des données brutes que nous exploitons ensuite avec excel pour obtenir une courbe de la croissance de la population.
Les resultats sont enregistre dans des fichiers automatiquement pour les garder.

Questionnement sur la veracité des statistiques

On va calculer la moyenne, l'écart type et l'intervalle de confiance a l'aide de la méthode vu dans le TP 3. On peut en revanche se questionner sur la véracité des résultats car la loi de Student fait intervenir le quotient entre une variable suivant une loi normale centrée réduite et la racine carrée d'une variable distribuée suivant la loi du X2. Or on suppose que notre simulation suit plutôt une loi exponentielle.

Discussion sur la reproductibilite

Nous avons verifié la bonne reproductibilité de nos resultats. Nous avons utilisé un génerateur random scientifique: le Mersenne Twister. Malgré tout, nous avons pu constater des anomalies de reproductibilité. Plus de details peuvent être trouvés dans le constat de non reproductibilité.

Etudes des resultats

Moyenne

Nous avons calculé la moyenne a chaque lot d'itération.

```
double mean = 0;
long* pop = new long[numberToRun];
long popTotal = 0;

for (int i = 0; i < numberToRun; ++i) {
    pop[i] = runOne(duration, startingCouple, seed);
    popTotal += pop[i];
}

mean = (float)popTotal / (float)numberToRun;
```

Ecart type

Nous avons également calculé l'écart type a chaque lot d'itération.

```
double s = 0;

for(int i = 0; i < nbExperiments; ++i) {
    s += pow(pop[i] - mean, 2);
}
s = s / (nbExperiments - 1);
```

Intervalles de confiance

Nous avons finalement calculé l'intervalle de confiance grâce aux deux données statistiques précedentes.
Pour cela, nous utilisons Student pour compenser l'approximation de l'écart type.

```
double t = 0, r = 0;
double* interval = new double[2] {0};

 if (nbExperiments <= 30) {
    t = STUDENT[nbExperiments - 1];
} else if(nbExperiments < 40) {
    t = STUDENT[30];
} else if(nbExperiments < 80) {
    t = STUDENT[31];
} else if(nbExperiments < 120) {
    t = STUDENT[32];
} else {
    t = STUDENT[33];
}

r = t * sqrt(s / nbExperiments);

interval[0] = mean - r;
interval[1] = mean + r;
```

Nous avons fait tourne 120 simulations avec une population de depart de 10 elements et pendant 6 ans (seed de depart: `999999`). Cela nous a permis de calculer un intervalle de confiance suivant:

Intervalle de confiance à 95%:
[562243 ; 656668]

Comme prévu, la méthode n'est pas concluante. Cela vient peut être du fait que la fonction n'est pas une loi normale.

Evolution de la population

Nos resultats sont présentés dans un fichier PDF separé.
Comme attendu, on observe une courbe exponentielle. De plus, on remarque que un aléatoire favorable au debut semble plus favorable qu'un aléatoire favorable vers la fin.