Alexis DRAI
22001161

24/03/2022
Y2_SIMU

# Lab #4

# Croissance de populations



## FIGURES



## Introduction

Le code présenté dans ce rapport ignorera certains aspects de la programmation, comme la déclaration de certaines variables et le détail de certaines fonctions annexes, pour se concentrer sur la logique du modèle. Le code complet est bien sûr fourni (et visible publiquement sur GitHub ici, à partir du 05/04/2022).

La Javadoc est téléchargeable via GitHub ici (vous pouvez ensuite ouvrir "index.html" dans un navigateur de votre choix et ainsi parcourir les fichiers en local).

On expérimente avec différentes simulations de croissance de population. Il s'agit dans un premier temps de modéliser une population de lapins qui croît en suivant la suite de Fibonacci. On expérimente ensuite avec des règles de reproduction plus réalistes. On s'appuie sur la programmation orientée objet, en Java, pour un programme plus robuste et lisible. On utilise plusieurs réplications pour avoir des intervalles de confiances supérieurs à 99% et informatifs.

## Une simulation rapide de croissance d'une population de lapins

La première approche proposée du problème est en fait d'utiliser la suite de Fibonacci pour déterminer le nombre de couples de lapins présents pour un mois donné, en commençant au mois n°1. La classe `SimpleRabbitSim` a une seule méthode statique : `popByMonth()`. On suit les étapes suggérées dans les slides du cours (voir commentaires en violet). Nos lapins sont immortels et produisent douze couples par an à partir de leur deuxième mois de vie.

```
public class SimpleRabbitSim
{
    static long popByMonth(int months)
    {
        long m_1, m_2, current = 0;
        // step1: take a couple of baby rabbits
        // step2: wait a month, the couple is now mature for reproduction
        m_1 = m_2 = 1;
        if (months <= 0) throw new IllegalArgumentException();
        if (months == 1 || months == 2)
        {
            System.out.println("Month: " + months + " | Pop: " + 1);
            return 1;
        }
        for (int i = 3; i <= months; i++)
        {
            // step3: apply the fibonacci formula
            current = m_1 + m_2;
            m_2 = m_1;
            m_1 = current;
        }
        // step4: display
        System.out.println("Month: " + months + " | Pop: " + current);
        return current;
    }
}
```

Dans le `main()`, on essaye de lancer l'expérience jusqu'à ce que le nombre de couples de lapins dépasse le nombre d'humains sur Terre en 2022. Ce choix est arbitraire. Cette simulation donne des résultats déterministes, et il n'y a donc pas d'intérêt à faire des réplications multiples.

```
while (pop < 8_000_000_000L)
{
    pop = SimpleRabbitSim.popByMonth(++i);
}
```

On atteint 610 couples au mois 15, et 12 586 269 025 au mois 50. Fibonacci oblige, la population croît de manière exponentielle.

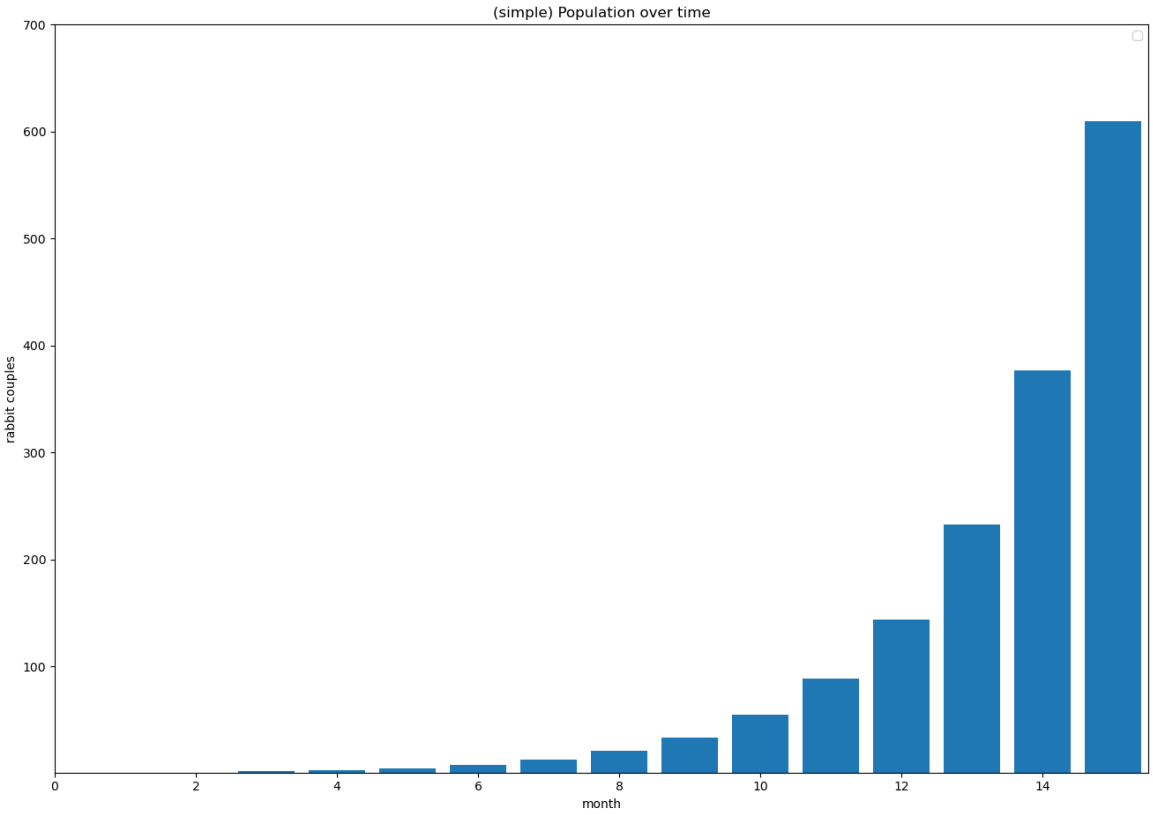


*Figure 1 – population (en couples) en fonction du temps (en mois)*

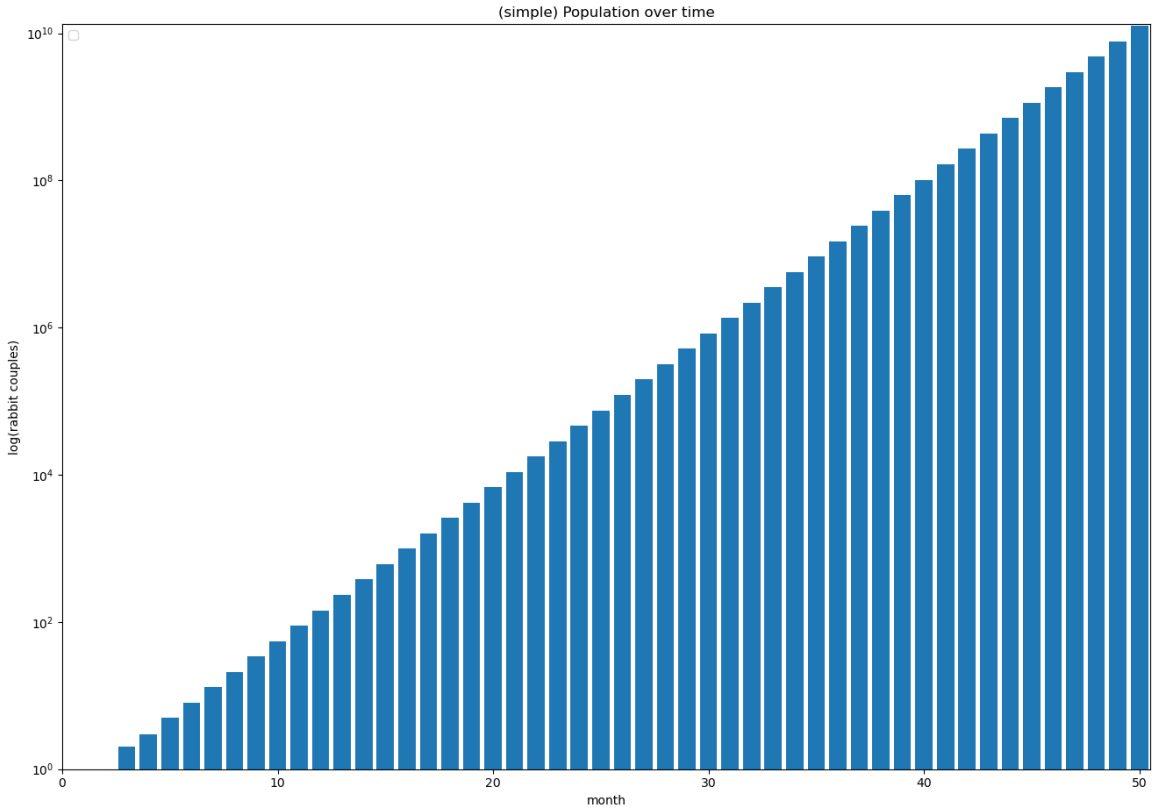


*Figure 2 – ln(population (en couples)) en fonction du temps (en mois)*

# Une simulation de croissance plus réaliste

## Main

On tâche maintenant de réaliser une simulation plus réaliste. La classe `Main` initialise notre `MersenneTwister` en statique et établit quelques constantes publiques pour nos expériences.

```
public final class Main
{
    public static final MersenneTwisterFast MT;
    static
    {
        MT = new MersenneTwisterFast(new int[]{0x123,0x234,0x345,0x456});
    }
    public static final int MONTHS_PER_YEAR      = 12;
    public static final int YEARS_PER_EXPERIMENT = 20;
    public static final int TOTAL_MONTHS         = MONTHS_PER_YEAR
                                                   * YEARS_PER_EXPERIMENT;
```

On décide ensuite du nombre de réplications, grâce à quoi on assigne un *t* de *Student* approprié en préparation pour les calculs statistiques finaux. Notre simulation commence avec 10 femelles, 5 mâles, et sera accomplie 2000 fois, avec un *t* de 2,5759[1]. On prépare nos variables pour les résultats statistiques, on crée un fichier qui contiendra nos résultats, et on lance nos réplications. Pour cela, on instancie un objet de la classe `RabbitModel`, et on appelle `model.run()`.

```
    public static void main(String[] args)
    {
        final int    REPLICATES = 2000;
        final double STUDENT_T  = 2.5759;

        final int MALES   = 5;
        final int FEMALES = 10;

        double mean     = 0;
        double variance = 0;
        double stdDeviation;
        double stdError;
        double errorMargin;
        int[] results  = new int[REPLICATES];

        int tmp;

        for (i = 0; i < REPLICATES; i++)
        {
            RabbitModel model = new RabbitModel(FEMALES, MALES);
            tmp = model.run();
```

Assez tôt dans ce projet, il est apparu qu'il serait intenable de simuler des grands nombres de `Rabbit`s, et notre programme n'admet pas de populations au-delà de `Integer.MAX_VALUE`. Nous montrerons comment prochainement ; mais c'est pourquoi nos résultats sont rangés dans un tableau d'`int`s.

## RabbitModel

Le `RabbitModel` se charge de faire croître et diminuer nos populations. On lui donne donc les constantes statiques qui concernent la prédation et les naissances : 50% des lapins sont femelles, elles donnent le jour à un nombre aléatoire gaussien (à 99% de chances dans [2 ; 6], moyenne de 4) de lapereaux par portée, et ont 15% de chances de mourir en mettant bas.

[1] ces chiffres sont modifiés dans la version soumise, pour améliorer la rapidité des tests du correcteur

```
public final class RabbitModel
{
    private static final double FEMALE_RATIO              = 0.5;
    private static final int    MEAN_KITTEN_PER_LITTER    = 4;
    private static final double STD_DEV_KITTEN_PER_LITTER = 2 / 3.0;
    private static final double DEATH_IN_LABOR_RATE       = 0.15;
```

C'est notamment à l'aide de l'introduction d'un rôle de prédateur basique que nos populations sont contrôlées, et il convient de dédier une courte parenthèse ici aux raisons de ce choix : les lapins se reproduisant bien plus vite qu'ils ne disparaissent, leur nombre augmente toujours de manière exponentielle. Or, ce modèle emploie des objets qu'il faut instancier un à un, et non plus des variables de type primitif. Ce modèle doit aussi parcourir l'ensemble de ces objets à chaque pas (chaque mois). En d'autre termes, le nombre d'opérations coûteuses augmente avec le nombre de lapins présents : le temps d'exécution de chaque itération est de l'ordre de $O(n)$, avec $n$ la taille de la population au début de l'itération. Or, la taille de la population croît de manière exponentielle, et le temps d'exécution du programme entier est donc de $O(2^p)$ avec $p$ le nombre de pas dans la simulation. On choisit de limiter la taille de la population : si elle dépasse un seuil de 50 000, tous les "prédateurs" cachés dans la population deviennent actifs, et tuent un nombre aléatoire gaussien (à 99% de chances dans [500 ; 2500], moyenne de 1500) de lapins chacun tous les mois.

```
    private static final int    PREDATOR_THRESHOLD        = 50_000;
    private static final double MEAN_KILLS                = 1500;
    private static final double STD_DEVIATION_KILLS       = 1000 / 3.0;
```

Toujours pour optimiser la performance, nos objets sont rangés dans un `HashSet` : on peut y insérer, retirer, et vérifier l'existence d'éléments en temps constant ($O(1)$), et le fait que l'ordre d'insertion n'est pas préservé ne nous dérange pas. On store les chiffres liés à la population dans des attributs séparés, pour éviter d'appeler `size()`, et pour avoir une vue plus détaillée.

```
    private final Set<Rabbit> rabbits  = new HashSet<>();
    private long    deaths             = 0;
    private long    births             = 0;
    private int     predators          = 0;
    private boolean arePredatorsActive = false;
```

Le constructeur unique suit les instructions vues dans le `main()`.

```
public RabbitModel(int numFem, int numMal)
{
    for (int i = 0; i < numFem + numMal; i++)
    {
        this.rabbits.add(makeRabbit(i < numFem));
    }
}
```

Il appelle `makeRabbit()` en imposant un sexe. Cette dernière incrémente le nombre de naissances et instancie un lapin du sexe voulu. Si ce lapin est un "prédateur", c'est aussi répertorié.

```
private Rabbit makeRabbit(boolean isFemale)
{
    Rabbit rabbit;
    this.births++;
    if (isFemale)
    {
        rabbit = new FemaleRabbit();
    }
    else
    {
        rabbit = new Rabbit();
    }
    if (rabbit.isRabbitOfCaerbannog())
```

```
    {
        this.predators++;
    }
    return rabbit;
}
```

Notons ici que les lapins femelles appartiennent à une sous-classe de `Rabbit` : `FemaleRabbit`. En effet, les lapins femelles de notre modèle sont simplement des `Rabbit`s avec quelques attributs et méthodes en plus, et on tire avantage de l'héritage et du polymorphisme en faisant ce choix d'implémentation, bien qu'il s'éloigne de la réalité.

## Rabbit

`Rabbit` inclus les constantes statiques concernant la génération aléatoire des chiffres de maturité, de vieillesse, de prédation, et des taux de mortalité annuels et mensuels ; à l'exception des taux de la première année de vie de chaque individu, car ceux-ci dépendent de chiffres calculés pendant l'instanciation. On impose qu'un lapin aie 25% de chances de mourir chaque année, jusqu'à la fin de ses 7 ans, après quoi ces chances augmentent de 15% chaque année jusqu'à sa mort assurée, à la fin de ses 12 ans. Pour un lapereau, c'est 75% par an jusqu'à sa maturité.

```
public class Rabbit
{
    private static final int      EARLIEST_MATURITY_START      = 5;
    private static final int      INTERVAL_SIZE_MATURITY_START = 3;
    private static final int      MAX_AGE_MONTHS               = 156;
    private static final double   CAERBANNOG_RATIO             = 1/(2**14);
    private static final double   KIT_YEARLY_MORTALITY         = 0.75;
    private static final double[] YEARLY_MORTALITIES           = {
// [0;1[ (will be ignored in static monthly mortality rate calculations)
            0.25,     // [1;2[
            0.25,     // [2;3[
            0.25,     // [3;4[
            0.25,     // [4;5[
            0.25,     // [5;6[
            0.25,     // [6;7[
            0.25,     // [7;8[
            0.4,      // [8;9[
            0.55,     // [9;10[
            0.7,      // [10;11[
            0.85,     // [11;12[
            1.0       // [12;13[
    };
    private static final double   KIT_MONTHLY_MORTALITY        =
        Math.pow((1 + KIT_YEARLY_MORTALITY), (1 / 12.0)) - 1;
    private static final double[] MONTHLY_MORTALITIES          =
        new double[MAX_AGE_MONTHS - 12];
```

On convertit ces taux en taux mensuels constants, dès la compilation.

```
    int k = 0;
    for (double yearlyMortality : YEARLY_MORTALITIES)
    {
        for (int i = 0; i < 12; i++)
        {
            double m = Math.pow(1 + yearlyMortality, 1 / 12.0) - 1;
            MONTHLY_MORTALITIES[k++] = m;
        }
    }
```

Un lapereau naît non-mort, non-mature, âgé de 0 mois, le mois de sa maturité est généré aléatoirement avec une distribution continue sur [5 ; 8], et il a 1 chance sur $2^{14}$ d'avoir le rôle caché de prédateur.

```
    private final boolean isRabbitOfCaerbannog;
    private final int     maturityStart;

    private boolean isMature    = false;
    private boolean isDead      = false;
    private int     ageInMonths = 0;

    Rabbit()
    {
        this.maturityStart =
                Main.MT.nextInt(INTERVAL_SIZE_MATURITY_START + 1)
                + EARLIEST_MATURITY_START;
        this.isRabbitOfCaerbannog = Main.MT.nextBoolean(CAERBANNOG_RATIO);
    }
```

## FemaleRabbit

`FemaleRabbit` a tous les attributs et méthodes de Rabbit, et quelques autres liés à la fertilité et à la procréation. On impose que 90% sont fertiles, qu'elles ne restent fertiles que pour 48 mois à partir de leur mois de maturité[2], et mettent bas un nombre aléatoire gaussien (à 99% de chances dans [3 ; 9], moyenne de 6) de portées par an. On emploie un tableau de booléens comme planning annuel de portées, et les portées y sont réparties dynamiquement en fonction du nombre aléatoire annuel.

```
public final class FemaleRabbit extends Rabbit
{
    private static final double FERTILITY_PROB           = 0.9;
    private static final int    INTERVAL_SIZE_FERTILITY  = 48;
    private static final int    MEAN_LITTERS_PER_YEAR    = 6;
    private static final double STD_DEV_LITTERS_PER_YEAR = 1.0;

    private final boolean isPotentiallyFertile;
    private final boolean[12] pregnancyPlanner;
    private boolean isFertile = false;

    FemaleRabbit()
    {
        this.isPotentiallyFertile = Main.MT.nextBoolean(FERTILITY_PROB);
    }
```

Tout cela conclut l'instanciation du `RabbitModel`. Le `main()` appelle alors la fonction `model.run()`.

## RabbitModel.run()

Cette fonction fait passer le modèle par un nombre donné d'itération (on a choisi 240 mois[3], passé en constante globale), enregistre les résultats détaillés dans un fichier .csv, et renvoie le nombre final d'individus. Pour ce faire, on parcourt notre structure de lapins à l'aide d'un itérateur. L'itérateur nous permet de supprimer des élément de l'ensemble tout en continuant à le parcourir.

```
int run()
{
    for (int i = 1; i <= Main.TOTAL_MONTHS; i++)
    {
        Iterator<Rabbit> it = this.rabbits.iterator();
```

[2] contrainte ajoutée par-dessus celles du sujet : on cherche à limiter la croissance de la population, et cela reflète approximativement la réalité.

[3] ces chiffres sont modifiés dans la version soumise, pour améliorer la rapidité des tests du correcteur

On traite chaque lapin ainsi : en premier lieu, s'il est vivant, on le fait vieillir d'un mois. C'est cette fonction `ageUp()`[4] qui met à jour l'âge du lapin, indique s'il est mort et/ou mature, et pour les femelles, déclenche une mise à jour potentielle de son planning de procréation. Ensuite, s'il est mort, on appelle `destroyRabbit()` et on retire l'élément de notre ensemble, avant de passer à l'élément suivant. Retirer les éléments inactifs dès que possible améliore les performances.

```
        while (it.hasNext())
        {
            Rabbit rabbit = it.next();
            if (!rabbit.isDead())
            {
                rabbit.ageUp();
            }
            if (rabbit.isDead())
            {
                this.destroyRabbit(rabbit);
                it.remove();
                continue;
            }
```

On traite alors les lapins femelles qui ont une portée due ce mois-ci.

```
            if
            (
                rabbit instanceof FemaleRabbit &&
                FemaleRabbit)rabbit).getPregnancyPlanner()[
                    rabbit.getAgeInMonths() % Main.MONTHS_PER_YEAR
                ]
            )
```

Si elles meurent en mettant bas, on appelle `destroyRabbit()` et on retire l'élément de notre ensemble, avant de passer à l'élément suivant. Les lapereaux ne sont pas instanciés : on suppose qu'ils meurent également.

```
            {
                if (Main.MT.nextBoolean(DEATH_IN_LABOR_RATE))
                {
                    this.destroyRabbit(rabbit);
                    it.remove();
                    continue;
                }
```

Si elles survivent, on génère un nombre aléatoire gaussien décrit plus haut, et on instancie ce nombre de nouveaux objets. Ces lapereaux sont placés dans une `List` annexe, `toAdd`, et seront intégrés à `rabbits` entre chaque parcours.

```
                else
                {
                    int kitten = (int) Math.round(Main.MT.nextGaussian()
                                    * STD_DEV_KITTEN_PER_LITTER
                                    + MEAN_KITTEN_PER_LITTER);
                    for (int j = 0; j < kitten; j++)
                    {
                        this.toAdd.add(this.makeRabbit());
                    }
                }
            }
```

[4] voir `Rabbit.checkDead()` et `FemaleRabbit.updatePregnancyPlanner()` pour les détails d'implémentation : le détail des fonctions annexes est visible dans la *Javadoc* (et bien sûr, dans le code).

Quel que soit son sexe, on vérifie si la population a dépassé le seuil assigné, et on active les prédateurs le cas échéant.

```
            if (this.getPop() >= PREDATOR_THRESHOLD)
            {
                this.setPredatorsActive(true);
            }
        }
```

Pour éviter les modifications concurrentes, c'est ici, en dehors du parcours du `Set`, qu'on insère les lapereaux de ce mois.

```
        this.rabbits.addAll(this.toAdd);
        this.toAdd.clear();
```

Enfin, si les prédateurs sont activés, on les fait travailler avec `cull()`.

```
        if (this.arePredatorsActive())
        {
            this.cull();
        }
    }
```

À la fin du nombre de mois demandé, on peut renvoyer le total final de population.

```
    return this.getPop();
}
```

`cull()` permet d'enlever un grand nombre de lapins. On contrôle si on est toujours au-dessus du seuil indiqué (on désactive les prédateurs et sort de la fonction dans le cas contraire) puis on détruit un nombre aléatoire gaussien de lapins pour chaque prédateur (à 99% de chances dans [500 ; 2500], moyenne de 1500), suite à quoi on les supprime de notre ensemble à l'aide d'un itérateur. Puisqu'un `HashSet` présente ses éléments dans le désordre, on peut détruire des lapins au hasard en appelant simplement `remove()` successivement.

```
private void cull()
{
    if (this.getPredators() > 0)
    {
        for (int i = 0; i < this.getPredators(); i++)
        {
            if (this.getPop() < PREDATOR_THRESHOLD)
            {
                this.setPredatorsActive(false); return;
            }
            int j = 0;
            int kills = (int) Math.round(Main.MT.nextGaussian()
                                         * STD_DEVIATION_KILLS
                                         + MEAN_KILLS);
            Iterator<Rabbit> it = this.rabbits.iterator();
            while (j < kills && it.hasNext())
            {
                this.destroyRabbit(it.next());
                it.remove();
                j++;
            }
        }
    }
}
```

## Conclusion : Résultats, interprétation

La variance est très élevée, comme le suggèrent les trois réplications ci-dessous ; et la moyenne de population finale est artificiellement tirée vers le seuil arbitraire d'activation des prédateurs.

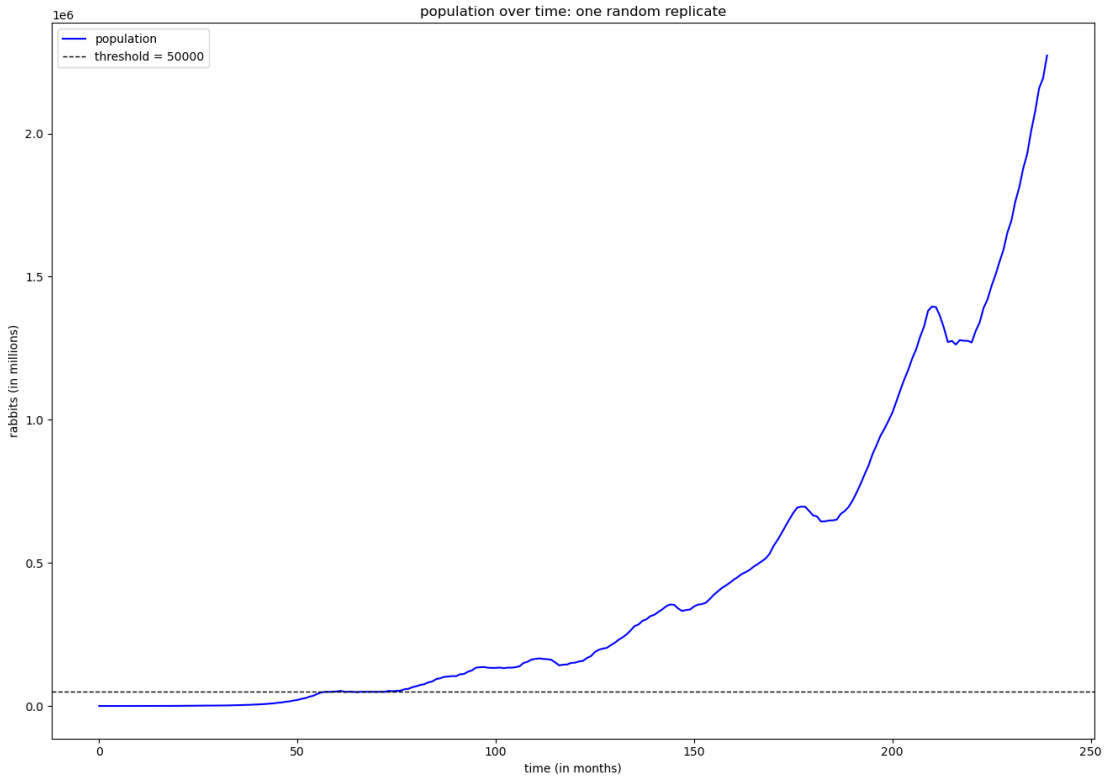


*Figure 3 – une réplication au hasard (env. 2.2M)*

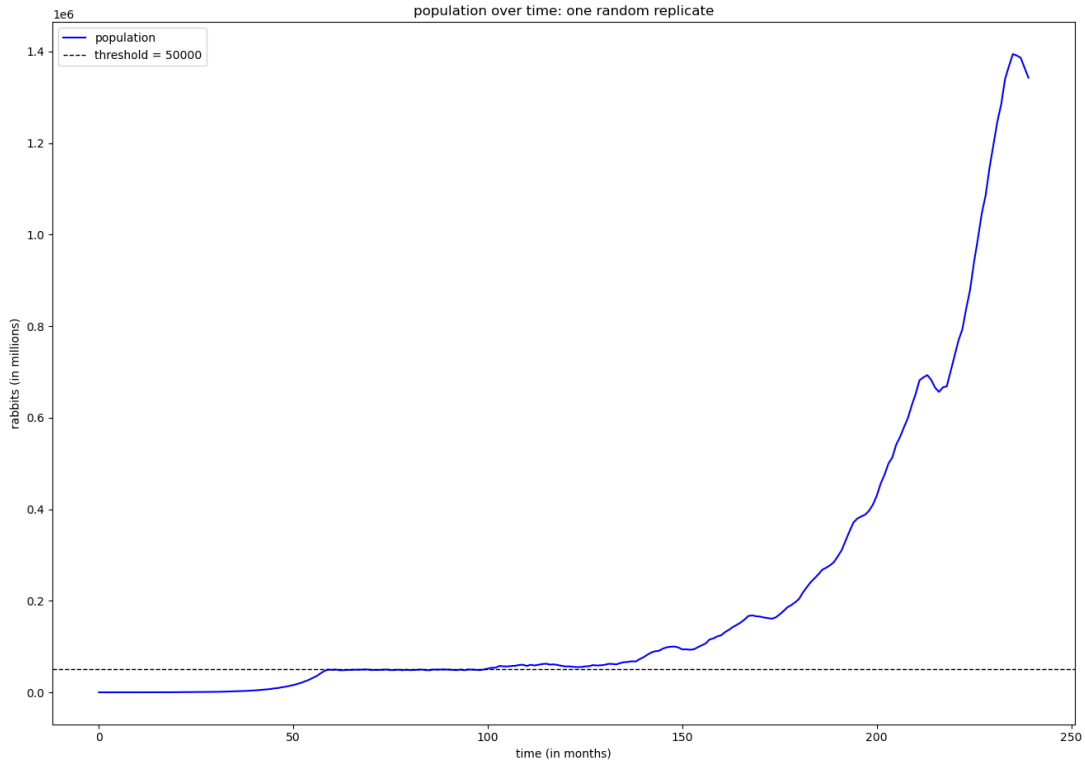


*Figure 4 – une réplication au hasard (env. 1.4M)*

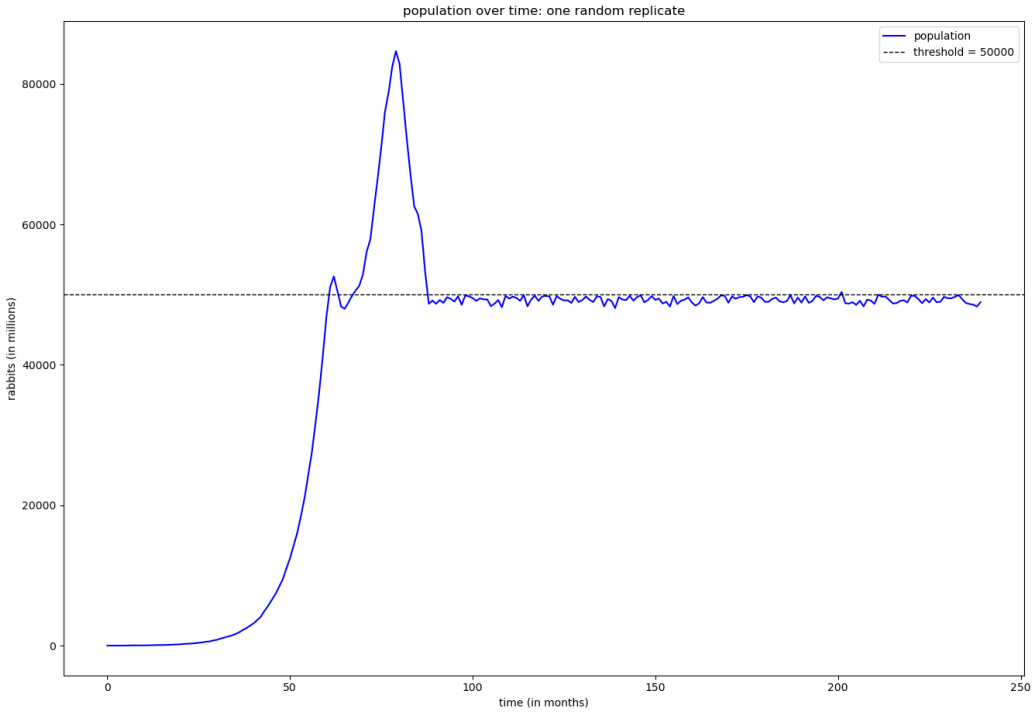


*Figure 5 – une réplication au hasard (env. 50k)*

Les résultats obtenus sont détaillés ci-dessous.

```
after 2000 replicates of a 20-year-long experiment
with 10 female and 5 male starting rabbits
observed population levels were such:

mean                            = 2_852_640.034
variance                        = 7_454_020_340_374.987
standard deviation              = 2_730_205.183
standard error                  = 61_049.244
margin of error                 = 157_256.747
99% confidence interval         = [2_695_383.287, 3_009_896.781]
```

Grâce à nos paramètres ajoutés, et notamment grâce à la prédation, la courbe de croissance de la population est quelque peu aplatie. C'est, en partie, ce qui nous a permit de faire 2000 réplications en moins de 48 heures.

La moyenne estimée est de 2 852 640, avec un intervalle de confiance à 99% de [2 695 383 ; 3 009 896]. On observe, dans l'histogramme plus bas, que plus de 20% des réplications nous donnent des populations finales dans l'intervalle [0 ; 200 000].

C'est une opportunité de nous rappeler ce qu'un intervalle de confiance nous indique vraiment : il ne nous dit pas que 99% des réplications atterriront dans l'intervalle de confiance, mais plutôt qu'on estime que la vraie moyenne théorique a 99% de chances de se trouver dans cet intervalle.

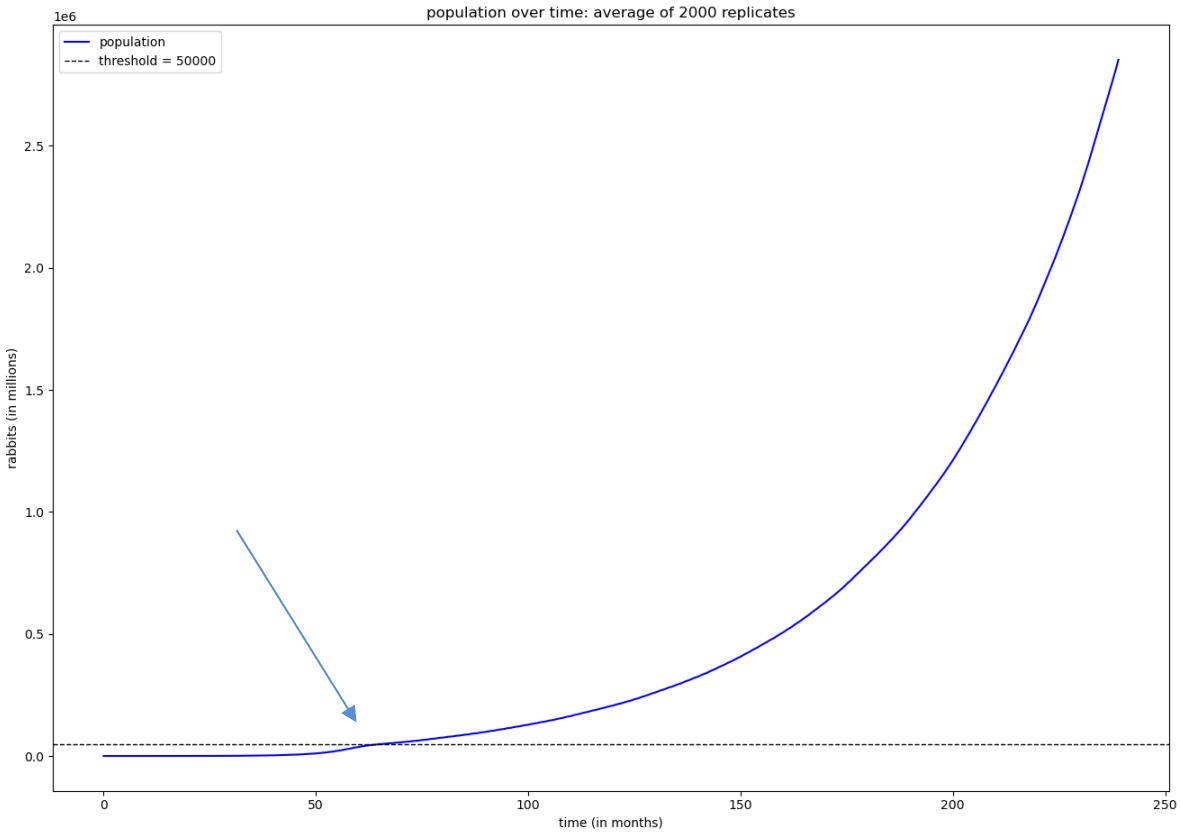


*Figure 6 – population moyenne sur 2000 réplications*

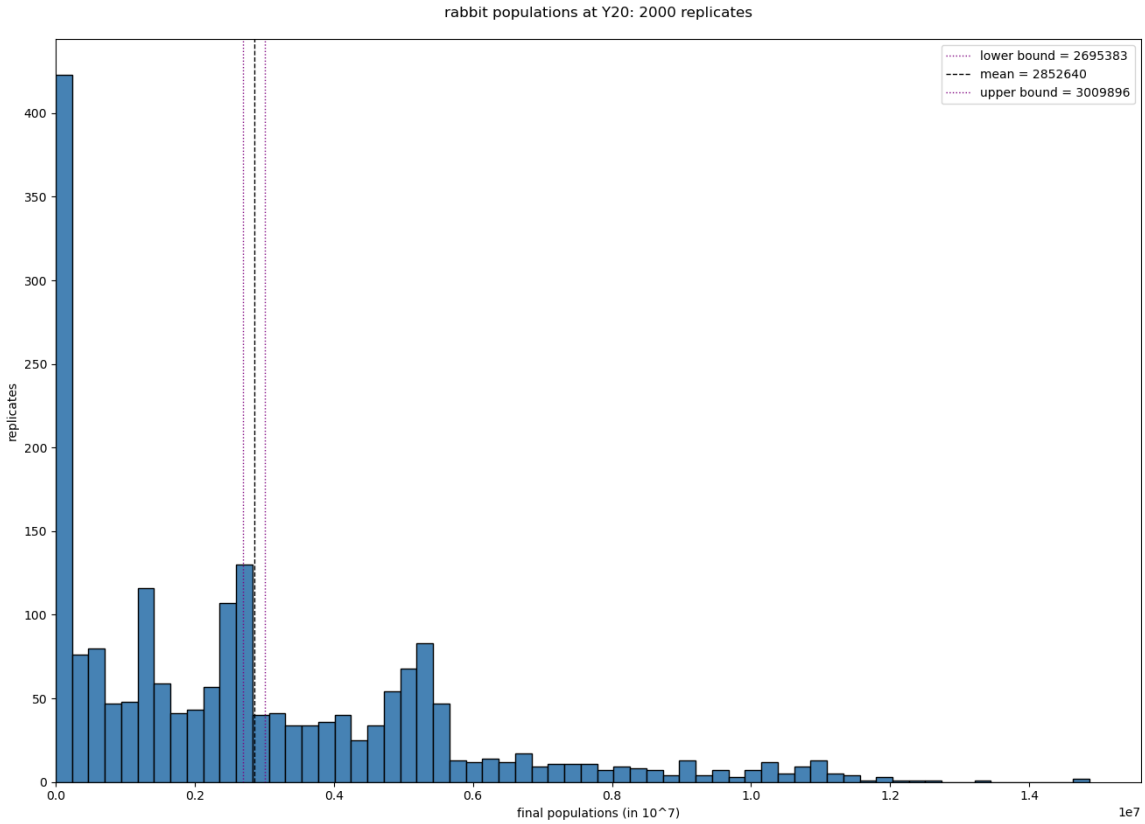


*Figure 7 – comptes finaux sur 2000 réplications*