Simulation de Population de Lapins en Langage C

EL ALLALI Achraf

15 janvier 2026

Résumé

Ce rapport présente une analyse détaillée du code C implémentant une simulation de population de lapins. Nous expliquerons la structure du programme, les algorithmes utilisés, les choix de conception importants. Enfin, nous analyserons les résultats obtenus à travers les graphes générés par le script Python d'analyse.

Table des matières

1 Introduction

La simulation de population de lapins est un modèle informatique qui simule l'évolution d'une population de lapins sur plusieurs mois. Le programme est écrit en langage C pour des raisons de performance et utilise des générateurs de nombres aléatoires pour modéliser les événements stochastiques tels que la naissance, la mort et la reproduction.

Le code principal se trouve dans les fichiers `main.c`, `rabbitsim.c` et `rabbitsim.h`. Il utilise la bibliothèque PCG pour la génération de nombres pseudo-aléatoires et implémente différents modèles de taux de survie.

2 Structure du Code C

2.1 Les Structures de Données

Le programme utilise plusieurs structures de données pour représenter les lapins et l'état de la simulation.

2.1.1 Structure s_rabbit

La structure `s_rabbit` représente un lapin individuel :

```c
typedef struct rabbit {
    int sex;                    // 0 pour femelle, 1 pour mâle
    int status;                 // 0 mort, 1 vivant
    int age;                    // Age en mois
    int mature;                 // 0 immature, 1 mature
    int maturity_age;           // Age de maturation
    int pregnant;               // 0 non enceinte, 1 enceinte
    int nb_litters_y;           // Nombre de portées par an
    int nb_litters;             // Nombre de portées cette année
    float survival_rate;        // Taux de survie
    int survival_check_flag;    // Indicateur de vérification
} s_rabbit;
```

Chaque lapin possède des caractéristiques biologiques et un état dans la simulation.

2.1.2 Structure s_simulation_instance

Cette structure gère l'ensemble de la simulation :

```c
typedef struct {
    s_rabbit *rabbits;          // Tableau dynamique de lapins
    size_t rabbit_count;        // Nombre actuel de lapins
    size_t dead_rabbit_count;   // Nombre de lapins morts
    size_t rabbit_capacity;     // Capacité allouée
    int *free_indices;          // Indices libres pour réutilisation
    size_t free_count;          // Nombre d'indices libres
    int sex_distribution[2];    // Distribution des sexes
    s_monthly_stats *monthly_data;  // Données mensuelles
    int monthly_data_capacity;
    int monthly_data_count;
    int deaths_this_month;
    int births_this_month;
} s_simulation_instance;
```

2.2 Gestion de la Mémoire Dynamique

2.2.1 Tableaux Dynamiques vs Listes Chaînées

Le programme utilise des tableaux dynamiques (`realloc`) plutôt que des listes chaînées pour stocker les lapins. Ce choix présente plusieurs avantages :

- **Accès direct O(1)** : L'accès à un lapin par son indice est instantané, contrairement aux listes chaînées qui nécessitent un parcours O(n).
- **Locality of reference** : Les lapins sont stockés de manière contiguë en mémoire, améliorant les performances du cache CPU.
- **Simplicité d'itération** : Les boucles sur tous les lapins sont plus simples et efficaces.
- **Gestion des indices libres** : Au lieu de supprimer physiquement les lapins morts, le programme maintient un tableau d'indices libres pour réutiliser les emplacements, évitant les reallocations fréquentes.

Cependant, ce choix présente un inconvénient : l'insertion/suppression peut nécessiter des déplacements de mémoire. Mais dans ce contexte de simulation où les lapins sont principalement ajoutés et rarement supprimés individuellement, c'est optimal.

2.2.2 Fonction ensure_capacity

La fonction `ensure_capacity` gère la croissance du tableau :

```
int ensure_capacity(s_simulation_instance *sim) {
    if (sim->rabbit_count < sim->rabbit_capacity)
        return 1;
    size_t new_capacity = (sim->rabbit_capacity == 0) ?
       INIT_RABIT_CAPACITY : sim->rabbit_capacity * 1.3;

    s_rabbit *temp_rabbits = realloc(sim->rabbits, sizeof(s_rabbit) *
       new_capacity);
    if (!temp_rabbits)
        return 0;
    sim->rabbits = temp_rabbits;

    int *temp_indices = realloc(sim->free_indices, sizeof(int) *
       new_capacity);
    if (!temp_indices)
        return 0;
    sim->free_indices = temp_indices;

    sim->rabbit_capacity = new_capacity;
    return 1;
}
```

La capacité augmente à chaque fois, assurant une complexité amortie O(1) pour les ajouts.

2.3 Génération de Nombres Aléatoires

Le programme utilise la bibliothèque PCG (Permuted Congruential Generator) pour la génération de nombres pseudo-aléatoires :

```c
double genrand_real(pcg32_random_t *rng) {
    return (double)pcg32_random_r(rng) / (double)UINT32_MAX;
}
```

2.3.1 Choix de PCG au lieu de Mersenne Twister (MT19937)

La raison principale du choix de PCG (Permuted Congruential Generator) plutôt que l'algorithme Mersenne Twister (MT19937) réside dans la nécessité de lancer des simulations en parallèle. En effet, le programme utilise OpenMP pour exécuter plusieurs simulations simultanément, et PCG offre des avantages cruciaux dans ce contexte :

- **Thread-safety et parallélisation** : Contrairement à MT19937 qui nécessite une synchronisation complexe pour éviter les conflits entre threads, PCG permet à chaque thread d'avoir son propre générateur indépendant avec un état minimal (seulement 8 octets). Cela élimine les goulots d'étranglement liés à la synchronisation et permet une parallélisation efficace.
- **Performance supérieure** : PCG est généralement plus rapide que MT19937, avec un temps de génération par nombre plus court. Cela est crucial pour les simulations nécessitant des millions de nombres aléatoires, particulièrement lorsqu'elles sont exécutées en parallèle.
- **Consommation mémoire réduite** : Avec un état interne de seulement 8 octets contre 2500 octets pour MT19937, PCG permet de créer facilement des instances séparées pour chaque thread sans impact mémoire significatif, facilitant ainsi l'exécution de multiples simulations parallèles.
- **Qualité statistique** : PCG offre d'excellentes propriétés statistiques, surpassant souvent MT19937 dans les tests de suites aléatoires. Sa période est très longue (2^{64} pour PCG32), comparable à celle de MT19937 ($2^{19937-1}$).
- **Modernité et sécurité** : PCG est un algorithme plus récent (2014) que MT19937 (1997), bénéficiant des avancées en cryptographie et en théorie des nombres.

Dans le contexte de cette simulation où plusieurs instances de simulation s'exécutent en parallèle (via `multi_simulate`), le choix de PCG permet non seulement des performances optimales mais aussi une isolation parfaite entre les threads, garantissant la reproductibilité et la qualité statistique de chaque simulation individuelle.

2.4 Modèles de Taux de Survie

Le programme implémente trois méthodes de calcul des taux de survie :

1. **Statique** : Utilise des valeurs constantes définies.
2. **Gaussienne** : Ajoute une variation normale autour de la valeur de base.
3. **Exponentielle** : Utilise une distribution exponentielle.

```c
float calculate_survival_rate_gaussian(float base_rate, pcg32_random_t *
    rng) {
    // Implémentation avec Box-Muller pour la distribution normale
    double u1 = genrand_real(rng);
    double u2 = genrand_real(rng);
    double z0 = sqrt(-2.0 * log(u1)) * cos(2.0 * M_PI * u2);
    return base_rate + 5.0f * z0; // Variation de +-5%
}
```

2.5 Boucle Principale de Simulation

La fonction `simulate` constitue le cœur de la simulation :

1. Initialisation de la population
2. Boucle sur chaque mois :
   — Mise à jour de l'âge des lapins
   — Vérification de la maturation
   — Calcul des taux de survie
   — Vérification de la survie (morts)
   — Gestion de la reproduction
   — Naissances
   — Collecte des statistiques
3. Retour des résultats

2.6 Gestion de la Reproduction

La reproduction suit un modèle réaliste :
— Les femelles matures peuvent être enceintes
— Nombre de portées par an généré aléatoirement
— Chaque portée contient un nombre de lapereaux basé sur Fibonacci
— Les nouveau-nés ont un taux de survie initial différent

```
int give_birth(s_simulation_instance *sim, size_t i, pcg32_random_t *rng
    ) {
     int nb_new_born = fibonacci(sim->rabbits[i].nb_litters);
     create_new_generation(sim, nb_new_born, rng);
     return nb_new_born;
}
```

3 Choix d'Optimisation

3.1 Script boost_task.sh

Le script `boost_task.sh` optimise l'exécution du programme en configurant le système pour des performances maximales :

```
# Réglage du gouverneur CPU sur performance
echo performance | sudo tee $cpu/cpufreq/scaling_governor > /dev/null

# Priorité élevée (nice -10)
sudo nice -n -10 ...

# Priorité I/O en temps réel
ionice -c 1 -n 0 ...

# Utilisation de tous les coeurs CPU
taskset -c 0-$((CORES-1)) "$PROGRAM" $ARGS
```

Ces optimisations sont cruciales pour les simulations longues avec de grandes populations, où les performances CPU et I/O peuvent devenir des goulots d'étranglement.

3.2 Parallélisation

Le programme utilise OpenMP pour paralléliser les multiples simulations :

```
#define NUM_THREADS 1
#pragma omp parallel for num_threads(NUM_THREADS)
for(int sim = 0; sim < nb_simulations; sim++) {
    // Simulation individuelle
}
```

Bien que NUM_THREADS soit défini à 1, la structure permet une parallélisation facile.

4 Résultats et Analyse

4.1 Configuration des Simulations

Les simulations ont été exécutées avec les paramètres suivants :
- Population initiale : 3 lapins
- Durée : 80 mois
- Nombre de simulations : 5
- Méthode de survie : Statique

4.2 Analyse des Graphes

4.2.1 Population au Cours du Temps

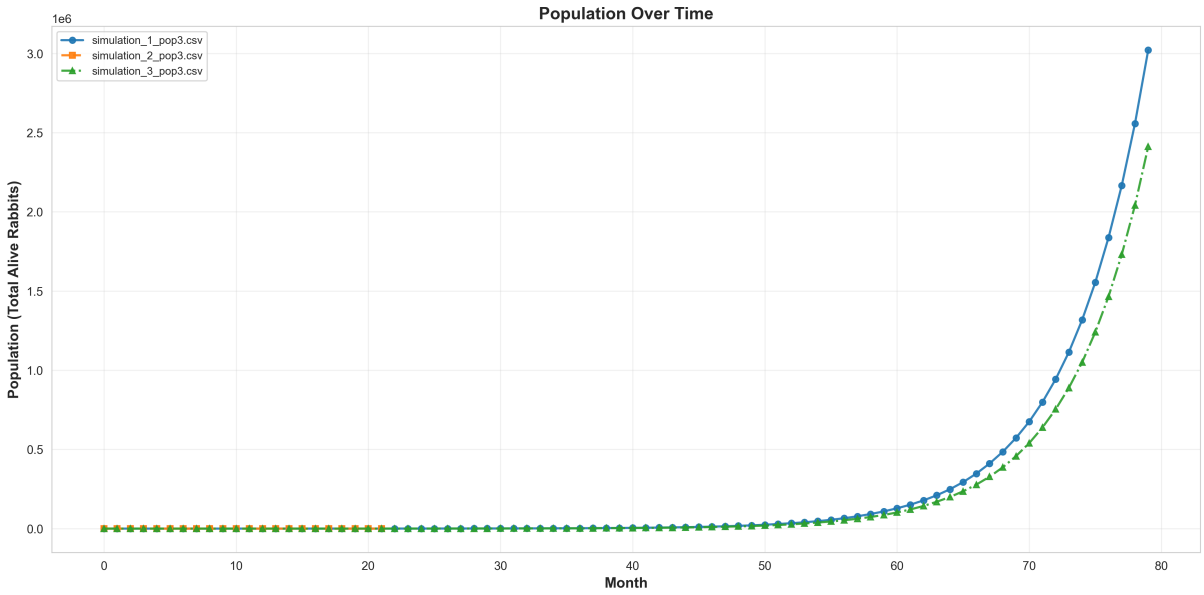

FIGURE 1 – Évolution de la population totale au cours du temps pour chaque simulation

Ce graphique illustre l'évolution de la population de lapins sur une période de 80 mois pour différentes conditions de simulation. On observe une phase initiale de croissance lente, suivie d'une augmentation très rapide de la population, de type exponentiel. Les

différences entre les courbes traduisent l'influence des paramètres du modèle sur la dynamique de croissance, certaines simulations conduisant à une population finale plus élevée que d'autres.

4.2.2 Taux de Croissance

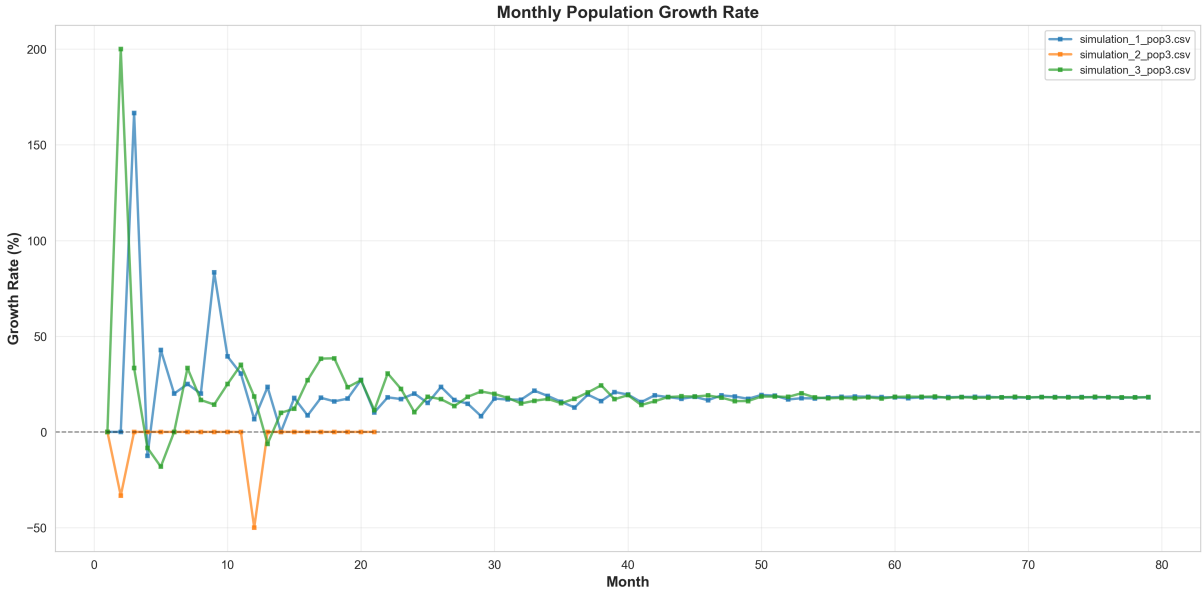

FIGURE 2 – Taux de croissance mensuel de la population

Ce graphique présente l'évolution du taux de croissance mensuel de la population au cours du temps pour les différentes simulations. On observe de fortes fluctuations durant les premiers mois, traduisant une phase transitoire du système. Par la suite, le taux de croissance se stabilise autour d'une valeur positive constante, indiquant une croissance régulière de la population. Les valeurs positives correspondent à une augmentation de la population, tandis que les valeurs négatives indiquent une diminution temporaire.

4.2.3 Diagramme de Phase

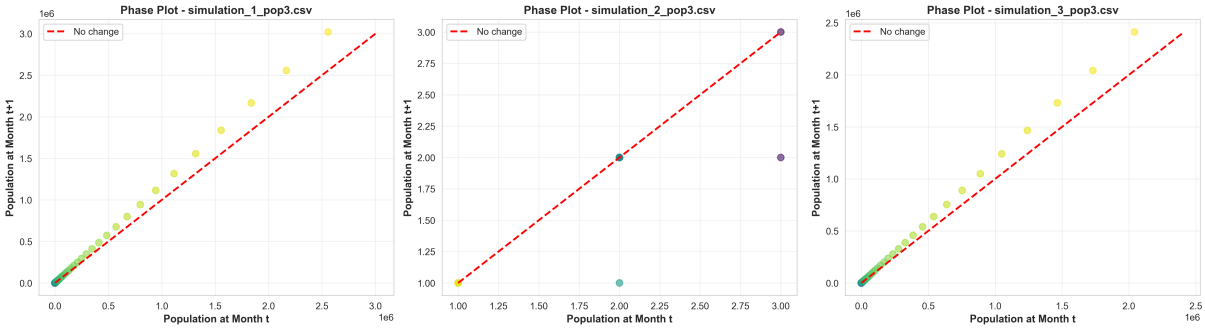

FIGURE 3 – Diagramme de phase : population au mois t vs t+1

Le diagramme de phase illustre la relation entre la population à un mois donné (t) et celle du mois suivant (t + 1). La diagonale rouge représente une population constante

d'un mois à l'autre. Les points situés au-dessus de cette diagonale indiquent une croissance de la population, tandis que ceux situés en dessous traduisent une décroissance. L'alignement progressif des points met en évidence la convergence du système vers un régime de croissance stable.

4.2.4 Structure de la Population

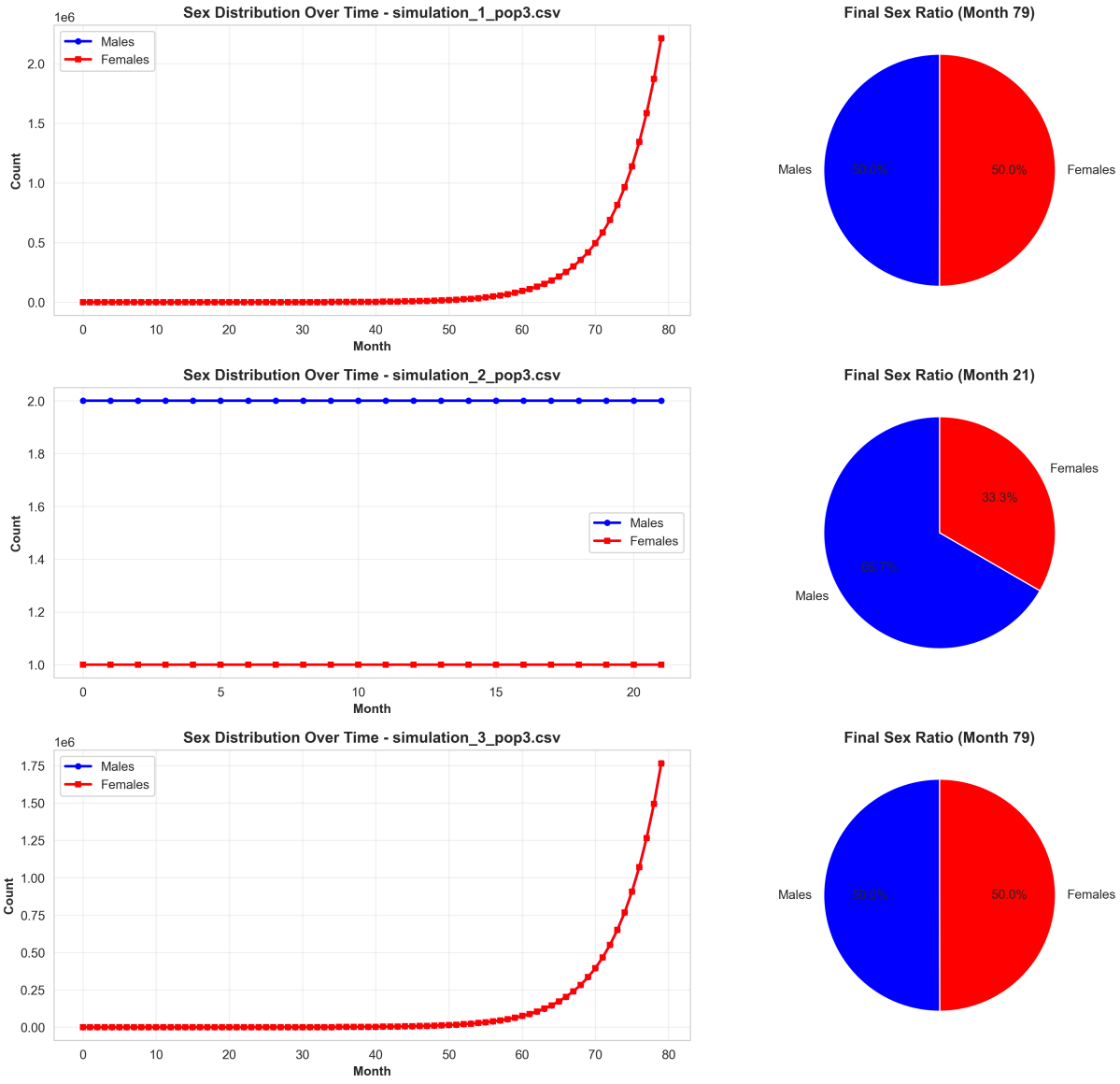

FIGURE 4 – Distribution des sexes au cours du temps et ratio final

Cette visualisation montre l'équilibre entre mâles et femelles, crucial pour la reproduction.

4.2.5 Naissances vs Décès

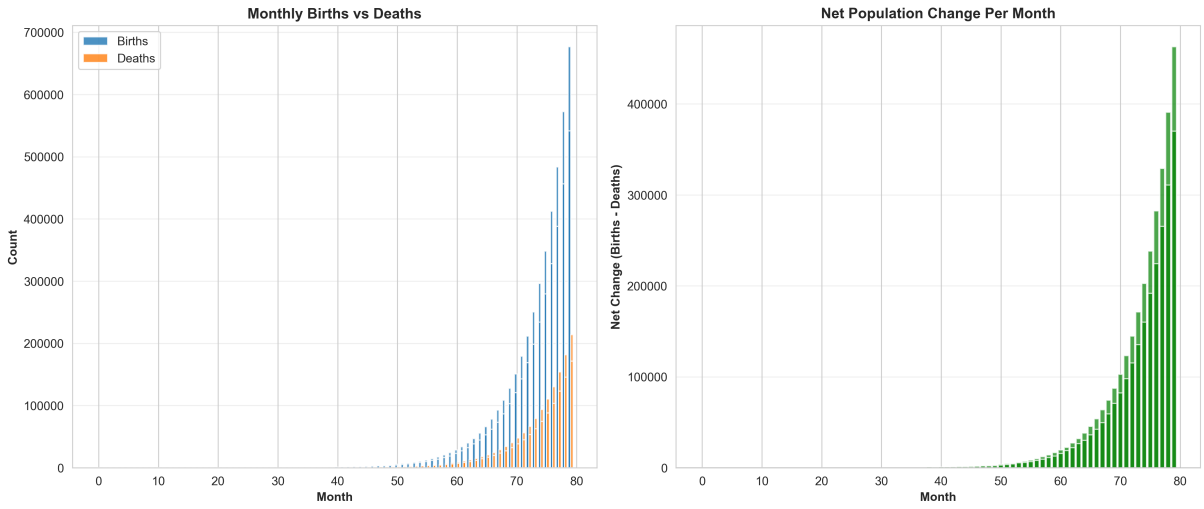

FIGURE 5 – Naissances et décès mensuels, et changement net

Le graphique révèle les dynamiques de reproduction et mortalité. Un excès de naissances sur les décès entraîne une croissance.

4.2.6 Résultats Finaux

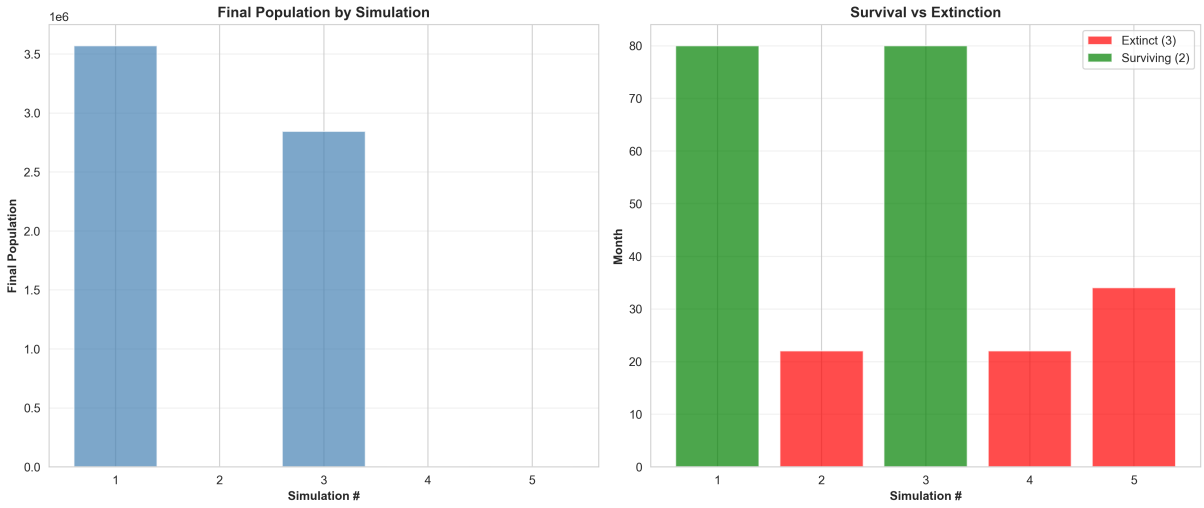

FIGURE 6 – Population finale et survie vs extinction

Ce graphique compare les résultats finaux des différentes simulations en termes de population finale et d'issue du système. Le diagramme de gauche présente la population atteinte à la fin de chaque simulation, tandis que celui de droite distingue les simulations menant à une survie de la population de celles conduisant à une extinction. Les résultats mettent en évidence l'influence des paramètres du modèle sur le devenir de la population, certaines configurations permettant une population finale élevée, tandis que d'autres aboutissent à l'extinction.

4.3 Interprétation des Résultats

Les simulations mettent en évidence plusieurs comportements clés de la dynamique de la population :

- La population connaît une croissance rapide durant les premiers mois, principalement due à un taux de reproduction élevé lorsque les ressources ne sont pas limitantes.
- Des taux de survie constants, lorsqu'ils sont insuffisants pour compenser la mortalité, peuvent conduire à une extinction de la population à long terme.
- L'équilibre des sexes est globalement maintenu au cours du temps, ce qui permet une reproduction continue tant que la population ne devient pas trop faible.
- Les variations stochastiques du modèle introduisent une incertitude dans l'évolution de la population et peuvent provoquer des extinctions prématurées, même dans des conditions initialement favorables.

5 Ressources

Cette section liste les ressources et références utilisées dans le développement et l'analyse de cette simulation.

5.1 Générateurs de Nombres Aléatoires

- **PCG (Permuted Congruential Generator)** : Générateur pseudo-aléatoire performant et de haute qualité utilisé pour les événements stochastiques du modèle.
- **MT19937 (Mersenne Twister)** : Alternative classique pour la génération de nombres aléatoires avec une période très longue.

5.2 Documentation et Outils

- **The C Programming Language** : Kernighan and Ritchie - Référence fondamentale pour le développement en C.
- **Makefile** : Outil de compilation automatisée pour la gestion des dépendances et la construction du projet.
- **Python 3** : Utilisé pour l'analyse des résultats et la génération des graphiques via le script `analyze_simulation.py`.
- **Matplotlib** : Bibliothèque Python pour la visualisation des données de simulation.
- **GNU C Compiler (GCC)** : Compilateur utilisé pour générer l'exécutable C.

5.3 Modèles Mathématiques

- **Dynamique des Populations Biologiques** : Modèles de croissance exponentielle et logistique appliqués à la simulation.
- **Processus Stochastiques** : Utilisation de probabilités pour simuler les événements aléatoires (naissances, décès, reproduction).
- **Modèles de Taux de Survie** : Taux constants et taux dépendants de l'âge pour modéliser la mortalité réaliste.

5.4 Fichiers du Projet

- **main.c** : Point d'entrée du programme et interface de simulation.
- **rabbitsim.c et rabbitsim.h** : Implémentation du moteur de simulation et des structures de données.
- **pcg_basic.c et pcg_basic.h** : Implémentation du générateur PCG.
- **mt19937ar.c et mt19937ar.h** : Implémentation du Mersenne Twister.
- **Makefile** : Fichier de configuration pour la compilation.
- **analyze_simulation.py** : Script d'analyse et de visualisation des résultats.

6 Conclusion

Ce programme C implémente une simulation réaliste de population de lapins utilisant des techniques efficaces de gestion mémoire et des modèles stochastiques appropriés. Les choix de conception comme les tableaux dynamiques et l'optimisation système permettent des simulations performantes de grandes populations.

Les résultats montrent des dynamiques complexes typiques des populations biologiques, avec croissance initiale suivie de stabilisation ou extinction.