TP4 - ZZ2 F2

Saad Amal
Abdelkader Kantaoui
Prof: David HILL
Date: December 20, 2024

Abstract

Ce rapport examine la croissance de la population de lapins à partir d'une population initiale sous deux modèles distincts. Le premier modèle, développé dans le Lab 1, est une approche déterministe simplifiée basée sur la suite de Fibonacci. Il suppose que les lapins ne meurent pas et se reproduisent continuellement, avec une maturation des descendants en un mois. Bien que simple et nécessitant peu de discussion, ce modèle sert de comparaison de base.

En revanche, le Lab 2 introduit un modèle stochastique plus réaliste mis en œuvre en utilisant le modèle de conception Model-View-Controller (MVC) en C++. Ce modèle prend en compte des variables telles que le sexe, l'âge et les taux de mortalité, avec une simulation basée sur les individus où chaque lapin a des caractéristiques uniques influençant la dynamique de la population. Les résultats de ces simulations sont traités et transférés vers Python pour une visualisation et une analyse avancées.

Tout au long de ce laboratoire, nous explorons les choix de modélisation, les techniques d'optimisation et interprétons les résultats, en abordant les anomalies et les perspectives, y compris les biais de perception tels que le 'biais de survie'. De plus, nous étudions l'impact de l'introduction de contraintes telles que des taux de survie variables et des conditions de reproduction. Ces ajustements fournissent une compréhension plus approfondie du comportement de la population dans des scénarios plus complexes et réalistes.

Le rapport met en évidence l'intégration de C++ pour une simulation robuste et de Python pour la présentation des résultats, démontrant l'importance de combiner ces outils dans les études de simulation stochastique.

Contents

1 Introduction

L'étude de la dynamique des populations a été un sujet de fascination et d'utilité dans divers domaines, de l'écologie à la conception d'algorithmes. Dans ce rapport, nous nous concentrons sur la modélisation de la croissance de la population de lapins dans des conditions de plus en plus complexes. Cette exploration commence par un modèle déterministe fondamental et s'étend à une simulation stochastique basée sur des individus. En simulant et en analysant les comportements de la population, nous visons à combler le fossé entre les modèles théoriques simplistes et les scénarios réalistes qui tiennent compte des facteurs biologiques et environnementaux.

1.1 Contexte

Les origines de cette étude remontent à Leonardo de Pise (Fibonacci), qui a proposé un modèle de croissance simplifié basé sur des conditions idéalisées. Ce modèle déterministe, bien que élégant, suppose une absence de mortalité et une reproduction indéfinie, entraînant une croissance exponentielle. En revanche, les populations réelles sont influencées par des facteurs tels que le vieillissement, la répartition par sexe, les taux de mortalité et les contraintes environnementales. Les outils informatiques modernes nous permettent de simuler ces dynamiques avec une plus grande précision, offrant des perspectives sur l'interaction entre la variabilité stochastique et les paramètres biologiques.

1.2 Objectifs

Les principaux objectifs de cette étude sont :

- Mettre en œuvre un modèle déterministe simple comme référence.
- Développer et analyser un modèle stochastique plus réaliste en utilisant C++, en suivant le modèle de conception Model-View-Controller (MVC).
- Transférer les résultats de la simulation vers Python pour une visualisation avancée.
- Explorer les effets de contraintes telles que les taux de mortalité, la répartition par sexe et les biais de survie sur la dynamique de la population.
- Comprendre comment les conditions initiales et la stochasticité influencent les tendances de croissance à long terme.

1.3 Structure du rapport

Le rapport est organisé comme suit :

- **Approche de modélisation :** Un aperçu des modèles déterministes et stochastiques, y compris les détails de mise en œuvre.
- **Résultats de la simulation :** Analyse des résultats, en mettant l'accent sur l'identification des tendances, des anomalies et des biais tels que le 'biais de survie'.
- **Discussion :** Évaluation des choix de modélisation et de l'impact de l'introduction de contraintes.
- **Conclusion et travaux futurs :** Résumé des conclusions et suggestions pour étendre l'étude.

2 Aperçu du projet

Le projet est implémenté en C++ et utilise le modèle de conception MVC (Model-View-Controller), permettant une séparation claire des préoccupations tout en le rendant modulaire et évolutif. Chaque composant du modèle MVC est associé à des fichiers et des responsabilités spécifiques. Cette décision architecturale a été prise dès le début, lorsque nous avons mis en œuvre le mécanisme de flux de données du programme principal vers Python. L'ensemble du processus de construction est simplifié à l'aide d'un Makefile.

2.1 Système de construction

```
# Compilation de base
make

# Nettoyer les artefacts de construction
make clean

# Exécuter le programme avec sortie de données Python
make run        # exécute : ./build/main 2>data.py

# Lancer la surveillance Python
make watch
```

2.2 Architecture MVC

Le modèle de conception MVC divise l'application en trois couches interconnectées.

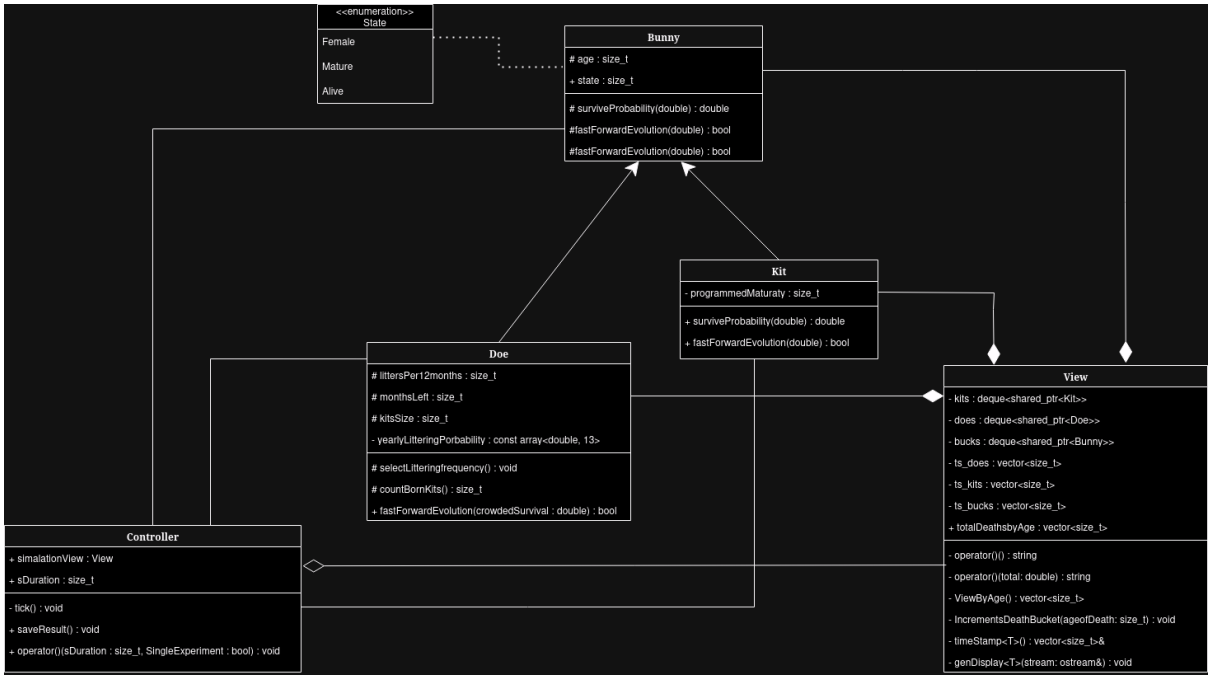

Figure 1. diagramme de classe

- **Modèle :** Représente les divisions de lapins (Bunny, Doe, Kit). Encapsule les données et comportements des lapins. Contient les méthodes d'évolution individuelle.
- **Vue :** Stocke la population actuelle en fonction de chaque division d'agents individuels. Contient des méthodes pour récupérer des informations spécifiques sur l'état actuel de la population.
- **Contrôleur :** Gère les entrées utilisateur et utilise les modèles pour mettre à jour la vue, par une étape d'un mois.

Aperçus

Dans cette section, je discuterai plus en détail des choix de conception faits pendant le développement et fournirai des explications supplémentaires.

- **Progression mensuelle :** Le plus petit incrément dans le contexte de la simulation actuelle, représenté par `Controller::tick()`, met à jour les entités dans un ordre logique qui évite les engagements doubles. Ces agents sont stockés dans une deque en tant que shared_ptr et sont traités de manière circulaire, où le popping du pointeur est une opération de complexité temporelle O(1) (contrairement à O(N) pour un vecteur/tableau). Si l'entité a survécu au mois en cours, elle est mise en file d'attente dans la division appropriée ; sinon, elle est automatiquement détruite, grâce à la nature du smart pointer. La seule perspective de la vue qui est mise à jour dans cette fonction est les divisions. La progression est cruciale pour maintenir la précision temporelle de la simulation.

- **Expérience en tant que foncteur :** Le choix de conception de démarrer l'expérience comme une logique de type fonction, malgré le fait qu'il s'agisse d'un objet du contrôleur, permet de reprendre l'expérience même après sa fin. Cela signifie que si l'expérience commence avec une population X et se termine avec Y, elle peut toujours continuer en partant de Y comme population initiale.

- **Gestion des signaux :** Il peut sembler inhabituel d'avoir un gestionnaire de signaux, mais cela a du sens lorsque le programme est arrêté une fois qu'il consomme trop de mémoire. Pour une seule expérience, environ pour mon ordinateur autour de 6 Go, le signal est déclenché par un Ctrl+C. L'appeler deux fois en succession rapide déclenche un mécanisme qui arrête instantanément le programme sans sauvegarder. Cependant, le déclencher une fois indique au contrôleur d'arrêter uniquement l'expérience en cours, puis de continuer automatiquement la fonction principale.

3 Méthodologie

L'implémentation est divisée en composants Modèle, Vue et Contrôleur, chacun remplissant un rôle distinct dans l'application.

3.1 Modèle (Famille de lapins)

La couche Modèle comprend des classes telles que Bunny, Doe et Kit. Ces classes stockent des attributs comme l'âge, la couleur et le comportement, et fournissent des méthodes pour l'interaction et la manipulation des données.

- **Génération pseudo-aléatoire avec état :** Pour garantir l'indépendance des événements tels que déterminer si un agent doit rester en vie pour le mois en cours ou déterminer la taille et la fréquence des portées pour l'année en cours, la méthode membre `inline double Bunny::getRandom() const`
- **Drapeau d'état :** le drapeau `size_t Bunny::state` est une variable publique qui sert de modèle de conception complémentaire basé sur l'état. Il est principalement utilisé lorsque le Kit atteint l'âge de maturité et sera soit incrémenté dans la division des Doe, soit dans celle des Buck. Par conséquent, il doit être cohérent avec toutes les instances, il utilise des bits de Bunny::State pour déterminer si l'état actuel existe. Par définition, il supporte un maximum de 64 états !!! Toute expansion supplémentaire peut nécessiter l'utilisation d'une concaténation de `char` ou tirer parti de la capacité du type `unsigned __int128`.
- **Comportement de portée des Doe :** La probabilité qu'une doe mette bas est pré-calculée et stockée dans le projet en utilisant un tableau constant :

```
const std::array<double, 13> Doe::yearlyLitteringProbability;
```

Ces données sont utilisées pour déterminer combien de fois la doe mettra bas au cours des 12 prochains mois. Le calcul est géré dans la méthode

```
void Doe::selectLitteringFrequency();
```

et est intégré à la fonction redéfinie

```
bool fastForwardEvolution(double crowdedSurvival = 1.0) override;
```

L'implémentation repose sur la fonction `Bunny::getRandom()`, supposée être un générateur fiable qui distribue uniformément les X portées sur 12 mois. Bien que cette approche soit efficace, elle manque de précision dans de rares cas où la doe pourrait produire plus ou moins de portées que X. Cependant, d'après des tests approfondis, de telles déviations sont rares.

Une méthode alternative que nous avons tentée impliquait de transformer le modèle en un système en temps réel. Cette approche visait à générer dynamiquement entre 3 et 9 portées au cours de l'année, avec une probabilité plus élevée de produire 5, 6 ou 7 portées. Ce système en temps réel déterminait la fréquence des portées mois par mois plutôt que de pré-calculer le nombre annuel de portées.

L'approche en temps réel était importante pour simuler des agents qui n'étaient pas conscients des événements futurs, tels que les risques de mortalité ou les opportunités de portée manquées en raison de facteurs externes. Malgré ses avantages conceptuels, la mise en œuvre de cette approche s'est avérée difficile et a nécessité un investissement significatif d'une semaine, nous conduisant finalement à privilégier la méthode pré-calculée actuelle.

Pour l'approche pré-calculée, nous avons adopté une distribution gaussienne avec des ajustements mineurs. Après des tests itératifs utilisant des outils comme Desmos.com, nous avons finalisé la distribution suivante :

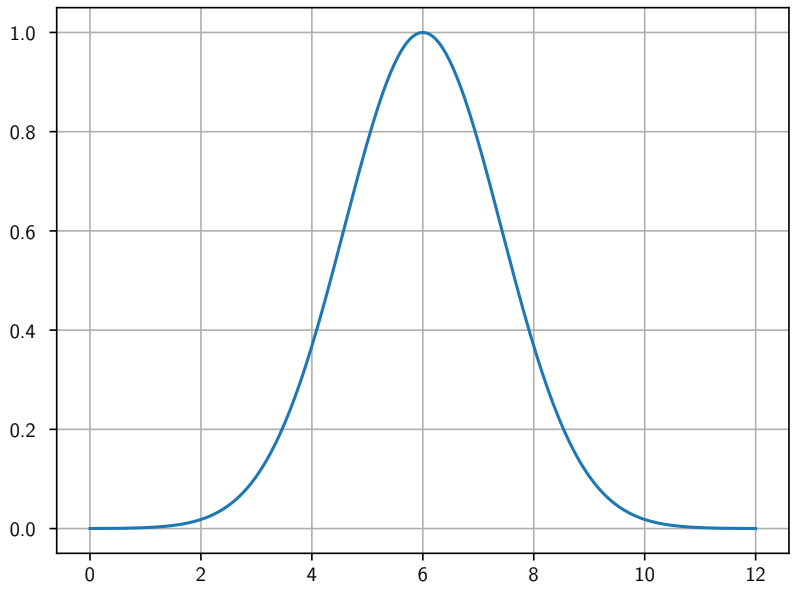

Figure 2. Graphique de $f(x) = e^{-\left(\frac{x-6}{2}\right)^2}$

De plus, la fonction de distribution cumulative (CDF) correspondante a été dérivée :

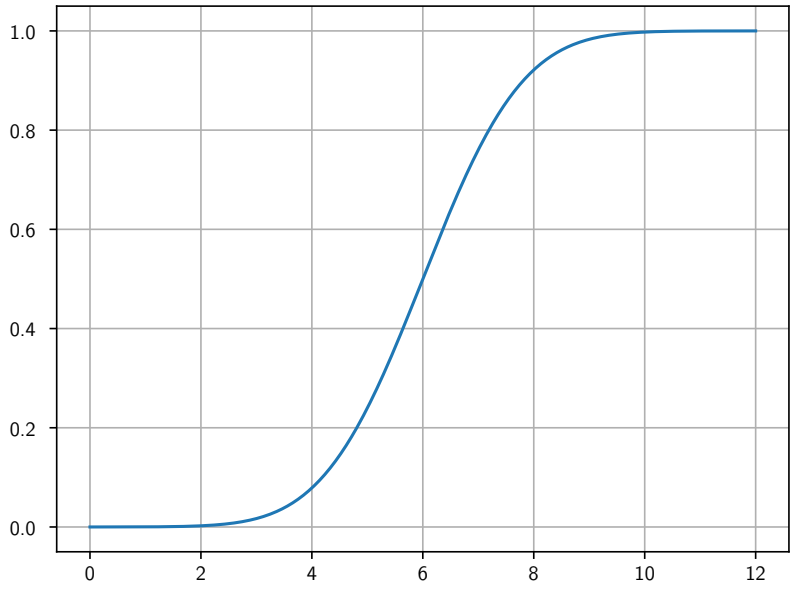

Figure 3. Graphique de $g(x) = \frac{1}{2\sqrt{\pi}} \int_{-\infty}^{x} e^{-\left(\frac{y-6}{2}\right)^2} dy$

Pour assurer une application pratique, car les valeurs fractionnaires de portée n'ont pas de sens, nous avons appliqué une transformation aux résultats calculés :

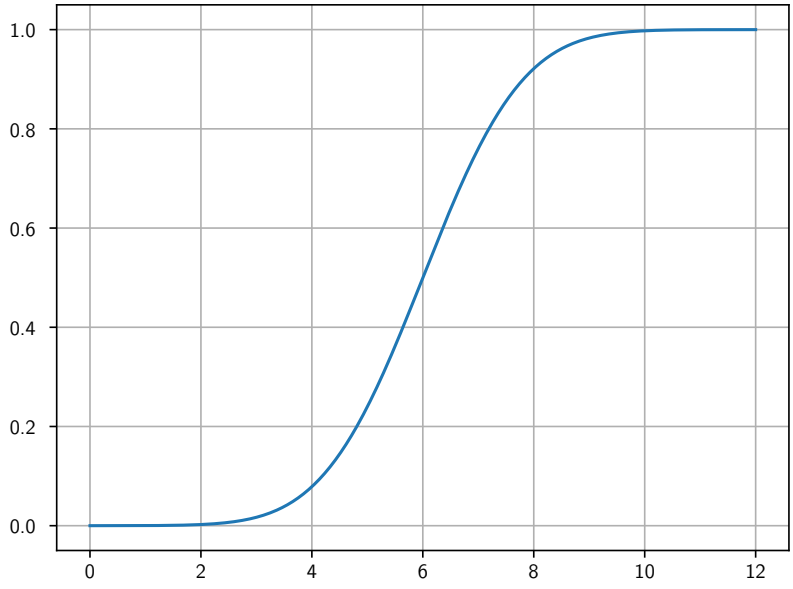

Figure 4. Graphique de $g(x) = \frac{1}{2\sqrt{\pi}} \int_{-\infty}^{\lfloor x \rfloor} e^{-\left(\frac{y-6}{2}\right)^2} dy$

- **Transformation de la mortalité :** Un aspect crucial du projet est la fonction de mortalité déterminant la probabilité de survie des agents. Bien que les métriques

données dans l'énoncé du problème soient des probabilités annuelles et que la simulation fonctionne par étapes mensuelles, une transformation mathématique a été mise en œuvre plutôt que de pré-calculer les âges de décès à l'initialisation. Cette approche est conforme au principe selon lequel les agents ne devraient pas avoir de connaissance préalable de leurs états futurs.

La fonction implémentée intègre :

- Pour les lapins matures :
  - Une probabilité de survie annuelle de base de 0,6, transformée en probabilité mensuelle en utilisant $P_{monthly} = P_{yearly}^{1/12}$
  - Un mécanisme de déclin basé sur l'âge commençant à 120 mois (10 ans)
  - Une diminution progressive au cours des 60 mois suivants (5 ans), calculée comme $((180 - age)^{1/12})$
  - Intégration avec un facteur de surpopulation affectant la survie
- Pour les kits (lapins immatures) :
  - Une probabilité de survie annuelle de base plus faible de 0,35
  - Le même principe de transformation mensuelle
  - Application du facteur de surpopulation

La fonction responsable distribue efficacement les probabilités annuelles en probabilités mensuelles, maintenant à la fois la précision temporelle et les principes de conception de la simulation.

```
inline double Bunny::surviveProbability(double crowdedSurvival) const{
    // return 1.0;
    if((state & (size_t) State::Mature)){
        static const double baseMatureYearlyProbability = 0.6;
        static double MatureSurviveProbabilityPerMonth =
        ↪  std::pow(baseMatureYearlyProbability,1.0/12);
        static const int ageDecline = 120;
        if(age < ageDecline)
                return MatureSurviveProbabilityPerMonth *
                ↪  crowdedSurvival;
        return ((age> ageDecline + 60)? 0 : std::pow((ageDecline + 60 -
        ↪  age) ,1.0/12))  * crowdedSurvival; //120 = 10ans, 60 = 5ans
        ↪  restant

    }
    //for kits
    static const double baseBabyYearlyProbability = 0.35;

    static double BabySurviveProbabilityPerMonth =
    ↪  std::pow(baseBabyYearlyProbability,1.0/12);

    return BabySurviveProbabilityPerMonth * crowdedSurvival ;

}
```

3.2 Vue (Différentes vues et affichage)

La couche **Vue** dans ce projet joue un rôle crucial dans le rendu et le formatage de l'état de la simulation des lapins. Son objectif principal est de fournir un affichage clair, interactif et lisible par l'homme des données traitées par le **Modèle**. Implémentée en C++, la classe `View` encapsule des responsabilités telles que la visualisation des distributions d'entités, la gestion des horodatages et le formatage des données pour une analyse ou une intégration externe (par exemple, le scripting Python).

Responsabilités principales

- **Gestion des entités :**
  - La classe `View` suit trois divisions principales d'entités : **Kits**, **Does** et **Bucks**.
  - Chaque groupe est géré via des conteneurs `std::deque` de `std::shared_ptr`, assurant une insertion, une suppression et une gestion de la mémoire efficaces.
  - Des méthodes dédiées telles que `division<T>()` fournissent un accès spécifique au type à ces groupes, tirant parti de la spécialisation des templates pour plus de flexibilité.
- **Visualisation des données :**
  - La méthode `ViewByAge()` agrège les comptes d'entités en fonction de leur âge, redimensionnant dynamiquement le `std::vector` interne pour accueillir l'entité la plus ancienne de la simulation.
  - La sortie de ces visualisations peut être formatée en chaînes structurées à l'aide de la méthode `fancyVectorDisplay()`, fournissant un résumé concis des distributions d'âge.
- **Agrégation axée sur la performance :**
  - La méthode `timeStamp<T>()` fournit un suivi efficace du temps pour chaque type d'entité, stocké dans `std::vector<size_t>`.
  - Les surcharges de `operator()` renvoient des résumés de la composition de la population, soit sous forme de comptes bruts, soit sous forme de pourcentages par rapport à la population totale. Cela garantit que les informations critiques peuvent être rapidement accessibles pour l'affichage ou l'analyse.
- **Intégration avec Python :**
  - La méthode `pythonFormatting()` fait le lien entre les données de simulation et les workflows Python, permettant une intégration avec des outils comme NumPy. Cette fonctionnalité est particulièrement utile pour la visualisation et l'analyse avancées des données en dehors de l'application C++.
- **Suivi et analyse :**

– La méthode `IncrementsDeathBucket(size_t ageofDeath)` enregistre le nombre de décès par âge, mettant à jour dynamiquement le vecteur `totalDeathsbyAge` pour refléter les résultats de la simulation.

Considérations de conception La classe `View` démontre les principes clés de l'architecture **MVC** en isolant la représentation des données de la logique métier. En utilisant la spécialisation des templates et des structures de données flexibles, la conception assure évolutivité et adaptabilité, permettant des extensions futures sans modifications significatives.

De plus, l'utilisation de conteneurs STL efficaces (`std::deque` et `std::vector`) et la propriété partagée via `std::shared_ptr` reflètent une focalisation sur la performance et la sécurité dans la gestion des entités dynamiques.

Limitations et améliorations Bien que la couche `View` soit robuste, elle est limitée par les limitations des unités de traduction de C++, nécessitant l'instanciation en fichier des spécialisations de templates. Cela pourrait potentiellement être amélioré avec des instanciations explicites dans un fichier d'implémentation séparé pour réduire l'encombrement. De plus, étendre les capacités de formatage à d'autres langages de script ou introduire un rendu graphique pourrait améliorer l'interactivité et l'utilisabilité de la couche `View`.

3.3 Contrôleur (Gestion des interactions)

Le `Contrôleur` sert d'intermédiaire entre les entrées utilisateur, le `Modèle` et la `Vue`, orchestrant le flux d'informations et assurant la synchronisation au sein de la simulation des lapins. Il joue un rôle crucial dans la gestion des mises à jour de la simulation, le traitement des données et la garantie que la `Vue` reflète fidèlement l'état du `Modèle`.

Responsabilités

Le `Contrôleur` est responsable de :

- Initialiser la simulation en créant la population initiale de lapins (mâles, femelles et petits) et en configurant la `Vue`.
- Gérer les étapes temporelles de la simulation via la fonction `tick`, qui met à jour toutes les entités et gère les transitions de cycle de vie (par exemple, naissance, maturité, décès).
- Collecter et stocker des instantanés de données, tels que les distributions de population par type et par âge, pour une analyse ou une visualisation ultérieure.

- Fournir une interface pour exécuter des simulations de durées variées et enregistrer les résultats dans un format compatible avec Python.

Détails de l'implémentation

Le `Contrôleur` est implémenté en tant que classe C++ avec les éléments clés suivants :

Initialisation Le constructeur du `Contrôleur` initialise la simulation en :

- Allouant l'objet `Vue` pour gérer les entités de la simulation et leurs interactions.
- Peuplant la `Vue` avec les comptes spécifiés de mâles (`malesCount`), de femelles (`femalesCount`) et de petits (`babiesCount`) :
  - Les mâles et les femelles sont initialisés avec un état de maturité, représenté par le drapeau commun `Bunny::State::Mature`.
  - Les petits sont ajoutés directement à la `Vue` sans état de maturité initial.

Mises à jour des étapes temporelles (`tick()`) La fonction `tick` met à jour l'état de la simulation pour chaque étape temporelle :

1. **Femelles :** La progression de l'âge, la reproduction et la survie sont gérées. Les naissances sont ajoutées à la population des petits.
2. **Mâles :** La survie et le vieillissement sont suivis.
3. **Petits :** Les petits mûrissent en mâles ou en femelles ou peuvent mourir avant la maturité.
4. **Nouveaux-nés :** Les nouveaux petits sont ajoutés à la population en fonction des taux de reproduction.

Exécution de la simulation (`operator()`) La fonction `operator()` gère le processus global de simulation :

- Exécute la simulation pour une durée spécifiée (`sDuration`).
- Optionnellement, produit des statistiques de population à chaque étape temporelle lorsque `SingleExperiment` est vrai.
- Met à jour les données horodatées pour les mâles, les femelles et les petits afin de suivre les changements de population au fil du temps.

Exportation des données (`saveResult()`) La fonction `saveResult` prépare les données de simulation pour une analyse externe :

- Génère des tableaux formatés pour Python des données horodatées de la population.
- Produit des données sous forme de tableaux `numpy` pour une intégration transparente dans des pipelines de visualisation ou d'analyse de données.

Méthodes clés et interactions

- `tick()`: Met à jour le cycle de vie de toutes les entités dans la simulation.
- `operator()`:
  - Gère l'exécution globale de la simulation.
  - Synchronise l'état du `Modèle` et met à jour la `Vue`.
- `saveResult()`: Formate les données dans une structure compatible avec Python pour une utilisation externe.

Conclusion

Le `Contrôleur` est l'épine dorsale de la simulation, garantissant que les interactions entre le `Modèle` et la `Vue` sont cohérentes et logiques. Son design modulaire et ses fonctions explicites en font un composant robuste de l'architecture MVC.

4 Résultats & Discussion

4.1 Modèle agrégé

La modélisation de la dynamique de population des lapins basée sur la suite de Fibonacci nous a permis d'explorer la croissance d'une population idéalisée. La problématique centrale était de comprendre comment une population se développe en considérant uniquement les cycles de reproduction et la maturation des individus, sans tenir compte des contraintes environnementales.

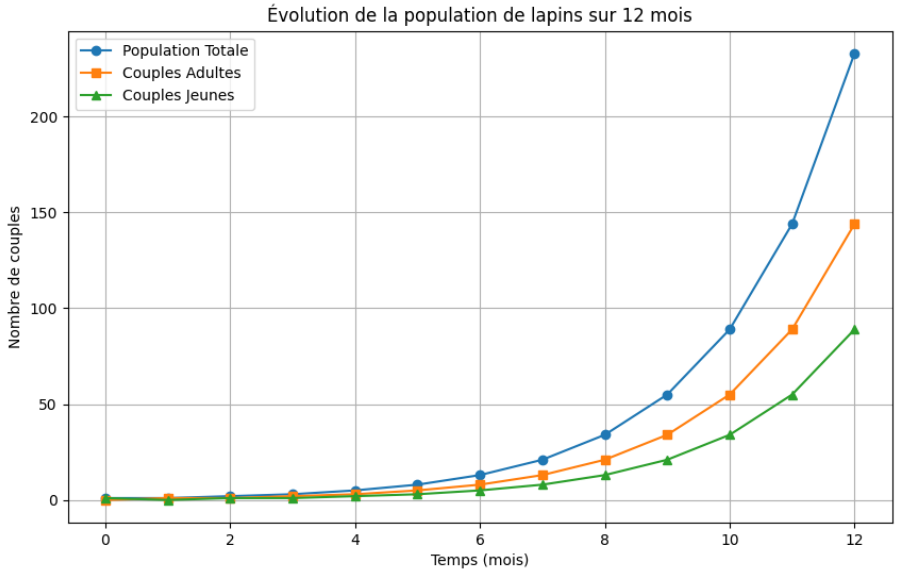

Figure 5. Modèle simple du premier Lab

Les résultats de simulation montrent une croissance exponentielle caractéristique, avec une distinction claire entre couples jeunes et adultes. Cette croissance suit fidèlement la relation récursive F(n) = F(n-1) + F(n-2), illustrant la progression théorique de Fibonacci. L'analyse des courbes révèle une augmentation continue de la population totale, accompagnée d'oscillations dans la proportion de couples jeunes, reflétant les cycles de reproduction mensuels. Cependant, ce modèle agrégé présente des limitations significatives : l'absence de mortalité, une croissance illimitée et des cycles de reproduction fixes. Ces simplifications, bien qu'utiles pour une première approche, ne reflètent pas la complexité des systèmes biologiques réels. Ces limitations ouvrent naturellement la voie à une approche plus sophistiquée basée sur les agents individuels. Un tel modèle permettrait d'intégrer des comportements individuels, des interactions locales et des variations stochastiques dans les processus de reproduction et de mortalité. Cette transition vers une modélisation individu-centrée constitue une évolution naturelle pour capturer la complexité inhérente aux dynamiques de population réelles.

4.2 Modèle d'Agents

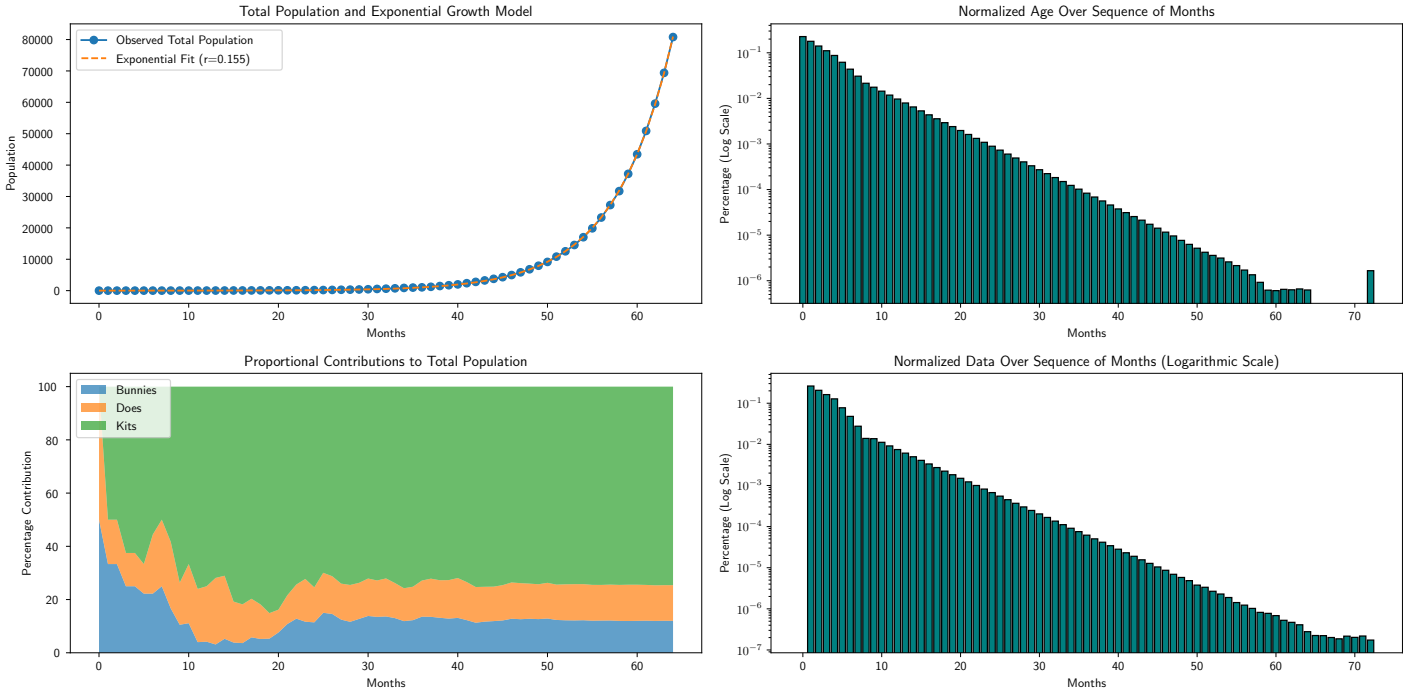

Figure 6. 10,000 expériences de 70 mois sans contraintes supplémentaires

Analyse de la Dynamique de Population de la Simulation des Lapins

L'analyse de la simulation de la population de lapins sur 70 mois révèle quatre aspects clés de la dynamique du système :

Modèle de Croissance Exponentielle La population totale montre une croissance exponentielle avec un taux $r = 0.155$, décrit par :

$$P(t) = P_0 e^{rt}$$

où P_0 est la population initiale. Le système présente deux phases distinctes : une période de croissance lente ($t < 40$ mois) suivie d'une expansion exponentielle rapide, atteignant finalement environ 8×10^4 individus à $t = 60$ mois.

Biais de Survie dans la Distribution d'Âge La distribution d'âge normalisée montre une anomalie critique qui reflète le biais de survie. Ceci est exprimé mathématiquement par :

$$S(t) = \begin{cases} P(t) & \text{pour } t \leq t_{\text{sim}} - t_0 \\ P_{\text{initial}}(t) & \text{pour } t_{\text{sim}} - t_0 < t \leq t_{\text{sim}} \end{cases}$$

où $t_{\text{sim}} = 70$ représente la durée totale de la simulation. L'écart de 7 mois entre les générations initiales et suivantes émerge parce que :

- Seule la population initiale peut atteindre la durée totale de la simulation
- Les deuxième et suivantes générations sont limitées par $(t_{\text{sim}} - t_{\text{birth}})$
- Cela crée un plafond de probabilité de survie pour toutes les cohortes non initiales

Composition de la Population et Dynamiques d'Attracteur Les dynamiques de distribution proportionnelle révèlent un comportement complexe d'attracteur :

$$\lim_{t\to\infty} \phi_i(t) = \phi_i^*$$

Au taux actuel $r = 0.155$, le système présente un attracteur ponctuel unique avec :

$$\begin{aligned} \phi^*_{\text{Kits}} &\approx 0.70 \\ \phi^*_{\text{Does}} &\approx 0.15 \\ \phi^*_{\text{Bunnies}} &\approx 0.15 \end{aligned}$$

Un comportement critique de bifurcation est prédit pour des valeurs de r plus élevées :

$$\exists r_c : r > r_c \implies \text{séquence de bifurcation}$$

Cela suggère :

- Émergence potentielle de proportions en cycle de 2
- Autres bifurcations menant à des motifs en cycle de 4
- Transition possible vers un régime chaotique à des valeurs de r plus élevées
- Multiples attracteurs coexistants dans l'espace des phases

Distribution de la Mortalité Le taux de mortalité spécifique à l'âge $\mu(t)$ fournit des informations complémentaires au biais de survie :

$$\mu(t) = \mu_0 e^{-\lambda t}$$

Cette distribution est particulièrement précieuse car :

- Elle représente les événements de décès réels plutôt que les probabilités de survie
- Reste non affectée par le biais de la durée de la simulation
- Confirme l'hypothèse de taux de risque constant dans le modèle
- Fournit une métrique de validation pour les hypothèses du modèle

Le modèle de décroissance exponentielle indique une mortalité la plus élevée dans les premiers mois ($t < 10$), suivie d'un déclin constant, établissant les dynamiques démographiques de base indépendamment des artefacts de simulation.

4.3 Injection de Contraintes Biaisées

Dans le code, j'ai intentionnellement laissé une marge pour injecter une probabilité nommée
`double crowdedSurvival = 1.0`, la probabilité est calculée globalement et affecte les survivants, nous allons donc l'exploiter davantage maintenant, elle affecte les agents.

```
size_t grandTotal = simulationView.division<Bunny>().size() +
    simulationView.division<Doe>().size() +
    ↪  simulationView.division<Kit>().size();

double crowdedSurvival = (grandTotal && grandTotal <= 2* 1e4 )?
↪  (grandTotal/((double) 2* 1e4)) : 0;

crowdedSurvival = std::pow(1.0 - std::pow(crowdedSurvival,0.9),1/0.9);
```

Appliquer ce plafond suggère ce qui suit :

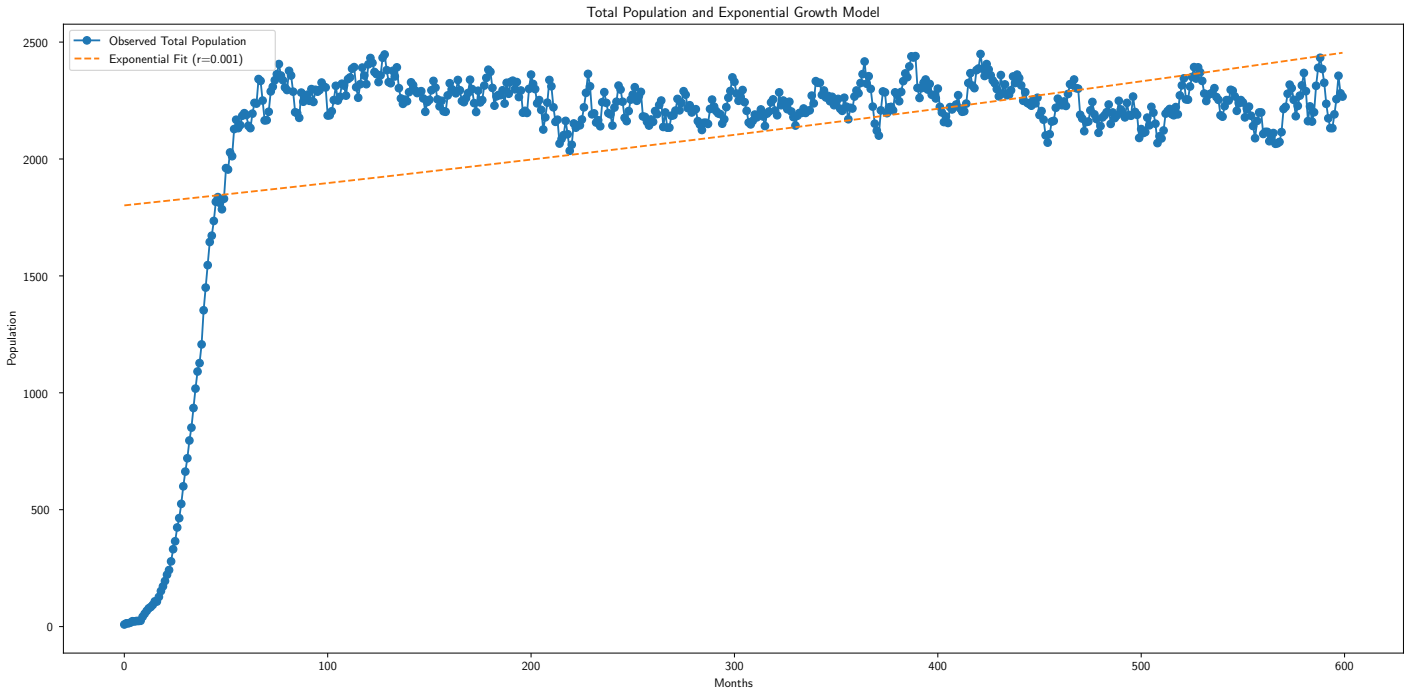

Figure 7. 1 expérience de 600 mois avec plafond de population

5 Conclusion

Le projet met en évidence l'efficacité du modèle de conception MVC pour créer des applications modulaires et évolutives. La séparation des préoccupations permet un débogage,

des tests et des améliorations futures plus faciles. Les améliorations potentielles incluent l'ajout de comportements supplémentaires ou l'optimisation des performances de rendu.