sim-rhinite

Projet Informatique L2

par Léo Peyronnet

Juin 2024

Table des matières

1 Projet de simulation

Ce projet s'inscrit dans l'évaluation d'un enseignement dispensé en L2 Informatique par l'UCA nommé "Projet Informatique". Cet enseignement a pour sujet la **réalisation de simulations scientifiques à l'aide d'outils informatiques**. Ce projet répond au sujet de session de rattrapage de l'année 2024.

1.1 Présentation de la simulation et de ses objectifs

sim-rhinite est un projet de simulation de maladie infectieuse type rhinite. L'objectif principal de cette simulation est de modéliser la propagation d'un virus non mortel dans une population et de produire des statistiques à partir de ce modèle qui pourront être analysées ultérieurement. Les objectifs spécifiques de cette simulation sont les suivants :

1. **Modélisation la population et l'espace de simulation**
   - Création une grille 2D torique pour représenter l'espace dans lequel les individus se déplacent.
   - Initialisation d'une population avec des positions aléatoires sur cette grille.
2. **Dynamique de déplacement**
   - Simulation du déplacement aléatoire des individus sur la grille, de 6h à 22h chaque jour, avec un pas de temps d'une heure.
3. **Propagation et évolution du virus**
   - Implémentation des règles de propagation du virus avec une probabilité de transmission dépendant de l'état de la maladie chez l'individu infecté et de son voisinage de Moore d'ordre 2.
   - Modélisation de la période d'incubation et les différentes phases de contagiosité du virus sur une période de 12 jours.
4. **Variables**
   - Initialisation de la simulation avec différents nombres de personnes contaminées (de 1 à 10) afin d'observer les effets sur la propagation.
5. **Statistiques**
   - Les simulations sont répétées 30 fois pour chaque valeur de notre variable d'expérience (nombre initial de contaminés).
   - Les résultats sont collectés et présentés sous forme de tableaux avec des intervalles de confiance à 95%.

1.2 Choix conceptuels

1.2.1 Constante temporelle

Dans le sujet, il est demandé de tester la simulation sur 1 mois, 3 mois, 6 mois et 1 an. Il est alors possible de faire de la durée de simulation une seconde variable de notre expérience afin de produire des statistiques pour chaque combinaison de valeurs de variables (durée et nombre initial de contaminés). Cependant, les informations des simulations sur 1, 3 et 6 mois sont également contenues dans la simulation sur 1 an.

Soit $1, 2, 6$ et 12 des ensembles représentant les informations récoltées lors d'une simulation de durée égale à leur symbole (en mois), alors :

$$1 \subset 3 \subset 6 \subset 12$$

Il n'est alors nécessaire de simuler qu'avec la durée maximale, c'est à dire 1 an (12 mois). Nous pouvons donc considérer la durée comme une constante.

1.2.2 Positionnement de la population

Dans le sujet, il est indiqué que l'on positionne sur la grille les personnes avec des coordonnées aléatoires. Cependant, rien n'est explicité dans le cas où deux personnes obtiennent les mêmes coordonnées. J'ai donc jugé bon que les coordonnées soit unique à une personne.

1.2.3 État de la personne au terme de la contamination

Dans le sujet, rien n'est explicité quant à l'état d'une personne contaminée lorsqu'elle arrive au terme de sa période de contamination (j+12). Est-elle considéré comme à nouveau saine (et donc contaminable) ou considérons nous qu'elle est "guérie", c'est à dire immunisée à la maladie et donc non contaminable ?

Suite à la consultation du responsable de l'enseignement, un compromis a été décidé. Au terme de sa période de contamination, la personne est considérée comme guérie pendant une durée établie, puis au terme de cette dernière, la personne est reconsidéré comme saine (et donc contaminable).

Nous reviendrons ultérieurement sur le choix de valeur de cette durée d'immunité.

2 Programme informatique

La simulation a été implémentée par un programme écrit en langage C. Le répertoire du projet est disponible sur un dépôt distant, fourni en annexe.

2.1 Choix d'implémentation

2.1.1 Générateur d'aléatoire

Dans le sujet, il est demandé d'utiliser "le meilleur générateur de nombre pseudo-aléatoire vu en cours". Conformément à ce que l'on a vu en cours et en TP, j'ai choisi le **Mersenne Twister** dans son implémentation MT 19937. Ainsi, dans la suite de cette rédaction, tous les appels à l'aléatoire sont à comprendre comme des appels à ce générateur.

2.1.2 Format des résultats finaux

Le programme doit produire des données statistiques sur la simulation. J'ai choisi le format **CSV** pour stocker ces résultats, notamment pour ma familiarité avec ce format, sa simplicité et la possibilité d'être lu par des outils type tableurs que j'ai utilisé pour la création de visuels (diagrammes, courbes, etc...).

Les fichiers ainsi crées par le programme seront placés dans un répertoire spécial situé à la racine et nommé **out/**.

2.2 Structure du projet

Le projet est composé de deux programmes exécutables :

— **stats_sim** : Il permet de produire des statistiques sur l'évolution de la contamination de la maladie en fonction du nombre d'infectés initiaux.

— **idle** : Annexe permettant de visualiser le comportement de la population simulée.

Le répertoire racine du projet comprend plusieurs sous-répertoires :

— **redaction/** : comprend l'ensemble des fichiers relatif à ce compte-rendu.

— **source/** : comprend l'ensemble des fichiers sources du programme.

— **packages/** : comprend l'ensemble des dépendances logicielles du programme.

La racine contient également un builder et un fichier **README.md** contenant des informations sommaires sur le projet et des consignes d'usage.

2.2.1 Packages employés

Le sous-répertoire **packages/** contient :

— **affichage/** : fonctions d'affichage de tableau et de matrices.

— **csv/** : fonction d'écriture de matrices au format CSV.

— **menuing/** : ensemble de fonctions permettant la création, la gestion et l'affichage de menus dynamiques dans le terminal.

— **mt19937ar/** : fonctions relatives à la génération de nombres pseudo-aléatoires selon le Mersenne Twister.

2.2.2 Code source et builder

Les fonctions (fonctions métier, initialisation et fonction launch_sim) ainsi que les structures de données communes aux deux programmes sont définies dans le fichier source **rhinite.c** et son fichier d'en-tête **rhinite.h**. Ensuite, chaque programme possède son propre fichier source qui fait appel à **rhinite.h**.

Comme expliqué précédemment, le projet est équipé d'un builder (**makefile**) qui facilite le processus de compilation des fichiers sources du projet mais aussi des différents packages. Les informations relatives à son utilisation sont disponibles dans le fichier **README.md** mentionné précédemment.

2.3 Structuration des données

Le fichier **rhinite.h** défini deux types personnalisés :

— **etat_t** : énumération des états dans lesquels une personne pourra être. Défini un état sain, incubant, contaminant et guéri.

```
typedef enum {SAIN, INCUBANT, CONTAGIEUX, GUERI} etat_t;
```

— **personne_t** : structure représentant un individu (ou personne) d'une population. Composée d'un couple d'entiers représentant la position de la personne sur la grille, d'un état comme défini au-dessus et d'un autre entier comptant le nombre de jours depuis que la personne est infectée.

```
typedef struct {
    int x, y;
    etat_t etat;
    int jour_infection;
} personne_t;
```

Une population pourra ainsi être représentée par un tableau de personne.

2.4 Fonctions d'initialisation

Le fichier **rhinite.c** permet l'initialisation d'une population grâce à une fonction nommée **init_population**. Cette fonction permet donc l'initialisation d'une population, induisant qu'elle fourni à ses individus des coordonnées aléatoires uniques. La fonction retourne un tableau de n personnes.

Une autre fonction nommée **init_contamination** permet de définir dans une population les individus qui seront les infectés initiaux. Pour rappel, leur nombre correspond à notre variable d'expérience (cf. 1.1). La sélection de ces individus parmi la population se fait de manière aléatoire.

2.5 Fonctions métier

Le fichier **rhinite.c** implémente également plusieurs autres fonctions permettant de manipuler une population.

2.5.1 deplacement_alea

deplacement_alea permet de déplacer chaque individu vers une case adjacente à la sienne. Cette case est choisie de manière aléatoire parmi celles qui ne sont pas occupées par un autre individu de la population. Les coordonnées des individus leurs sont donc ainsi toujours uniques. Aussi, si toutes les cases adjacentes sont occupées, le déplacement est annulé.

La fonction prends également en compte le fait que la grille sur laquelle les individus sont placés est torique.

2.5.2 evolution_journaliere_maladie

evolution_journaliere_maladie permet de faire évoluer l'état de la maladie pour chaque individu infecté. Elle incrémente donc le compteur **jour_infection** (cf. 2.3) pour tous les individus infectés. Elle gère ensuite leur changement d'état en fonction des durées de chacun des états de la maladie.

2.5.3 contamination

contamination gère les contaminations entre les individus. Pour chaque individu contagieux, elle réalise un lancer aléatoire pour chaque autre individu présent dans le voisinage de Moore d'ordre 2 de l'individu contagieux.

La probabilité de contamination est définie de manière discrète en fonction de la valeur de **jour_infection**. Ainsi, si le lancer possède une valeur inférieure ou égale à la probabilité de contamination, l'individu voisin est infecté.

Cette fonction comptabilise également le nombre de contaminations réalisées durant son appel (qui correspond à une heure dans le contexte de notre simulation). Cette comptabilisation sert de base pour le travail d'analyse statistique que nous verrons ultérieurement. **contamination** renvoie donc un entier représentant cette comptabilisation.

2.6 Fonction launch_sim

La fonction **launch_sim** permet de réaliser une simulation complète suivant des paramètres donnés et fait appel à l'ensemble des fonctions présentées précédemment. Elle permet notamment de leur apporter un contexte temporel.

Ainsi, elle commence par faire appel aux fonctions d'initialisation ; d'abord **init_population** qui lui renvoie un tableau de personnes, puis **init_contamination** qui prends, entre autres paramètres, ce tableau.

Ensuite, pour chaque jour prévu dans la simulation, elle lance un cycle de 24 heures à l'intérieur duquel, pour chaque heure, est fait appel à la fonction **contamination**, et si l'heure est comprise entre 6 heures et 22 heures, un appel à la fonction **deplacement_alea**.

Enfin, au terme de chaque jour, la somme des contaminations qui ont eu lieu dans la journée est stockée dans un tableau d'entiers initialisé au préalable et un appel à **evolution_journaliere_maladie** est effectué pour mettre à jour les états journaliers.

launch_sim se clôt en revoyant le tableau de contamination journalière, ayant donc une taille égale au nombre de jours compris par la simulation.

2.7 Programmes exécutables

2.7.1 stats_sim

Le programme **stats_sim** fait appel à **launch_sim** afin de produire des statistiques sur l'évolution de la contamination de la maladie en fonction du nombre d'infectés initiaux.

Après avoir initialisé le générateur de pseudo-aléatoire, le programme initialise et défini l'ensemble des valeurs et constantes du programme :

```
// Déclaration constantes programme
double        proba_contamination[12] = {0, 0, 0, 0.6, 0.8, 0.7, 0.6, 0.5,
    0.4, 0.3, 0.2, 0.1};
int           jours = 365; // un an de simulation
int           nb_replications = 30;

// Déclarations variables programme (constantes expérience)
int           taille_grille;
int           num_personnes;
int           duree_incubation; // en jours
int           duree_contagion; // en jours
int           duree_imunitee; // en jours

// Déclaration variable programme (variable expérience)
int           num_infect_init;

// Initialisation constantes expérience
taille_grille     = 50;
num_personnes     = 100;
duree_incubation  = 2;
duree_contagion   = 9;
duree_imunitee    = 15;
```

Les valeurs de constantes présentées ici représentent les valeurs par défaut du programme et correspondent à celles indiquées dans le sujet. Nous appellerons désormais cette collection de valeurs "**default**" et elle servira de base dans notre travail d'analyse (cf. 3)

Ensuite, le programme boucle pour chaque valeur de **num_infect_init** de 1 à 10 (inclus). Pour chaque itération de la boucle, une seconde boucle est définie. Son nombre d'itération est égal à la valeur de **nb_replications** (dans notre cas, 30).

Ainsi, pour chaque itération de la seconde boucle, un appel à **launch_sim** est effectué et le tableau retourné par cette dernière est stocké dans un tableau de pointeurs initialisé en amont. Au terme de la seconde boucle, le tableau de pointeurs contient donc 30 tableaux d'entiers qui

contiennent chacun j entiers, avec j la durée de la simulation exprimée en jours (cf. constante **jours**).

Le programme exploite ce tableau de pointeur pour réaliser des calculs statistiques. Pour chaque jour, il calcule le nombre de contamination moyen des 30 réplications, leur écart-type et à partir de cela, un intervalle de confiance à 95%. L'intervalle est stocké dans un second tableau ayant une taille égale à j.

Enfin, le programme inscrit ce tableau d'intervalles dans un ficher formaté CSV avec un nom indiquant la valeur de **num_infect_init** avec laquelle la simulation a tournée. Ceci conclu une itération de la première boucle. Une fois cette dernière finie, le programme se termine.

2.7.2 idle

Le programme **idle** est une annexe permettant de visualiser le comportement de la population simulée. Il utilise une fonction nommée **launch_idle** qui correspond à une "redéfinition" de la fonction **launch_sim** auquel des affichages dans le terminal ont étés ajoutés.

idle démarre avec un menu donnant à l'utilisateur le choix entre plusieurs modes de lancement du programme :

- Vérification collisions : lance une simulation de un seul jour dans une grille minime (2x2) afin de visualiser le fonctionnement de la fonction métier **deplacement_alea** et sa gestion des "collisions" entre les individus.
- Valeurs par défaut : lance une simulation réduite par rapport à celle effectuée dans **launch_sim**. La durée de la simulation est de 15 jours, dans une grille 30x30 avec 50 personnes, dont 5 sont des infectés initiaux. Cette simulation permet d'avoir une vue plus globale du comportement d'une population tel que décrit par les fonctions métier.

Pour gérer l'affichage de la grille, **idle** défini trois fonctions :

comparer_personnes

Cette fonction de comparaison est utilisée par la fonction **qsort** pour trier un tableau de structures personne_t. Elle prend en entrée deux pointeurs génériques a et b, qu'elle convertit en pointeurs de type personne_t.

Elle compare d'abord les coordonnées x des deux personnes. Si elles sont différentes, la fonction retourne la différence entre ces deux coordonnées, ce qui permet de les trier en ordre croissant de x. Si les coordonnées x sont égales, la fonction compare alors les coordonnées y de manière similaire. Ainsi, les personnes sont triées d'abord par leur position x, puis par leur position y en

cas d'égalité sur x.

tri_croiss_population

Cette fonction prend un tableau de structures personne_t et le nombre d'éléments dans ce tableau. Elle alloue de la mémoire pour une copie de ce tableau, vérifie si l'allocation est réussie, et copie chaque élément du tableau original dans la copie.

Ensuite, elle utilise la fonction **qsort** pour trier la copie du tableau en utilisant la fonction de comparaison **comparer_personnes**. La fonction retourne finalement le pointeur vers la copie triée du tableau.

affich_population

Cette fonction affiche la population sur une grille de taille **taille_grille**. Elle commence par trier la population en utilisant la fonction **tri_croiss_population** et stocke le résultat dans **population_ordone**.

Ensuite, elle affiche une bordure supérieure en utilisant le caractère '#'. Pour chaque ligne de la grille, elle affiche également une bordure latérale gauche, puis vérifie pour chaque colonne si une personne de la population triée occupe cette position (c'est à dire que les coordonnées x et y correspondent à celles de la grille). Si une personne est présente, son état est affiché ; sinon, un espace vide est affiché. Après chaque ligne de la grille, une bordure latérale droite est affichée.

Finalement, une bordure inférieure est affichée. La fonction libère ensuite la mémoire allouée pour la copie triée de la population.

3 Analyse des résultats

4 Annexes

4.1 Graphiques

4.2 Script de création des graphiques

Afin de m'aider à réaliser les graphiques utilisés dans cette rédaction, j'ai écrit un petit script en Python nommé **graphiques.py**, que vous pouvez retrouver dans le répertoire **redaction/**.

Il est conçu pour générer un graphique représentant des intervalles de confiance à partir de données stockées dans un fichier CSV. Il fonctionne en important les packages **pandas** et **matplotlib.pyplot** qui servent respectivement à la manipulation de données et à la création de graphiques.

Le script prévoit l'usage d'arguments par ligne de commande. Seul le premier argument passé après le nom du script compte et est interprété comme le nom du répertoire dans lequel les fichiers des graphiques seront crées.

Des constantes sont définies pour indiquer où le script doit lire les fichiers CSV et où il doit créer les fichiers contenant les graphiques. Il lit donc les fichiers CSV crées par **stats_sim** qui se situent dans le répertoire **out/** et dessine ses graphiques dans le répertoire **redaction/graphiques/**. L'argument passé par ligne de commande définit alors le sous-répertoire dans lequel les fichiers seront crées, tel que **redaction/graphiques/sous-repertoire/**.

Ensuite, le script définit et fait appel à une fonction nommée **draw**. Comme son nom l'indique, c'est elle qui est chargée de créer un fichier et de dessiner dedans un graphique correspondant à un fichier CSV. Elle prend comme paramètres un DataFrame (df) contenant les bornes inférieures et supérieures des intervalles de confiance, un titre pour le graphique, le chemin (path_file) où enregistrer le graphique, et le nom du fichier (file_name). Elle crée un graphique avec deux lignes (pour les bornes inférieure et supérieure) et une zone remplie entre elles, représentant les intervalles de confiance.

Le script fait appel à **draw** pour chaque fichier source CSV afin de créer un graphique par fichier source. Chaque appel est précédé par la création d'un DataFrame où les données de la première colonne du fichier csv sont contenues dans un champ correspondant à la borne inférieure et la seconde colonne dans un champ correspondant à la borde supérieure. L'appel se fait donc avec ce DataFrame en argument.

4.3 Crédits et liens externes

Projet et rédaction réalisé par Léo Peyronnet

— Portfolio - https://portfolio.leopeyronnet.fr/

— Email - peyronnet.leo@gmail.com

Le dépôt distant du projet est trouvable sur Github à cette adresse :
https://github.com/LaiPe/sim-rhinite