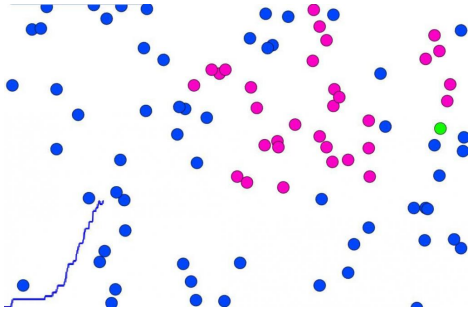
Projet C++ – *Contamination*

ENSAI — 2e année – 2020



T. Guyet

Présentation du projet

Objectifs

Ce projet est destiné à vous faire manipuler avec une plus grande autonomie l'ensemble des concepts de la programmation orientée objet en C++ présentés en cours et en TP :

- création de classes, encapsulation, protection de données,
- réutilisation de composants par héritage (dérivation) et composition,
- STL,
- allocation dynamique.

Même si vous n'avez pas eu le temps de finaliser les TP en séance, vous pourrez vous aider des corrections mises à disposition sur Moodle.

Organisation du projet

Votre retour doit se composer d'une part des sources de votre programme, ainsi que d'un document court (2 pages maximums) me permettant de juger au mieux votre travail. En particulier, vous pouvez mettre en avant les propositions particulières sur lesquelles vous souhaitez attirer mon attention (parce que vous pensez que c'est bien) ou apporter des explications à des solutions que vous ne pensez pas optimales (mais pour lesquelles vos tentatives n'ont pas abouti).

Quelques informations complémentaires mais importantes :

- vous pouvez poser vos questions par mail tout au long du projet (réponse en délai "raisonnable" ;
- le TP est effectué en groupe d'au plus 2 personnes ;
- le TP sera à rendre sur la plate-forme Moodle : les fichiers (.cpp, .h + 1 rapport en PDF). L'ensemble de ces fichiers sera déposé ensemble au travers de Moodle (**non-zippé**).
- Le code que vous me fournissez doit être fonctionnel ! C'est à dire que :
  - Le code doit compiler sans erreur (en norme C++98 ou C++11)
  - Le programme doit s'exécuter sans erreur
- la date limite de retour est fixée au **8/04/2020 – 23h55** [1].

Ce sujet de projet est moins dirigé que les TP. Les étapes sont peu décrites pour préférer la spécification du résultat. Il est néanmoins important de conserver de bonnes pratiques de programmation pour faciliter la conduite de votre projet :

- Pensez à **compiler régulièrement** votre programme pour régler les problèmes dès qu'ils apparaissent.
- Faites des **tests** de fonctionnalité de vos classes, vos fonctions (et faites vos tests **au fur et à mesure** de votre avancée !). NB : les tests que vous effectuerez pourront être organisés sous la forme de fonctions placées dans le fichier `main.cpp`. Certains tests sont explicitement à réaliser mais n'hésitez pas à en ajouter vous-mêmes.
- N'hésitez pas à faire beaucoup d'affichages dans un premier temps, cela vous aidera pour le débogage.
- Ajouter des commentaires facilitant la correction de votre code, ainsi que le travail collaboratif.
- Pensez à nettoyer votre code avant de le rendre (supprimer les commentaires et tests inutiles)

Une partie de l'évaluation sera basée sur l'application de tests automatiques. Il est donc important de **respecter le nom des classes et des fonctions** qui vous sont proposées.

1. La plate-forme Moodle sera configurée pour interdire les dépôts ultérieurs à la date limite.

1 Contexte du projet

1.1 Contexte général

Le contexte générale est celui de l'étude de l'impact du confinement sur la propagation d'un virus contagieux dans une population. Ce TP est fortement inspiré par un article du Washingtown Post [2] qui proposait un simulateur de propagation. Je vous invite à consulter rapidement la page pour vous figurer le principe de ce simulateur.

Il s'agit ici d'une simulation "physique". On ne cherche pas à mettre en équation un phénomène, mais à le simuler de manière "réaliste" pour en analyser le comportement, et faire des raisonnements prospectifs.

Dans ce projet, vous allez mettre en place un simulateur simple pour procéder au même type d'analyse. L'objet du projet est de faire le simulateur, et non de réaliser l'analyse du modèle sous-jascent.

Le simulateur que je vous propose de réaliser comporte les éléments suivants :

- une Zone rectangulaire représente l'espace dans le lequel évolue des agents. La zone est un espace Euclidien (chaque agent à une coordonnées (x, y)). En particulier, ce n'est pas une grille.
- un ensemble d'agents qui circulent dans la zone :
  - un agent peut être infecté ou non
  - un agent se déplace toujours à la même vitesse et dans une direction fixe (sauf lorsqu'il rencontre un bord de la zone où il rebondit dessus)
  - une proportion d'agent est confiné (i.e., qu'ils ne peuvent pas bouger)
  - contrairement à l'illustration du Washington Post, les agents ne rebondissent pas les uns sur les autres, ils se croisent et s'ignorent
- un agent est contaminé lorsqu'il se trouve à moins de x mètres d'un agent déjà contaminé (on prendra par defaut $x = 1$)
- le simulateur fonctionne par pas de temps successifs. À chaque pas de temps, le simulateur
  1. déplace chacun agent selon son vecteur de déplacement
  2. évalue les nouvelles contaminations
- le simulateur s'arrêtera lorsqu'un nombre de pas de temps maximum est atteint ou bien que tous les agents sont affectés

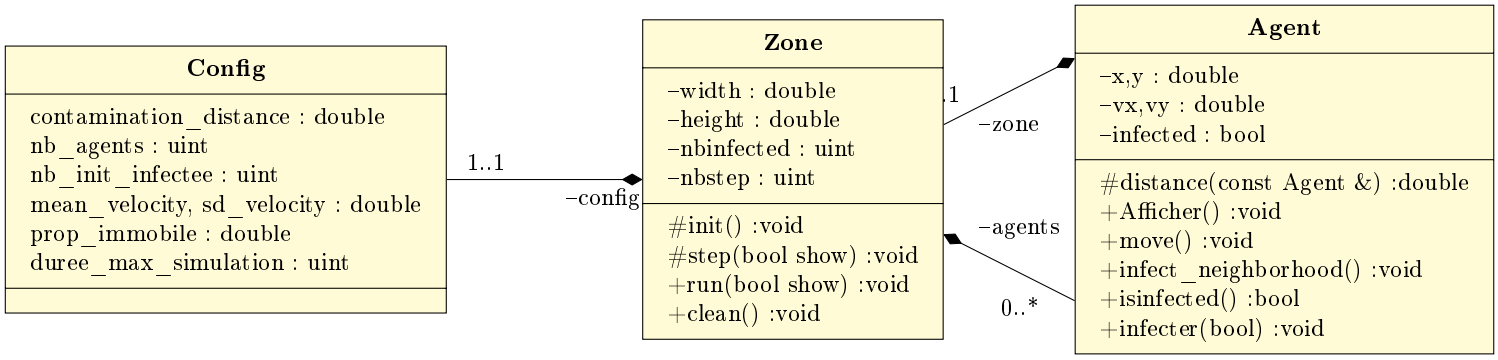
Malheureusement, il serait trop complexe de mettre en place une visualisation. Le simulateur se contentera de faire "tourner" la simulation en affichant l'évolution du nombre d'agent contaminés en fonction des étapes pour établir des vitesses de propagation.

L'intérêt du simulateur sera de pouvoir observer ces évolutions en fonction des paramètres de la simulation. On espère ainsi observer que la propagation est ralentie par des mesures de confinement (préventive ou

2 Étapes de réalisation du projet

Pour mener à bien le projet, je recommande de le réaliser par étapes, de sorte à construire et tester au fur à mesure votre simulateur.

2.1 Proposition de modélisation



2. https://www.washingtonpost.com/graphics/2020/world/corona-simulator/

2.2 Un agent qui bouge

1. implémenter la classe **Config** dans laquelle tous les attributs membres seront en public. Le but de cette classe est juste de contenir tous les paramètres de la simulation qui pourront être utilisés (les valeurs utilisées pour ma correction sont données en fin de sujet). Cette classe n'a pas de méthode.
2. créer une classe **Agent** avec les fonctionnalités de mouvement (sans celle des infections, ni de référence à la zone)
   - un agent fait référence à une **Zone**
   - les attributs **x** et **y** donnent les coordonnées dans la zone de l'agent
   - les attributs **vx** et **vy** donnent les composantes de vecteur de déplacement de l'agent. À chaque nouvelle étape, l'agent se déplace à la position nouvelle $(x+vx, y+vy)$ (il faudra aussi prendre en compte les rebonds sur les bords de la zone). C'est l'opération réalisée par la fonction **move**()
   - la fonction **afficher** () doit permettre d'afficher dans **std**::**cout** les coordonnées de l'agent (sans retour à la ligne)
   - le constructeur de la classe aura le profil suivant : **Agent**(**Zone** &, **double**, **double**, **double**, **double**) qui initialisera les attributs d'un nouvel agent.
3. créer une classe **Zone** avec uniquement 1 agent, avec toutes les fonctions
   - une **Zone** est caractérisée par sa taille, par une configuration et deux attributs outils
   - vous ajouterez un *getter* pour **_config**
   - la **Zone** contient, dans cette section, un unique agent. Pour la suite, **j'impose que les agents de la classe Zone soient sous la forme d'un pointeur**.
   - le constructeur de la classe (**Zone**(**double w**, **double h**)) initialisera les variables **nbstep** et **nbinfected**. Le pointeur vers l'agent sera mis à **nullptr**.
   - la fonction **init** () créera une instance d'un **Agent**. Il sera alors nécessaire d'avoir recours à de l'allocation dynamique (utiliser un **new**). La position de l'agent sera au centre du plateau, et vous pourrez prendre comme vecteur de déplacement $(0.2, 0.85)$. **init** () mettra également les attributs **nbstep** et **nbinfected** à 0.
   - la fonction **step**(**bool**) réalise une étape du simulateur. Ici, il s'agit de déplacer l'agent. Le paramètre doit permettre d'activer ou non l'affichage sur **std**::**cout**.
   - la fonction **run**(**bool**) est la fonction qui implémente la *boucle de simulation*. Cette boucle commence par l'appel à **init** () puis réalise en boucle l'opération **step**() en mettant à jour **nbstep**. Le paramètre est similaire à celui de **step**(). La fonction **run**() pourra afficher l'état final de la zone et le nombre d'étapes réalisées.
   - la fonction **clean**() réinitialise la simulation : elle remet à zéro les attributs et détruit l'agent si nécessaire.
4. implémenter une fonction principale qui va créé une zone, et lancer la simulation (en affichant les positions successives de l'agent).

L'objectif est de mettre en place les classes du projet et de s'assurer dans un premier temps qu'un agent se balade bien (sans déborder) de la zone.

Quelques remarques/aides :

- On constate qu'il y a une dépendance circulaire entre **Agent** et **Zone**, le jeu des **#include** pose problème dans ce cas, et il faut procéder de la manière suivante :
  - l'implémentation de la classe **Zone** se fera normalement, en particulier, vous pourrez ajouter **#include** `"Agent.h"` dans votre préambule.
  - l'implémentation de la classe **Agent** n'utilisera pas le même procédé, les fichiers `Agent.h` et `Agent.cpp` auront la forme ci-dessous. Il faudra alors implémenter les fonctions de la classe dans le fichier `Agent.cpp`.

Fichier `Agent.h` :

```
class Zone; //Ne pas include "Zone.h"

class Agent {
private:
  Zone &zone;
  (...)
```

```
public:
  Agent(...);
  (...)
};
```

Fichier `Agent.cpp` :

```
#include "Agent.h"
#include "Zone.h"

Agent::Agent(...) {
  ...
}
```

2.3 Faire bouger une collection d'agents (aléatoires)

Dans cette section, l'objectif est de modifier les implémentations précédentes pour qu'à la place de faire bouger un unique agent, la simulation fasse bouger une collection d'agents.

Les modifications à apporter sont les suivantes :

- modifier la classe **Zone** pour y ajouter un membre **std**:: **list <Agent *> _agents** qui soit une liste d'agents.
- vous ajouterez un *getter* pour accéder en modification à la liste des agents depuis l'extérieur de la classe [3]
- modifier la fonction **Zone**::**step**() pour qu'elle réalise le déplacement de tous les agents. Dans cette fonction, vous pourrez prévoir l'affichage de chaque agent.
- modifier la fonction **Zone**::**init** () pour qu'elle initialise de manière **_config.nb_agents** à des positions aléatoires dans la zone (tirages uniformes d'une valeur de x et d'une valeur de y), et avec des vecteurs aux directions aléatoires (tirage uniforme d'un angle, et tirage selon une loi normale de la norme du vecteur de déplacement).
- modifier également la fonction **clean**() en conséquent (suppression de tous les agents)

Vous testerez de nouveau votre simulateur pour vérifier son bon fonctionnement avec les agents.

2.4 Contamination des agents

- Dans la classe **Agent**
  - ajouter l'attribut **_infected** ainsi qu'un *setter* et un *getter*. Par défaut, un agent n'est pas injecté.
  - modifier la fonction d'affichage d'un agent pour qu'elle affiche une étoile si l'agent est infecté (rien sinon).
  - ajouter une fonction **double distance(const Agent &a)** qui calcule la distance Euclidienne entre deux agents
  - ajouter une fonction **infect_neighborhood()const** qui, si l'agent est infecté, parcourt tous les agents de la zone et contamine chaque agent qui se trouve à une distance inférieure à **Config**::**contamination_distance**.
- Dans la classe **Zone**
  - modifier la fonction **step**() pour qu'elle évalue les nouvelles contaminations après avoir déplacé tous les agents.
    De plus, vous ferez en sorte de mettre à jours la variable **nbinfected** et d'arrêter la boucle lorsque tous les agents sont infectés.
    Vous afficherez alors en fin de simulation, le nombre de boucle qui ont été nécessaire avant contamination totale.
  - modifier la fonction **init** () pour qu'elle attribut l'injection à **_config.nb_init_infectee** agents

3 Comparer différentes configuration du simulateurs

Maintenant que le simulateur est en place, vous pouvez le faire fonctionner avec des configurations différentes.

Dans l'idée, le programme principal pour relancer plusieurs fois une simulation avec les mêmes caractéristiques pour obtenir une durée moyenne d'infection de la totalité de la population. Vous pourrez alors observer les différences avec des distances de contamination plus ou moins grandes.

3. Ce n'est pas extrêmement propre comme modélisation, mais la bonne version devient un peu compliquée.

Pour étudier l'intérêt du confinement, il faut modifier légèrement la fonction **Zone**::**init**() pour que, lors de la création des agents, il y ait une proportion **_config.prop_immobile** d'agent qui ne bougent pas (vecteur de déplacement nul).

Si vous êtes arrivez à cette étape en ayant tout parfaitement accomplis, alors vous aurez une note maximale de 17/20. Les points complémentaires vont pour des réalisations additionnelles que vous êtes libres de proposer. La section suivante donne des idées d'extention

4 Extensions

— Le mouvement de l'agent est plutôt simple et tous les agents se déplacent de la même manière, vous pourriez alors implémenter des classes qui héritent de la classe **Agent** et qui spécialise le mouvement. Par exemple, vous pourriez implémenter un mouvement browniens dont l'intensité est proportionnelle à la norme du vecteur de déplacement.

— Ajouter un paramètre de contagiosité qui permette d'indiquer si un agent infecté est contagieux ou non. Dans l'idée, un agent infecté à la date t ne sera plus contagieux à partir de la date $t + 14$ (et ne pourra plus l'être)

— Donner la possibilité de tester l'hypothèse de confiner (i.e., ne plus bouger l'agent) à partir du moment où un agent est contaminé. Pour représenter mieux la réalité, on ajoutera un paramètre qui donnera la probabilité d'être confiné lorsqu'on est injecté (probabilité d'être diagnostiqué)

— tenir compte du temps de croisement : pour être injecté, il faut être exposé à moins d'un mètre pendant trois pas de temps.

— modéliser la charge virale d'un agent de manière continue (plus on l'infecte, plus elle augmente et plus il est contaminant)

— récupérer les données d'évolutions de l'épidémie en format CSV pour afficher des courbes en `R` ensuite ...

5 Valeurs des paramètres par défaut

Pour la zone, j'ai pris une zone de taille 10×10.
Pour la configuration, on propose les valeurs de paramètre par defaut suivants :

```
contamination_distance=1;
nb_agents=10;
nb_init_infectee=1;

mean_velocity=1;
stdev_velocity=0.3;

prop_immobile=0.6;

duree_max_simulation=10000;
```

NB : ces valeurs sont indicatives pour commencer ... sur ma correction, elles permettent d'observer des différences de comportement en fonction, soit de la distance de contamisation, soit de la proportion de personnes immobiles.
Vous pouvez tout à fait utiliser d'autres configurations.