Introduction au C++

Robotique/UV4.8 - Module Middleware #1 - Mars 2019

Supports de cours disponibles sur
`www.simon-rohou.fr/cours/c++`

L'objectif de ce TD est d'implémenter en C++ l'exercice 4.9 *Platooning* du cours de Robotique Mobile, qui avait été étudié en Python.

A. Des embouteillages sur une route circulaire

On considère $n = 15$ robots tournant sur une route circulaire de circonférence $l = 100$ et de rayon $r = {}^{l}/_{2\pi}$. Chaque robot $\mathcal{R}_i$ se décrit par les équations d'état suivantes :

$$\begin{cases} \dot{x_i} &= v_i, \\ \dot{v_i} &= u_i. \end{cases} \tag{1}$$

Le vecteur d'état de chaque robot est $(x_i, v_i)^\intercal$ où x_i correspond à la position du robot et v_i à sa vitesse. Chaque robot $\mathcal{R}_i$ est équipé d'un radar retournant la distance d_i au robot $\mathcal{R}_{i-1}$ positionné devant lui sur le cercle, ainsi que la dérivée $\dot{d_i}$.

Le but de cet exercice est de rendre les robots autonomes afin qu'ils évoluent tous sur le cercle à grande vitesse et sans embouteillage, contrairement à la scène représentée Figure 1.

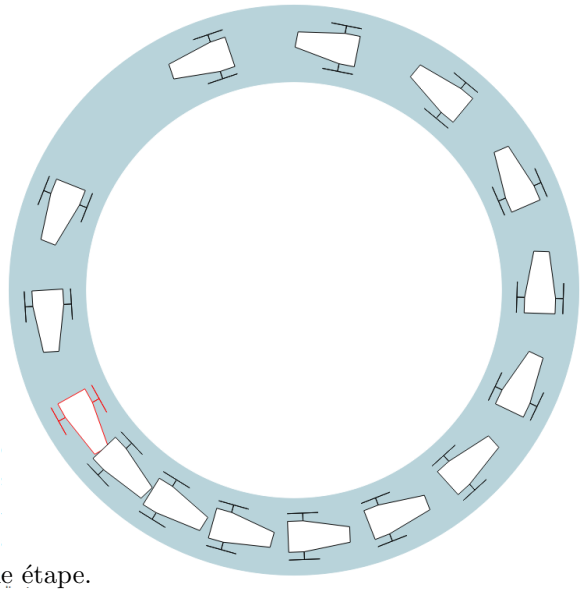

Figure 1: Embouteillages sur le cercle.

Important : vous devez compiler votre programme à chaque étape.

B. Initialisation du projet

1. Dans un nouveau répertoire, créer un fichier `main.cpp` déclarant les constantes $n = 15$, $\ell = 100$, $\delta = 0.05$, $d_0 = {}^{\ell}/_{n}$ et $v_0 = 3$.
2. Créer un fichier `CMakeLists.txt` contenant les directives pour `cmake`. En particulier :
   - renseigner la version minimum de `cmake` à 3.5
   - spécifier le C++ standard à 11
   - nommer le projet `platooning`
   - lister le premier fichier à compiler `main.cpp`
   - générer l'exécutable `ex_platooning` à partir de ce premier fichier source

C. La route sans fin

3. Dans de nouveaux fichiers, créer la classe `Road` ayant pour paramètre constant sa circonférence ℓ. La configuration `cmake` devra permettre la compilation de cette classe.
4. Instancier un objet de type `Road` dans le programme principal, et compiler.

5. En respectant le *Const Correctness*, ajouter deux méthodes de classe publiques de type *accesseurs* :
   - `length()`, qui renvoit directement la variable de classe correspondant à ℓ ;
   - `radius()`, qui calcule le rayon de la route.

D. Une voiture non-régulée

6. Dans de nouveaux fichiers, créer la classe `Car` ayant pour variables de classe :
   - l'adresse (par pointeur) de l'objet de la classe `Road`, en lecture seulement, car une voiture ne peut pas modifier la route ;
   - la position x de la voiture sur le cercle ;
   - la vitesse v de la voiture, nulle par défaut.
7. Créer les accesseurs pour x et v ainsi qu'un mutateur `set()` pour écrire simultanément ces deux valeurs. Enfin, créer un méthode `stop()` arrêtant la voiture.
8. Une voiture ne connaît que le véhicule se trouvant devant elle. On ajoutera les méthodes `set_front_car()` et `front_car()` renseignant et retournant l'adresse par pointeur en lecture de ce véhicule.

E. Programme principal

9. Dans le programme principal `main.cpp`, instancier une route de longueur ℓ et un ensemble de n voitures stockées dans un conteneur `vector`. Chaque véhicule $\mathcal{R}_i$ sera positionné en $x_i = 4i$.
10. Parcourir ce conteneur pour renseigner la *front car* de chaque véhicule.

F. Affichages graphiques

Le logiciel VIBes a été développé à l'ENSTA Bretagne afin de fournir des fonctionnalités graphiques simples et décorrélées du programme métier. Il va nous permettre d'afficher la `Road` et les n `Car` s'y trouvant. VIBes est un exécutable à lancer en parallèle d'un projet. Ce dernier devra envoyer des instructions graphiques à VIBes pour afficher des objets.

F1. Installation

Les sources de VIBes se trouvent sur le dépôt `https://github.com/ENSTABretagneRobotics/VIBES`.

11. Dans un autre répertoire, clonez le dépôt avec la commande suivante :
    `git clone https://github.com/ENSTABretagneRobotics/VIBES`
    (utiliser si besoin `git config --global http.proxy http://adresseduproxy:8080`)
12. Installez les paquets nécessaires :
    `sudo apt-get install qt5-default libqt5svg5-dev`
13. Les instructions graphiques forment une API (Application Programming Interface) permettant de communiquer avec VIBes. Elles se présentent sous la forme de fonctions C++ déjà implémentées. Copiez les fichiers de `./VIBES/client-api/C++/src/` dans votre répertoire de projet `platooning`.
14. L'exécutable VIBES doit être compilé indépendamment. Placez-vous dans `./VIBES/viewer/` et compilez le projet en utilisant `cmake`. Lancez l'exécutable `VIBes-viewer`.

F2. Afficher la route

15. Une seule route sera affichée dans la vue de notre simulation. On initialisera donc VIBes dans le constructeur de la classe `Road` avec les instructions suivantes :

```
vibes::beginDrawing();
vibes::newFigure("Jam");
vibes::setFigureProperties("Jam",
                           vibesParams("x", 100, "y", 100,
                                       "width", 400, "height", 400));
vibes::axisLimits(-20., 20., -20., 20.);
```

    et on quittera proprement et automatiquement l'affichage avec la commande `vibes::endDrawing();` lorsque l'objet sera détruit.

16. À l'aide de la fonction `vibes::drawCircle()`, afficher deux disques concentriques pour dessiner la route. Ces disques seront dessinés dans une nouvelle méthode `draw()` de la classe `Road` et on veillera à réinitialiser la figure avant leur affichage avec `vibes::clearFigure("Jam");`

F3. Afficher une voiture

17. La classe `Car` va elle aussi disposer de sa méthode `draw()`. Dessiner un tank à l'aide de la commande `vibes::drawTank(px, py, theta + (M_PI/2.), length)`. La position angulaire θ du véhicule sur le cercle est donnée par $\theta = {}^{x}/_{r}$ où r est le rayon de la route. Cette dernière information est accessible par le *const pointer* renseigné à la création de l'objet de classe `Car`.

18. La méthode `draw()` de la classe `Road` va directement afficher les voitures sur la route. Ajouter en paramètre de la fonction un vecteur de voitures à afficher (et donc en lecture seule), et itérer les objets pour les dessiner par dessus les disques.

19. Afficher la route dans le programme principal : les voitures devraient apparaître.

G. Distances circulaires et collisions

20. Dans `Road.cpp`, implémenter la fonction `sawtooth()`. Elle ne devra pas nécessairement figurer comme méthode de classe. Rappel :

$$\text{sawtooth}(x) = 2\arctan(\tan({}^{x}/_{2})) \tag{2}$$

21. Dans la classe `Road`, créer la méthode `circular_dist()` calculant la distance d_i entre deux véhicules sur la route :

$$d_i = r \cdot \left(\text{sawtooth}(-\pi + {}^{(x_{i-1}-x_i)}/_{r}) + \pi\right) \tag{3}$$

22. Implémenter une méthode retournant `true` si une voiture est en collision avec le véhicule se trouvant devant elle. On configurera la longueur constante d'une voiture à 4.

H. Simulation

23. Dans la classe `Car`, ajouter une méthode représentant la fonction d'évolution du système. Son prototype, `void f(float u, float& xdot, float& vdot) const`, retournera par arguments les composantes de la dérivée du vecteur d'état du véhicule. Se référer à l'Équation (1).

24. Implémenter également une méthode `float u(float d0, float v0) const`, retournant la commande du système en fonction d'une consigne de distance d_0 et vitesse v_0 nominales données. Dans le cas de la classe `Road`, non autonome, on retournera $u = 1$.

25. Dans le programme principal, simuler le système de $t = 0$ à $t = 100$ par pas de temps δ en utilisant les méthodes de la classe `Car` déjà implémentées.

26. Dans la boucle de simulation, arrêter tous les véhicules entrés en collision avec leurs voisins frontaux, et les afficher en rouge.

I. Voiture autonome

27. La circulation sur cette route peut être améliorée en rendant les véhicules autonomes. On considère la nouvelle commande u donnée en proportionnelle et dérivée par

$$u = (d_i - d_0) + (v_i - v_{i-1}) + (v_0 - v_i), \tag{4}$$

où v_{i-1} est la vitesse du véhicule se trouvant devant la voiture à réguler. Proposer une nouvelle classe `AutonomousCar` implémentant cette loi de commande, et héritant de la classe `Car`.

28. Dans le programme principal, remplacer `Car` par son équivalent autonome. Il sera nécessaire de faire un *cast* pour la méthode `draw()` permettant l'affichage de la route.

J. (optionnel) Performances du C++ par rapport à Python

29. En lançant plusieurs simulations avec différentes valeurs de n et ℓ suffisamment grandes, évaluer les performances du programme compilé en C++ par rapport à son équivalent en python.

K. (optionnel) Calcul vectoriel

30. En utilisant la bibliothèque Eigen, définir l'état d'une voiture comme un seul objet $\mathbf{x} \in \mathbb{R}^2$ de type `Eigen::Vector2d`. Voir la documentation : `https://eigen.tuxfamily.org/dox/index.html`

L. (optionnel) Afficher deux routes de circulation

31. Proposer une implémentation simulant simultanément deux routes dans deux fenêtres distinctes : n voitures seront non régulées (loi de commande $u = 1$) sur la première route, et n autres régulées sur la seconde.

M. (optionnel) Création d'une bibliothèque

32. Configurer `cmake` afin de séparer la simulation (`main.cpp`) des classes `Road`, `Car` et `AutonomousCar` qui pourraient constituer une bibliothèque `roblib` indépendante.