C++ / Programmation orientée objet

Robotique/UE 3.1 - TD 02 - 2025/2026

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

L'objectif de ce TD est de se familiariser avec les notions d'objets et de classes en complétant l'exercice *Platooning* vu au TD précédent.

A. Des embouteillages sur une route circulaire

On considère désormais $n = 15$ robots tournant sur une route circulaire de circonférence $\ell = 100$ et de rayon $r = \ell/2\pi$. Pour rappel, 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 sur la route (en abscisse curviligne) et v_i à sa vitesse. Chaque robot $\mathcal{R}_i$ évoluant sur le cercle, les valeurs $x_i = 0$ et $x_i = 2\pi r = \ell$ correspondent à la même position sur le cercle. Chaque véhicule est désormais équipé d'un radar retournant la distance d_i au robot $\mathcal{R}_{i-1}$ positionné devant lui sur le cercle.

Le but de cet exercice est de rendre les robots autonomes afin qu'ils évoluent tous sur le cercle à grande vitesse v_0 et sans embouteillage, contrairement à la scène représentée Figure 1.

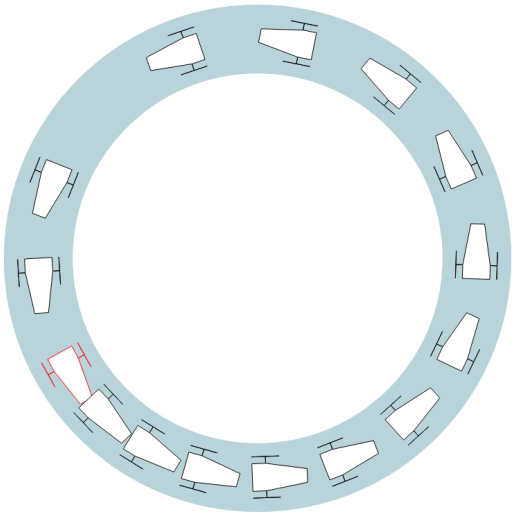

Figure 1: Embouteillages sur le cercle.

Objectif du TD : la programmation objet va permettre une implémentation simplifiée des robots et de leur environnement. Le code développé au TD 1 peut être repris dans ce TD et organisé par classes.

B. La route sans fin

1. Reprendre les sources développées au TD précédent.
2. Dans de nouveaux fichiers, créer la classe `Road` ayant pour paramètre constant sa circonférence ℓ.
3. 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 `vibes::beginDrawing()`, `vibes::newFigure(..)`, *etc.*, vues au TD précédent.
4. On quittera proprement et automatiquement l'affichage avec la commande `vibes::endDrawing()` lorsque l'objet `Road` sera détruit.
5. L'affichage des disques de la route se fera dans une nouvelle méthode `draw()` de la classe `Road` (on veillera à réinitialiser la figure avant leur affichage).
6. Instancier un objet de type `Road` dans le programme principal, et compiler.
7. En respectant la *Const Correctness*, ajouter deux méthodes de classe, publiques et de type *accesseurs* :
   - `length()`, qui renvoie directement la variable de classe correspondant à ℓ ;
   - `radius()`, qui calcule le rayon de la route.

C. Une voiture non-régulée

8. 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.
9. Créer les accesseurs pour x et v ainsi qu'un mutateur `set()` pour écrire simultanément ces deux valeurs. Enfin, créer une méthode `stop()` arrêtant la voiture.
10. 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.
11. La classe `Car` va elle aussi disposer de sa méthode `draw()`. Dessiner un tank[1] à l'aide de la commande `vibes::drawTank(px, py, (theta+M_PI/2.) * (180./M_PI), length)`. On rappelle que 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`.

D. Programme principal

12. 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 $x_i = 4i$.
13. Parcourir ce conteneur pour renseigner la *front car* de chaque véhicule. On utilisera le modulo `(i+1)%n` pour lier la dernière voiture à la première pendant l'itération.
14. 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.
15. Afficher la route dans le programme principal : les voitures devraient apparaître.

E. Distances circulaires et collisions

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

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

17. Dans la classe `Road`, créer la méthode `circular_dist(...)` calculant la distance d_i entre deux véhicules (passés en argument de la méthode) sur la route :

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

18. 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.

F. Simulation

19. 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).

[1] Si la commande `drawTank()` n'existe pas dans votre version de VIBes, utilisez `drawVehicle()` avec les mêmes paramètres.

20. 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 `Car`, non autonome, on retournera directement $u = 1$.

21. 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. La commande `usleep()` permettra un affichage fluide à l'écran :

```
#include <unistd.h> // pour usleep
...
usleep(dt * 500000.); // vitesse d'animation
```

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

G. Voiture autonome

23. 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

$$u = (d - d_0) + (v_{i-1} - v_i) + (v_0 - v_i), \quad \text{où :} \tag{4}$$

   - v_{i-1} est la vitesse du véhicule se trouvant devant la voiture à réguler ;
   - d est la distance entre les deux véhicules $\mathcal{R}_i$ et $\mathcal{R}_{i-1}$;
   - v_0 et d_0 sont les vitesse et distance de consigne.

   Proposer une nouvelle classe `AutonomousCar` implémentant cette loi de commande, et héritant de la classe `Car`.

24. 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 :

```
road.draw((const vector<Car>&)v_cars);
```

25. Visualiser le trafic régulé.

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

26. 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 vu en TD de robotique mobile.

I. (optionnel) Afficher deux routes de circulation

27. 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.