# C++ / Évaluation « Convoi de robots »

Robotique/UE 3.1 – Janvier 2021

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

## A. Présentation

On considère un *convoi*[1] de $n = 6+1$ robots décrits par les équations d'état suivantes :

$$\mathcal{R} \begin{cases} \dot{x} &= \vartheta \cos(\theta) \\ \dot{y} &= \vartheta \sin(\theta) \\ \dot{\theta} &= u_1 \\ \dot{\vartheta} &= u_2 \end{cases} \tag{1}$$

Le vecteur d'état de chaque robot est donc $\mathbf{x} = (x, y, \theta, \vartheta)^\intercal$, avec $\theta$ son cap (heading) et $\vartheta$ sa vitesse.

On cherche à simuler un convoi [1], comme illustré sur la Figure 1. Un robot *leader* nommé $\mathcal{R}_a$ suit une loi de commande donnée. Chaque autre robot *suiveur* doit suivre exactement le même chemin que le robot *leader*. Pour des raisons de sécurité, on souhaite que la distance *curviligne*[2] entre chaque robot soit $d = 5\text{m}$ en toutes circonstances. Le *leader* pouvant subir des changements d'accélération, on souhaite éviter un convoi évoluant en accordéon, c'est à dire avec un retard dans les accélérations de chaque *suiveur* par rapport au *leader*.

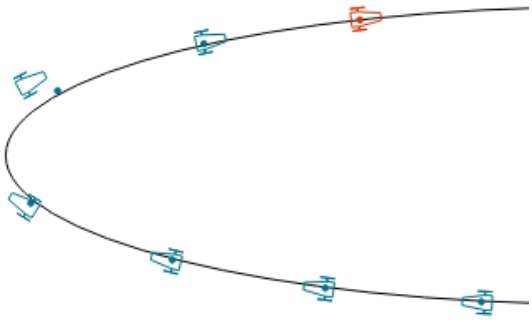

FIGURE 1 – Un convoi de plusieurs robots suivant un *leader*, dessiné en rouge.

En d'autres termes, tous les robots *suiveurs* sont contrôlés par une loi de commande basée sur :

- les positions passées du *leader*. Pour cela, un historique des états précédents du *leader* est enregistré et accessible à tous les *suiveurs* ;
- la vitesse courante du *leader*, pour éviter l'effet accordéon rencontrés dans les embouteillages.

## Critères d'évaluation

- Développement des questions
- Organisation propre des sources (indentation, nom des variables, fonctions)
- Commentaires et lisibilité du code
- Respect de la *Const Correctness*
- Pas de fuite mémoire
- *Malus :* rendu de votre exécutable binaire ou de fichiers temporaires ou compilés (`.o`, *etc.*)

1. *convoi* : groupe de véhicules qui font route ensemble.
2. *curviligne* : c'est à dire le long de la courbe, de la trajectoire suivie.

*Lisez attentivement les questions suivantes, et implémentez-les dans l'ordre, en compilant votre programme systématiquement.*

# B. Typedef

On choisit de représenter des vecteurs 2d en utilisant des tableaux de `double` de deux éléments. En C++, on peut utiliser l'instruction `typedef` pour déclarer un alias de type (donner un autre nom à un type déjà existant), et améliorer la lisibilité du code. C'est ce qui est fait dans le fichier `Robot.h`, avec la création d'un type `vec2d` basé sur un tableau :

```
typedef double vec2d[2];
```

Ce type est utilisé dans `Robot.h`. On utilise ainsi les variables issues de ce type exactement de la même manière qu'avec un tableau. Par exemple :

```
vec2d u;
u[0] = 0.; u[1] = 1.;
```

# C. Commentaires

1. Toutes les fonctionnalités développées ci-après devront être commentées.

# D. Un robot statique

2. Télécharger les fichiers `Robot.h`, `vibes.cpp` et `vibes.h` sur le site web du cours.
   La classe `Robot` est pré-définie avec :
   - un constructeur, qui créera un robot ayant pour état $\mathbf{x} = (0, 0, 0, 1)^\intercal$, (*i.e.* une vitesse de 1).
   - `set_position(..)` plaçant un robot à la position $(x, y)$.
   - `v()` retournant la vitesse courante $\vartheta$ du robot.
   - `draw(..)` dessinant un robot à l'aide de VIBes.
   - `u(w, dw, u)` calculant la commande $\mathbf{u}$ (retour par référence) à partir de deux vecteurs $\mathbf{w}$ et $\dot{\mathbf{w}}$ passés par valeur. $\mathbf{w}$ est le waypoint à suivre, et $\dot{\mathbf{w}}$ sa dynamique.
   - `euler(dt,u)` mettant à jour l'état $\mathbf{x}$ du robot par intégration sur un pas de temps $dt$, à partir de la commande $\mathbf{u}$.
3. Compléter les définitions des méthodes de `Robot.h` afin de respecter la *const correctness*.
4. Dans `Robot.h`, proposer un type complexe (une structure), nommée `State`, permettant d'implémenter un vecteur d'état.
5. Ajouter un attribut de ce type à la classe `Robot`. Il ne doit pas être possible de modifier cet état directement depuis l'extérieur de la classe.
6. Implémenter le constructeur ainsi que les trois méthodes : `set_position(..)`, `v()` et `draw(..)`.
7. Dans le programme principal, créer un tableau `Ri` de $n$ robots, chaque robot étant ensuite positionné à $(-i, 0)$.
8. Afficher les robots dans une vue VIBes. On prendra les paramètres suivants :

```
vibes::beginDrawing();
vibes::newFigure("Convoy");
vibes::setFigureProperties("Convoy",
      vibesParams("x", 100, "y", 100, "width", 900, "height", 500));
vibes::axisLimits(-25.,25.,-13.89,13.89);
```

# E. La simulation

9. La simulation s'étale sur $t_{\max} = 100$ unités de temps avec un pas de temps $dt = 0.005$. Le groupe de robots *suiveurs* est constitué de $n = 6$ robots. Les robots se tiennent éloignés d'une distance curviligne $d = 5$. Le robot *leader* suivra une trajectoire elliptique de paramètres $L_x = 20$ et $L_y = 5$. Ajouter ces constantes dans le programme principal.
10. Utiliser une méthode d'Euler pour intégrer les équations d'état des robots, en implémentant la méthode `euler` de la classe `Robot`.
11. Dans le programme principal, simuler les 6 robots de $t_0$ à $t_{\max}$ avec une commande $\mathbf{u} = (0.2, 0)^\intercal$.
    On mettra le programme en pause à la fin de chaque itération :

    ```
    usleep(1000.); // animation speed (microseconds)
    ```

    On veillera également à rafraîchir l'écran avec :

    ```
    vibes::clearFigure("Convoy");
    ```

    Les six robots doivent alors décrire des cercles.

# F. La commande

12. On implémente maintenant la méthode définissant la loi de commande permettant de rejoindre un point $\mathbf{w}$ ayant pour dynamique $\dot{\mathbf{w}}$. Le code de la loi de commande obtenue par bouclage linéarisant est donné ci-dessous, et à compléter pour s'adapter à votre structure d'état (`State`) :

```
{
   // --- Complete this ---
   const double &px = ..
   const double &py = ..
   const double &theta = ..
   const double &v = ..
   // ---------------------

   double A[2][2] = {
     { -v*sin(theta), cos(theta) },
     { v*cos(theta), sin(theta) }
   };

   double det_inv_A = A[0][0]*A[1][1]-A[0][1]*A[0][1];

   if(det_inv_A == 0.)
   {
     cout << "Error for matrix inversion" << endl;
     exit(1);
   }

   det_inv_A = 1. / det_inv_A;

   double inv_A[2][2] = {
     { det_inv_A*A[1][1], -det_inv_A*A[0][1] },
     { -det_inv_A*A[1][0], det_inv_A*A[0][0] }
   };

   vec2d y = { px, py };
   vec2d dy = { v*cos(theta), v*sin(theta) };
   vec2d e = { w[0]-y[0]+dw[0]-dy[0], w[1]-y[1]+dw[1]-dy[1] };

   u[0] = inv_A[0][0]*e[0] + inv_A[0][1]*e[1];
   u[1] = inv_A[1][0]*e[0] + inv_A[1][1]*e[1];
}
```

## G. Le *leader* et son passé...

13. Dans de nouveaux fichiers, créer une nouvelle classe `RobotLeader` héritant de `Robot`. On propagera les droits d'accès de `Robot` à sa classe dérivée.
14. Surcharger la méthode `draw` afin d'afficher automatiquement le robot *leader* en rouge (la méthode `RobotLeader::draw()` n'a donc pas de paramètre).
15. Ajouter dans cette classe, en attribut privé, une structure de données de la bibliothèque standard, nommée `history_x`, permettant d'enregistrer un ensemble d'états `x` passés.
16. On cherche à garder une copie de chaque état de ce robot toutes les $ds = 0.1$ unités de distance parcourue le long de sa trajectoire. Ajouter pour cela un second attribut privé `ds`. Surcharger la méthode `euler` pour intégrer l'état et incrémenter `ds` par intégrations successives de $\vartheta$ (la vitesse du robot). `ds` représente donc la distance curviligne parcourue. Toutes les 0.1 unités de distance parcourue, enregistrer l'état courant du robot dans `history_x`, et remettre `ds` à 0.
17. Enfin, ajouter une méthode `RobotLeader::states_history()` retournant en lecture l'ensemble des états enregistrés. On veillera à respecter la *const correctness*.

## H. La trajectoire elliptique

Le *leader* $\mathcal{R}_a$ suit une trajectoire elliptique en cherchant à suivre le point $\mathbf{w}_a = \begin{pmatrix} x_a \\ y_a \end{pmatrix}$ :

$$\mathbf{w}_a : \begin{cases} x_a = L_x \sin(\omega t) \\ y_a = L_y \cos(\omega t) \end{cases} \qquad \dot{\mathbf{w}}_a : \begin{cases} \dot{x}_a = \omega L_x \cos(\omega t) \\ \dot{y}_a = -\omega L_y \sin(\omega t) \end{cases} \tag{2}$$

18. Dans le programme principal, créer un *leader* `Ra`.
19. Dans la boucle de simulation, dessiner l'ellipse et définir ces deux vecteurs $\mathbf{w}_a$ et $\dot{\mathbf{w}}_a$, avec $\omega = 0.1$.
20. Calculer la loi de commande $\mathbf{u}_a$ du *leader* en fonction de $\mathbf{w}_a$ et $\dot{\mathbf{w}}_a$, et simuler son déplacement sur l'ellipse. Afficher éventuellement le point à suivre $\mathbf{w}_a$ dans la vue.

## I. Le convoi

Chaque robot *suiveur* $\mathcal{R}_i$ va suivre un point $\mathbf{w}_{a_j}$ correspondant à la position d'un précédent état $\mathbf{x}_j$ du *leader*. Comme on cherche à garder une distance de sécurité $d$ entre les robots, on cherche à maintenir une distance $d(i+1)$ entre les robots $\mathcal{R}_i$ et $\mathcal{R}_a$.

21. Dans la boucle de simulation, pour chaque robot $\mathcal{R}_i$, récupérer l'état $\mathbf{x}_j$ de $\mathcal{R}_a$ que doit suivre $\mathcal{R}_i$. On rappelle que la méthode `RobotLeader::states_history()` retourne les états passés, et qu'un nouvel état est enregistré toutes les `ds` distances parcourues. Si cet état n'existe pas encore (*i.e.* si $d(i+1)$ est inférieure à la distance déjà parcourue par $\mathcal{R}_a$ à l'instant $t$), alors on laissera $\mathcal{R}_i$ suivre la commande $\mathbf{u} = (0.2, 0)^\intercal$ précédemment définie (et tourner en rond).
22. Sinon, si $\mathcal{R}_i$ est en mesure de suivre une ancienne position $(\mathbf{x}_{j_x}, \mathbf{x}_{j_y})$ de $\mathcal{R}_a$, alors on calcule les vecteurs :

$$\mathbf{w}_{a_i} : \begin{cases} x_{a_i} = \mathbf{x}_{j_x} \\ y_{a_i} = \mathbf{x}_{j_y} \end{cases} \qquad \dot{\mathbf{w}}_{a_i} : \begin{cases} \dot{x}_{a_i} = \vartheta_a \cos(\mathbf{x}_{j_\theta}) \\ \dot{y}_{a_i} = \vartheta_a \sin(\mathbf{x}_{j_\theta}) \end{cases} \tag{3}$$

où $\mathbf{x}_{j_x}$, $\mathbf{x}_{j_y}$, $\mathbf{x}_{j_\theta}$, sont la position et le cap du leader $\mathcal{R}_a$ à l'état passé $\mathbf{x}_j$, et où $\vartheta_a$ est la vitesse courante de $\mathcal{R}_a$ (à l'instant $t$).

23. Former le convoi.

## Références

[1] Luc Jaulin. *Mobile robotics*. 2015. OCLC : 986557752.