Programmation synchrone (PSYN9)

Soutien au projet
(Version du 16 novembre 2020)

1 Introduction

Ce document récapitule les indications concernant la réalisation du projet fournies en cours et durant les séances de travaux pratique. Nous vous rappelons que le code ainsi que le sujet sont disponibles dans le dépôt Git du cours.

https://gaufre.informatique.univ-paris-diderot.fr/aguatto/progsync-m2-eidd-20-21/

(N.B. : le présent document **ne remplace pas une lecture attentive du sujet**. En cas d'incompatibilité, celui-ci fait autorité.)

2 Récapitulatif du cahier des charges

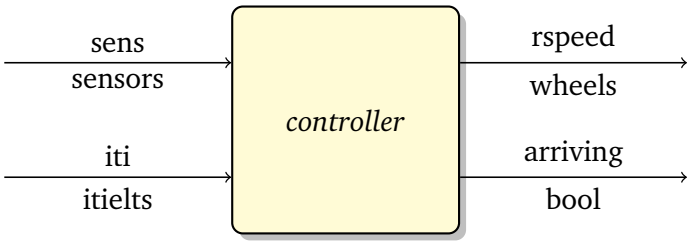
Le but du projet est de programmer en Heptagon un logiciel de guidage pour véhicule autonome. Ce logiciel sera évalué sur sa capacité à suivre des parcours donnés tout en obéissant à des impératifs de sûreté : tenue de route, arrêt aux feux rouges, arrêt d'urgence en cas de présence d'un piéton sur la route, respect des limitations de vitesse.

Le projet est à rendre le **20 décembre 2020** avant 23h50.

3 Indications

3.1 Indications générales

1. Il est fortement recommandé de **suivre la progression spécifiée dans le sujet** :
   https://gaufre.informatique.univ-paris-diderot.fr/aguatto/progsync-m2-eidd-20-21/blob/master/projet/sujet/sujet-projet.pdf
2. Votre code doit être ajouté au nœud *controller* du module *Control*. Vous ne **devez pas modifier le reste du code** du projet, sauf pour des tests sporadiques.
3. L'interface de *controller* est donnée ci-dessous.

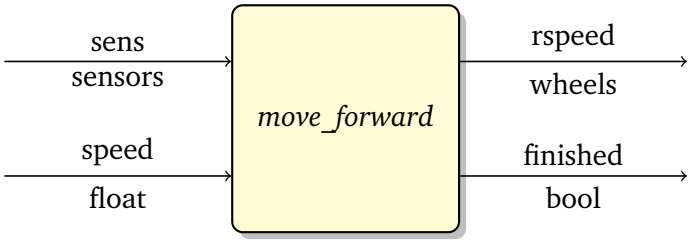


On rappelle que le type *wheels* consiste en un enregistrement à deux champs, *left* et *right*, chacun spécifiant par un flottant 64 bits la vitesse de rotation de la roue motrice correspondante. Cette vitesse est exprimée en **degrés par seconde**.

3.2 Déplacements de base

3.2.1 Déplacement à vitesse constante

1. Il est très utile de disposer d'un opérateur capable de faire avancer votre véhicule en ligne droite et à une vitesse spécifiée jusqu'à la prochaine étape (*waypoint*). Cet opérateur pourrait par exemple avoir l'interface suivante :



   Décrivons brièvement entrées et sorties : *sens* spécifie les entrées des capteurs de la voiture, *speed* est la vitesse du véhicule désirée, exprimée en **centimètres par seconde**, *rspeed* est la vitesse de rotation des roues choisie, et *finished* un booléen qui signale qu'on a atteint le prochain point d'intérêt.

2. La principale tâche de cet opérateur est de convertir *speed* en *rspeed*, la vitesse angulaire des roues. Ici, nous cherchons à avancer en ligne droite, les deux roues doivent donc tourner à la même vitesse. On peut supposer une loi proportionnelle liant la vitesse désirée V et la vitesse angulaire ω de chacune des roues :

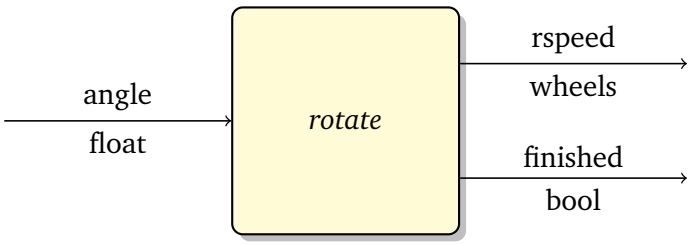
$$\omega = K_v \cdot V.$$

   Le simulateur vous fournissant la vitesse V, il est donc facile d'utiliser l'équation ci-dessus pour déduire la valeur du coefficient K_v expérimentalement, en fixant la valeur de ω.

3. Les points d'intérêt sont signalés par la présence sur la route d'un bandeau d'une couleur particulière. En ce qui concerne les étapes, il s'agit du **vert**. Vous pouvez donc les détecter en inspectant le champ *s_road* de *sens*, de sorte à positionner la sortie *finished*.

3.2.2 Rotation

1. Il est également utile de disposer d'un opérateur capable de faire pivoter votre véhicule d'un angle spécifié relatif à son orientation initiale.



   Ici, *angle* est l'angle de rotation du véhicule désiré exprimé en **degrés**, *rspeed* est la vitesse de rotation des roues choisie, et *finished* un booléen vrai une fois la rotation terminée.

2. Une rotation stationnaire peut être obtenue en faisant tourner une des roues à une vitesse ω et l'autre à une vitesse $-\omega$ pendant un certain temps. La durée Δ_t nécessaire

pour obtenir un angle de rotation α voulu dépend bien sûr du ω choisi. On peut de nouveau supposer que cette dépendance est proportionnelle, et mesurer expérimentalement le temps nécessaire à l'obtention de α pour un ω bien choisi.

$$\alpha = K_r \cdot \Delta_t \cdot \omega$$

3. Une fois le coefficient K_r mesuré, vous pouvez programmer votre opérateur maintenant les roues à ω et $-\omega$ suffisamment longtemps pour obtenir une rotation d'angle α. Le laps de temps physique s'étant écoulé durant un instant logique est déterminé par la constante globale *timestep*.

3.3 Suivi de route

Le but de cette partie du projet est d'être capable d'avancer le long d'une route potentiellement incurvée. Votre véhicule découvre le tracé exact de la route à mesure qu'il progresse le long de celle-ci, et doit donc être capable de corriger sa trajectoire à la volée. La route est équipée d'un **marquage au sol décrit dans le sujet** du projet.

On réalisera cette partie en **modifiant l'opérateur *move_forward*** décrit précédemment.

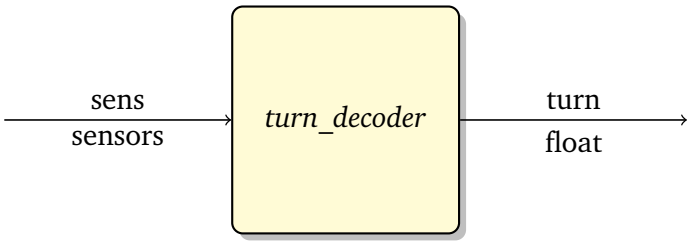
3.3.1 Suivi de route simple

1. En dehors des points d'intérêt, le marquage au sol vous indique à quelle distance du centre de la route la voiture se trouve. Le centre de la route est marqué en **bleu**, le coté gauche en **cyan**, le coté droit en **magenta**. Ces couleurs sont codées par les triplets RGB ci-dessous, définis comme constantes globales dans le code fourni.

	R	G	B
BLUE	0	0	255
CYAN	0	255	255
MAGENTA	255	0	255

N.B. : lorsque votre véhicule se trouve à la frontière entre deux zones de couleurs différentes, il **capte un mélange des dites couleurs**.

2. Il est très utile de réaliser un opérateur qui observe la route et détermine s'il faut tourner à droite, à gauche, ou bien maintenir le cap.



La sortie *turn* code l'action à effectuer. On peut par exemple choisir la convention suivante : le signe de *turn* détermine s'il faut **tourner à droite (signe négatif) ou à gauche (signe positif)**, et **sa valeur absolue indique à quel point l'angle doit être aigu**. Une valeur de zéro indiquera donc qu'il faut avancer tout droit.

3. Une solution naïve serait d'utiliser un bloc conditionnel Heptagon pour fixer *turn* à -1.0 lorsque le champ *sens.s_road* contient la constante *cyan*, à 1.0 lorsqu'il contient la constante *magenta*, et à 0.0 sinon. Toutefois, cette solution néglige les cas où le véhicule se situe à la frontière entre deux zones.

   Supposons une situation idéale où l'on se trouve exactement à la frontière entre bleu et cyan. On y percevrait donc le triplet RGB $(0, 127, 255)$. Il serait naturel de vouloir tourner à droite d'un angle moitié moins aigu que celui choisi dans le cas $CYAN = (0, 255, 255)$, et donc de positionner *turn* à -0.5. Symétriquement, pour $(127, 0, 255)$, on choisira $turn = 0.5$. Cela suggère la formule suivante :

$$turn = \frac{R - G}{B}.$$

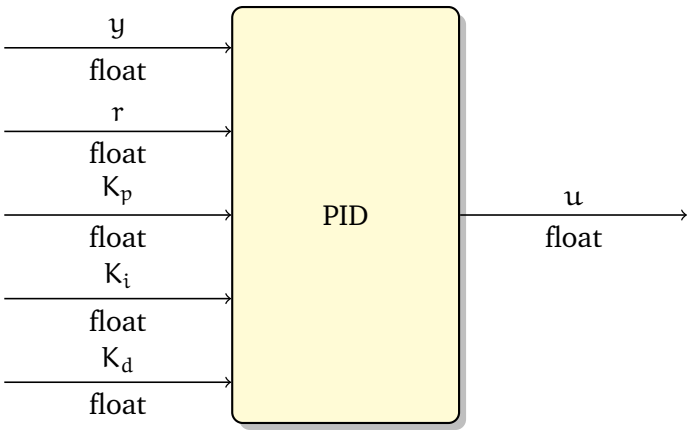
4. La formule ci-dessus suppose que R, G, B sont des nombres réels (ou rationnels). Cependant, les valeurs du champ *s_road* sont des entiers 32 bits (*int*). Il est recommandé de les **convertir en nombres à virgule flottante 32 bits (*float*)** via l'opérateur *Mathext.float* pour éviter des erreurs de calcul, par exemple lorsque R est nul.
5. On peut maintenant modifier *move_forward* de sorte à utiliser la sortie de *turn_decoder* pour influencer *rspeed*. Il s'agit donc de modifier la vitesse angulaire ω calculée précédemment pour lui appliquer l'influence de *turn*. Les équations finales gouvernant les vitesses des roues gauches et droites devraient ressembler à

$$left = K_v \cdot V + turn \cdot C$$
$$right = K_v \cdot V - turn \cdot C$$

   où la constante C est à choisir judicieusement.

3.4 Suivi de route évolué via un régulateur PID

1. La solution réalisée ci-dessus est très simple : elle calcule des trajectoires grossières qui tendent à osciller, sauf lorsque V est très faible. On peut l'améliorer via un peu d'automatique élementaire, par exemple en utilisant un régulateur proportionel, intégral, dérivé (PID) pour corriger *turn*. Un régulateur PID générique offrirait l'interface ci-dessous.



   L'entrée y est la mesure, l'entrée r est la consigne, c'est à dire la valeur vers laquelle on souhaite faire tendre la mesure. Les coefficients K_p, K_i et K_d contrôlent respectivement l'action proportionnelle, intégrale, et dérivée du régulateur. La sortie u est la commande.

Vous pouvez programmer un régulateur PID générique. On rappelle que la commande $y(t)$ est déterminée en fonction de l'erreur e, c'est à dire de la différence entre la mesure et la consigne. Plus précisément, on a $y(t) = P(t) + I(t) + D(t)$ où

$$P(t) = K_p \cdot e(t), \qquad I(t) = K_i \int e(t)dt, \qquad D(t) = K_d \cdot \dot{e}(t)$$

On approximera les opérations continues par des versions discrètes naïves.

2. Dans l'exemple qui nous occupe, à quoi correspondent les flots y, r et u ? Utilisez votre régulateur PID générique dans *move_forward* pour améliorer la trajectoire du véhicule. Les coefficients K_p, K_d et K_i seront déterminés par tâtonnement.

3.5 Suivi de l'itinéraire

1. Une fois un suivi de route minimal en place, on peut s'intéresser au suivi de l'itinéraire global. L'itinéraire vous est fourni par l'entrée *iti* de votre composant *controller*. Cette entrée transporte un flot constant de tableaux spécifiant une liste d'actions à effectuer à chaque étape et entre deux étapes. Il est recommandé de commencer par **lire la documentation pour comprendre les types *itielts* et surtout *itielt***.
2. Une fois la nature des différentes actions décrites par *itielt* comprise, il s'agit de comprendre comment les réaliser en combinant les opérateurs *move_forward* et *rotate* réalisés précédemment. Vous pouvez **réfléchir à l'architecture à adopter dans *controller*** : quand les différents opérateurs doivent-ils être activés ? Comment sait-on si une action de l'itinéraire a été pleinement réalisée, et qu'il faut **passer à l'action suivante** ? Quelles **structures de contrôle** employer ?
3. Pour démarrer, il peut être utile de relire les programmes Heptagon réalisés lors des séances de travaux pratiques antérieures. Vous pouvez notamment revisiter le fonctionnement des deux structures de contrôle employées jusqu'ici : les **automates** et les **blocs conditionnels (*if/then/else*)**.
4. L'action *Stop* est particulière : elle signale la fin de l'itinéraire. Sur quelle sortie de l'opérateur *controller* cette action peut-elle influer ?

3.6 Gestion des feux

1. Votre véhicule doit s'arrêter face à un feu de signalisation au rouge. La **présence** d'un feu est signalé par une **bande rouge** sur la route. La **couleur** du feu est détectable par le **capteur frontal** de votre véhicule, dont les observations sont accessibles via le champ *s_front* de *sens*.
2. Il existe plusieurs façons d'ajouter la gestion des feux rouges au code existant. On peut par exemple l'ajouter **à l'intérieur** de l'opérateur *move_forward*, ou bien **à l'extérieur** en l'intégrant à la structure de contrôle.

3.7 Difficultés

Si les sous-sections précédentes décrivent beaucoup des idées utiles à la réalisation du projet, assembler celles-ci et les tester sur les pistes fournies vous révèlera des difficultés spécifiques. Certaines d'entre elles sont discutées ci-dessous.

- Il est important de garder à l'esprit que les points d'intérêts (étapes et feux rouges) sont signalés par des bandes de couleur **épaisses de plus d'un pixel**. Quelles en sont les conséquences sur votre code, et en particulier sur le choix de passer à l'action suivante de l'itinéraire, ainsi que sur les rotations ?
- Quel est l'impact de ces bandes de couleur sur votre opérateur *move_forward* ? Quels problèmes peuvent se poser, et comment y remédier ?