UNIVERSITÉ DE PARIS
Université Paris Diderot

RAPPORT DU PROJET " ROBOT SUIVEUR DE LIGNE "

de 1^e année de Master en Informatique
parcours IMPAIRS
soutenu par

Djamel ALI
et
Hamza IDRISSOU

le 25 mai 2021

(Projet fait en 1^{er} et 2^{nd} semestre de M1)

Programmation d'un robot suiveur de ligne en utilisant la brique LEGO EV3

L'équipe enseignante

M. Hugo FÉRÉE
M. Giovanni BERNARDI
M. Charles FOUGERON

Année universitaire 2020-2021

Table des matières

Remerciements

Bonjour,

Avant tout développement et analyse de ce qui a été fait dans ce projet, il apparaît opportun de commencer ce rapport par des remerciements, à tous ceux qui nous ont beaucoup appris au cours de cette année d'étude.

Nous remercions particulièrement nos enseignants de ce cours : M Hugo FÉRÉE, M Giovanni BERNARDI et M Charles FOUGERON qui nous ont formés et accompagnés tout au long de cette année avec beaucoup de patience et de pédagogie.

Introduction

Qui sommes-nous ?

Nous sommes Djamel ALI et Hamza IDRISSOU, deux étudiants en M1 IMPAIRS au sein de l'Université de Paris (Université Paris Diderot).

Quel est l'objectif de ce projet ?

L'objectif de ce projet est de développer tout d'abord un robot suiveur de ligne, puis essayer d'améliorer ses comportements et performances afin de gagner la compétition des suiveurs de ligne qui se déroulera à la fin de l'année universitaire.

Que fait le robot ?

Le robot suiveur de ligne est un véhicule automatisé qui suit une ligne visuelle intégrée au sol, ayant une seule et même couleur sur toute la ligne (donc le robot lorsqu'il arrive à une intersection de deux lignes de couleurs différentes ne doit pas se tromper et suivre une ligne d'une autre couleur).

Quels sont les principaux scénarios d'usages ?

Ces robots ne sont pas seulement des robots qui servent lors de tournois, mais servent aussi dans l'industrie (assistance dans les processus de production automatisé, dans les services de livraison …) ou encore dans les transports en commun (bus autonomes).

Qu'est ce que ce projet nous a apportés ?

Plus largement, ce projet a été l'opportunité pour nous de découvrir (un petit peu) le monde des robots et des systèmes embarqués, et de comprendre comment sont (généralement) programmés ces derniers.

Quelles sont les principales difficultés rencontrées ?

- Des bugs très récurrents qui empêche le programme de s'exécuter (particulièrement une exception liée à l'initialisation des moteurs), sans rien modifier, ce même programme fonctionne parfois (pas d'exception) mais qui trop lent (i.e. Le robot sait qu'il doit tourner mais il exécute cette instruction très en retard); c'est ce comportement-là qui nous a fait perdre beaucoup de temps au 1er semestre. Après réinstallation du même OS (ev3dev : https://www.ev3dev.org/) le même problème persiste, donc on a installé l'OS leJOS EV3 à sa place, résultat : le même programme fonctionne très bien sur cet OS. c'est principalement à cause de ça que nous avons diminué la vitesse du robot lors de la pré soutenance, (conséquence: mauvaise note).

- [DIFFICULTÉS EXTERNES] Moi et binôme travaillons tous les 2 à temps partiel (environs 20h/semaine) en parallèle de nos études, ce qui nous empêche un petit peu parfois d'avancer plus rapidement.

Documentation

1. Si le programme n'est pas encore installé dans la brique EV3 (carte SD):

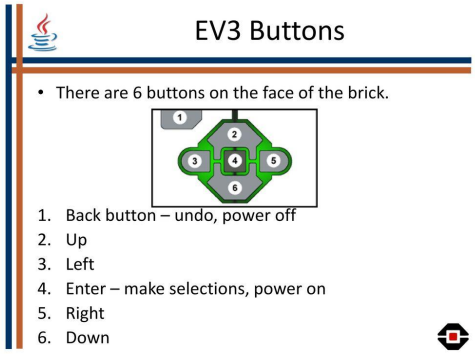
- Il faut suivre les étapes suggérées dans la documentation officielle http://lejos.org/ev3.php (ou le README de ce projet) pour comprendre les différentes manières de connecter une brique LEGO EV3 à une machine, comment transférer et exécuter le code sur la brique, ...etc.
- Vous pouvez maintenant utiliser le code disponible à cette adresse https://gaufre.informatique.univ-paris-diderot.fr/alid/ali-idrissou-plong-2020 pour l'envoyer vers la brique LEGO EV3.

2. Le programme est prêt (dans EV3) à être exécuté par notre robot:

- **Lancer le programme depuis Eclipse :**

  Si vous utilisez Eclipse et que vous avez installé le plugin *leJOS EV3* (c.f. https://sourceforge.net/p/lejos/wiki/Installing%20the%20Eclipse%20plugin/), vous pouvez lancer l'exécution du programme par la brique EV3 depuis Eclipse (c'est très pratique dans la phase de développement, particulièrement lors de la phase de test où il faut faire beaucoup de tests pour améliorer les performances du robot suiveur de ligne).

- **Lancer le programme depuis la brique EV3:**



En utilisant les boutons de la brique (c.f. l'image à côté) pour naviguer dans son arborescence de fichier, allez dans *'Run Default'* si vous n'avez qu'un seul programme par défaut,

ou sinon vous pouvez aller dans *'Programs'* puis sélectionner le fichier *'.jar'* que vous voulez exécuter, puis cliquez sur *'Execute program'*.

(pour notre cas '*MainClass.jar'* correspond au programme suiveur de ligne, mais vous pouvez en trouver d'autres comme *'HelloRobots.jar'* qui permet d'afficher un petit message de salutation sur l'écran de la brique).

Remarque:

Si vous voulez aussi tester (individuellement) les autres petits programmes *(HelloWorld.sayHello()* , *RunMotors.startTurning()*, *LearningColors.startLearning(int nbColors, int n Measures))* disponibles dans ce même projet (à l'adresse sus-cité), vous n'avez qu'à commenter et décommenter ce qu'il faut dans la classe *MainClass.java* et réexécuter ce code (qui est donc toujours *'MainClass.jar'*) dans la brique.

Détails d'implémentation

Nous avons adopté l'approche PID pour l'implémentation de ce programme, ce dernier se base sur deux parties (2 phases) principales qui sont:

1. Phase d'apprentissage des couleurs:

Le programme correspondant à l'apprentissage des couleurs est dans le fichier ***LearningColors.java*** (dans et la méthode principale est

```
public void startLearning(int nb_of_color_to_learn, int nb_of_measures_per_color)
```

Elle contient 2 boucles '`for`' imbriquées (une pour le nombre de couleurs à apprendre, et une autre pour le nombre de mesures par couleur. Les noms des variables sont assez significatifs pour tout comprendre).

Les principales étapes que suit cette méthode (à l'intérieur des 2 boucles '`for`' imbriquées) sont :

1. Elle attend que quelqu'un clique sur '*ENTRER*' pour récupérer un échantillon.
2. Elle vérifie que l'échantillon récupéré est valide (les valeurs RGB <=255), sinon msg d'erreur, et revenir à 1.
3. Si c'est la 1ère mesure d'une couleur à apprendre, elle la stock dans `acceptable_measures`, et initialise la variable `average_color` avec cette mesure.
4. Sinon (c'est la 2ème mesure ou plus), elle calcule la distance entre la couleur de cet échantillon (variable `sample`) et la couleur moyenne calculée jusqu'à présent (dans la variable `average_color`) **.**
5. Si cette distance est plus grande que `MAX_ALLOWED_DISTANCE`, elle recommence depuis la 1ère mesure pour cette couleur, et réinitialise la couleur moyenne `average_color` à 0,0,0 (pour R,G,B).
6. Sinon, elle ajoute cette mesure au tableau `acceptable_measures` et met à jour la couleur moyenne `average_color`.

   À la sortie de la boucle interne (i.e. elle a fait toutes les mesures d'une couleur) elle ajoute dans la liste des couleurs apprises (`listOfLearnedColors`) un nouvel objet de type `Color` instancié avec les valeurs R,G,B de la couleur moyenne `average_color` qui vient d'être calculée, elle réinitialise cette dernière avant de reboucler sur la boucle externe (pour apprendre une nouvelle couleur) ou sortir si elle a appris toutes les couleurs qu'elle devait devait apprendre.

2. Phase de suivi de la ligne:

Le programme correspondant se trouve dans le fichier **PIDLineFollower.java** (dans `/ali-idrissou-plong-2020/src/edu/robots/behaviours/`).

L'approche utilisée est **l'approche PID** (un des articles dont nous nous servons comme référence est à l'adresse http://www.inpharmix.com/jps/PID_Controller_For_Lego_Mindstorms_Robots.html).

Cette phase se compose de deux méthodes principales qui sont `public static void getReady()` et `public void followTheLine()` .

2.1. La méthode `void getReady()`:

Dans cette méthode on trouve:

A. La récupération de la variable **static** `listOfLearnedColors` qui a été remplie dans la phase précédente (car dans la méthode principale `main(string [] args)` qui est dans

MainClass.java, on fait appel à ces méthodes dans l'ordre, d'abord l'apprentissage puis le suivi de la ligne).

B. L'identification des 4 couleurs (ligne, arrière plan, frontière et stop (ou départ/arrivée)).
C. Le calcul de la distance maximale entre 2 couleurs (qui correspondrait donc à la distance entre la couleur de la frontière (qui est la couleur cible pour notre robot) et soit la couleur de fond (le capteur est complètement sur le fond) soit la couleur de la ligne (le capteur est complètement sur ligne)), on se sert de cette distance pour calculer la pente moyenne `avg_slope` qui nous servira à son tour pour calculer les paramètres kp, ki et kd du contrôleur PID.
D. Le calcul et l'initialisation de la vitesse cible `target_power` et des 3 paramètres de notre contrôleur PID.
E. L'affichage d'un message qui demande à l'utilisateur de cliquer sur ENTRER pour qu'il commence à suivre la ligne (pour notre cas, il faut placer le le robot sur la frontière droite de la ligne (de la 1ère couleur apprise), car c'est un suiveur de ligne à droite).

2.2. La méthode `void followTheLine()`:

Cette méthode commence à être exécutée dès qu'on clique sur le bouton ENTRER qui mettra fin à l'attente active citée dans le point **2.1.E**; au sein de cette méthode on trouve une boucle principale `do{...}while (Button.ESCAPE.isUp());` qui contient la suite d'instructions suivante:

A. Capturer un échantillon (`sample`)
B. Vérifier si `sample` < `MAXIMUM_TOLERATED_DISTANCE`
    a. Si c'est le cas, trouver vers quelle couleur cet échantillon est le plus proche (ligne, arrière-plan, frontière ou stop), initialise `the_closest_color`.
    b. Sinon, sortir de la boucle **do...while** principale, produire 2 bips sonores et arrêter les 2 moteurs.
C. Si **the_closest_color** est **stopColor**, alors quitter la boucle principale, produire les 2 bips sonores et arrêter les 2 moteurs.
D. Si **the_closest_color** est **lineColor**, alors initialise **I_AM_IN_LINE à 1** (cette variable pour déterminer le sens du virage: 1 = à droite; -1 = à gauche; 0 = tout droit).
E. Si **the_closest_color** est **backgroundColor**, alors initialise **I_AM_IN_LINE à -1**.
F. Sinon (i.e **the_closest_color** est **medianColor**, alors initialise **I_AM_IN_LINE à 0**.
G. Calculer la valeur de `sample_median_distance` qui est la distance entre notre échantillon et notre couleur cible (celle de la frontière).
H. Utiliser `sample_median_distance` afin de calculer la valeur de l'erreur `(error = sample_median_distance * I_AM_IN_LINE;)`.
I. Calculer/mettre à jour la valeur de l'intégral (qui est la mémoire de ce contrôleur) `integral = (int) (0.9f * integral + error);`
    a. **Remarque** concernant le rôle du paramètre **0.9f** :

    Cela réduit la valeur de l'intégrale précédente de **10%** à chaque fois que la boucle est parcourue. Si vous considérez le terme intégral comme la "**mémoire**" du contrôleur, alors **cet amortissement le force à oublier les choses qui se sont produites il y a "longtemps"**.

J. Mettre à jour la valeur de la dérivé (qui permettra à notre contrôleur d'anticiper les valeurs des erreurs à venir) `derivative = error - last_error;`
K. Calculer la valeur du virage `turn = Kp * error + Ki * integral + Kd * derivative;`

L. Calculer la vitesse des 2 moteurs

```
leftMotorSpeed = target_power + (float) turn;
rightMotorSpeed = target_power - (float) turn;
```

M. Mettre à jour l'ancienne erreur : `last_error = error;`