Algorithmique, robotique et systèmes temps réel (RO52)

TP N°3 – ROBOT SUIVEUR

Sébastien DOUTRELUINGNE, Samuel BERNARD

05/05/2023

utbm
université de technologie
Belfort-Montbéliard

Introduction

Dans le cadre de notre UV RO52, après avoir exploré et intégré différents algorithmes permettant au robots EV3 de LEGO de suivre un tracé au sol, nous devions réaliser un programme permettant au robot de suivre un autre robot EV3. Ce travail pratique nous a notamment permis de mettre en application les notions abordées lors de cours magistraux.

Pour réaliser ce TP, nous disposons d'un robot « Leader » dont l'objectif est de guider les autres robots. En effet, celui-ci avance pendant un temps T, puis s'arrête pendant ce même temps de manière cyclique. Nous disposons également d'un robot « Suiveur » dont l'objectif est de suivre et d'anticiper l'arrêt du robot leader à l'aide d'un capteur à ultrasons permettant de mesurer la distance entre le suiveur et le leader.

Le robot doit s'arrêter lentement en fonction de la distance avec le suiveur afin d'avoir un freinage réaliste. Pour cela, nous appliquerons les quatre politiques suivantes :

- « Tout ou rien »
- « A un point »
- « A deux points »
- « Communication WIFI »

Objectif

L'objectif principal est ainsi de faire avancer le robot suiveur tout en faisant varier sa vitesse en fonction de sa distance avec le robot leader afin d'obtenir un freinage amorti.

Afin de réaliser cette fonctionnalité étape par étape, nous avons tout d'abord développé le programme du robot leader, puis nous avons implémenté les quatre modes de fonctionnement présentés précédemment.

Par ailleurs, l'ensemble des programmes ont été réalisés en Python 3 avec les bibliothèques de développement adaptées pour les robots EV3.

Robot Leader

Tout d'abord, il est nécessaire de développer le programme du robot leader, qui nous permettra par la suite de réaliser nos tests sur les différentes politiques du robot suiveur. Le leader doit avancer à 40% de sa vitesse maximale.

Ainsi, nous avons défini une classe *Leader* qui comprend différents attributs :

- *time* : float défini par l'utilisateur (dans le constructeur) correspondant au temps pendant lequel le leader va avancer, puis ce même temps durant lequel il sera à l'arrêt
- *timestamp* : float prenant la valeur du temps actuel à chaque transition du leader entre ses états arrêté et en mouvement
- *elapsed_time* : float correspondant au temps écoulé depuis la dernière transition
- *flagMove* : bool prenant la valeur True si le leader peut avancer, False sinon

Cette classe possède une fonction move retournant la valeur de *flagMove* qui indique si le leader est en état de mouvement ou à l'arrêt :

```
def move(self):
        self.elapsed_time = time.time() - self.timestamp
        if self.elapsed_time>self.time:
            self.timestamp = time.time()
            self.flagMove = not self.flagMove
        return self.flagMove
```

Cette fonction va tout d'abord mettre à jour la valeur de *elapsed_time* (différence entre le temps actuel et la valeur de *timestamp)*. Ensuite, si cette valeur est supérieure à la valeur *time* définit par l'utilisateur, alors cela signifie que le leader est dans son état actuel depuis le temps *time*, et il doit donc nécessairement basculer dans son autre état. Ainsi, dans ce cas, la valeur *timestamp* est réinitialisée avec la valeur du temps actuel, puis *flagMove* prend sa valeur contraire. Si le robot est à l'arrêt, alors il sera en mouvement, et inversement.

```
if leader.move():
         robot.drive(0.4 * speed, angleN)
    else:
         robot.stop()
```

Dans le programme *main*, on appelle la fonction *drive* du robot lui permettant de se mettre en mouvement si la valeur retournée par la fonction move est *True*, sinon le robot s'arrête avec la fonction *stop*.

Robot Suiveur

Politique « Tout ou rien »

La première politique développée a été de faire avancer le robot à 50% de la vitesse maximale tant qu'il ne détecte pas d'obstacle et de l'arrêter lorsqu'il en détecte un. L'obstacle (qui peut être le robot leader qui s'arrête en chemin) est détecté par un capteur à ultrasons.

Pour cela, nous avons créé une fonction *move* dans la classe *Suiveur* qui permet d'informer si une présence est détectée à une distance de 15 cm. Si le robot peut bouger (c'est-à-dire qu'aucune présence n'est détectée à une distance de 15 cm), la fonction renvoie *True*, sinon *False*.

```
def move(self):
        if self.distanceController.getPresence(150):
            return False
        else:
            return True
```

Pour cela, nous avons utilisé la fonction *getPresence* de la classe *DistanceSensorControl* qui nous permet de savoir s'il y a une présence à une distance passée en paramètre de la fonction.

Cette fonction est appelée dans le programme *main* du robot suiveur de la même manière que la fonction homonyme du robot leader.

Politique « A un point »

La deuxième étape a consisté à faire freiner le robot en fonction de la distance entre le suiveur et le leadeur afin d'éviter un arrêt brusque et de privilégier un arrêt progressif du robot suiveur pour ainsi maintenir distance constante pendant sa course. Pour cela, sa vitesse doit varier en fonction de sa distance avec le leader détectée par le capteur à ultrasons.

Nous avons ainsi calculé un coefficient en temps réel correspondant au pourcentage de la vitesse maximale à appliquer au robot :

$$\mathrm{P_V}(t+Ts) = max(min(50, a \times (d(t) - D)),0)$$

$\mathbf{Ts}$: Temps d'un cycle
$\mathbf{d(t)}$: distance suiveur-leadeur du cycle précédent renvoyé par le capteur
$\mathbf{D}$: distance suiveur-leadeur à laquelle le suiveur ne peut pas dépasser.
$\mathbf{a}$: coefficient appliqué sur la différence entre la distance leadeur-suiveur et la distance maximum acceptable

$\mathbf{P_V(t+Ts)}$: Pourcentage de vitesse à appliquer sur la vitesse maximale du robot à l'instant $\boldsymbol{t+Ts}$. Ce coefficient est donc borné grâce à la formule ci-dessus. La vitesse du suiveur sera donc comprise entre 0% et 50% de sa vitesse maximale :

$$0 \leq P_V.(\boldsymbol{t+Ts}) \leq 50$$

Ainsi, nous avons créé une fonction nous renvoyant un pourcentage de réduction de la vitesse maximale à appliquer sur le robot.

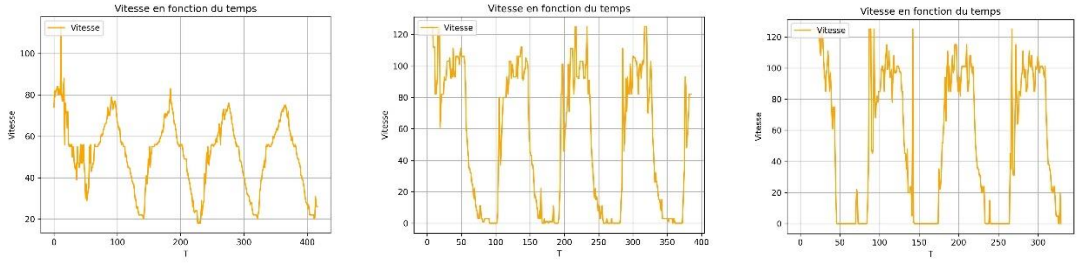
```
def move1Point(self):
        self.speedCoeff = max(min(50, self.a*(self.previousDistance-self.d)), 0) / 100
        self.previousDistance = self.distanceController.getDistance()
        return self.speedCoeff
```

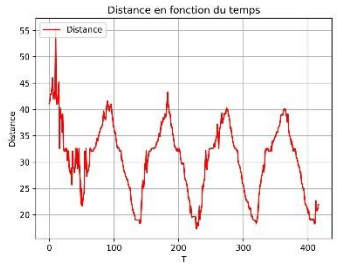
Le coefficient borné entre 0 et 50 % calcule la différence entre la distance détectée précédemment par le capteur *previousDistance* et la distance acceptable *d*. Un coefficient *a* est appliqué sur cette différence afin que le robot s'adapte plus ou moins rapidement à la distance détectée. Ensuite, *previousDistance* est mise à jour avec la nouvelle distance détectée et la valeur du coefficient de vitesse est retournée.

Tests réalisés

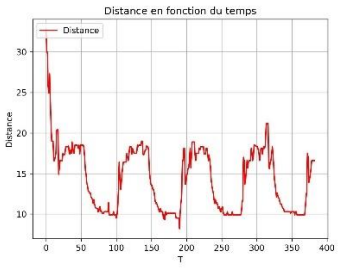
Nous avons ensuite réalisé différents tests afin d'identifier les impacts de variations des valeurs du coefficient a et de la distance D sur l'évolution de la vitesse v et de la distance inter-véhiculaire d_{iv} au cours du temps.

Afin de comparer les différents comportements du suiveur, nous avons créé des courbes représentant l'évolution de la vitesse v et de la distance inter-véhiculaire d_{iv} au cours du temps pour différentes valeurs de paramètres a et D. Afin de les créer, nous avons utilisé le programme informatique *evcurves.py* réalisé par Corenthin EHLINGER.

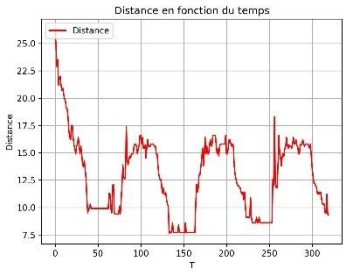




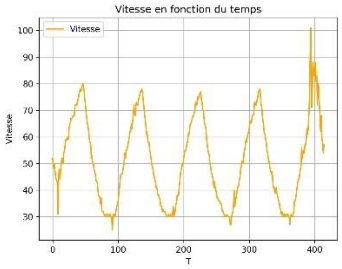
a = 0.1, distance = 100 cm

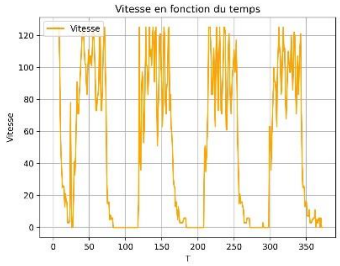


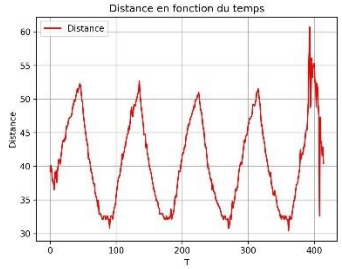
a = 0.5, distance = 100 cm



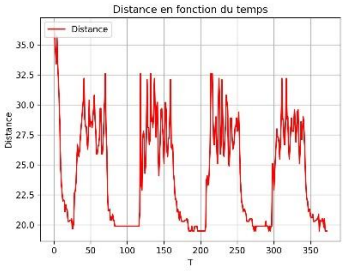
a = 0.7, distance = 100 cm







a = 0.1, distance = 200 cm



a = 0.5, distance = 200 cm

L'objectif était en premier lieu de trouver une valeur cohérente pour le **coefficient a** permettant au robot suiveur de suivre le leader avec efficacité dans l'objectif d'avoir une distance intra-véhiculaire qui soit constante au cours du temps. Nous avons ainsi testé trois valeurs de a : 0.1, 0.5 et 0.7.

Sur les graphiques présents ci-dessus, nous pouvons tout d'abord remarquer que le **coefficient a** à une influence sur la vitesse à laquelle le robot va réagir avec efficacité à l'arrêt du robot.

Nous pouvons remarquer que sur la première courbe, la distance d_{iv} varie entre 20 et 40 cm pour une distance D de 10cm ce qui montre qu'il n'a pas réussi à être à la distance donnée. De plus, cette courbe suit la courbe de la vitesse ce qui n'est pas bon étant donné que le robot doit répondre avec efficacité et rapidité à l'écart de distance intra-véhiculaire. La courbe de d_{iv} doit être la plus lisse possible.

En augmentant la valeur de a pour une même distance, nous avons une nette amélioration de la courbe de d_{iv} jusqu'à obtenir une variation de seulement 5cm avec a = 0.7. Le suiveur a donc correctement réussit à rattraper le leader lorsque celui-ci à recommencer à avancer en adaptant sa vitesse rapidement : Nous pouvons le remarquer sur la courbe de vitesse qui a fortement et rapidement augmenter lorsque le leadeur s'est mis à avancer contrairement au premier graphique (avec a = 0.1) avec une vitesse qui

augmente lentement. Toutefois, avec un a de 0.7, la vitesse devient trop élevée et le robot met trop de temps à s'arrêter. Ainsi, celui-ci s'arrête à une distance de 7.5cm du robot alors que la distance D est de 10cm.

Nous avons aussi essayé de changer la valeur de D (pour la fixer à 200m) avec les même valeurs de a et nous pouvons remarquer que plus la distance D est élevée, plus le temps d'adaptation du robot est élevé et plus le coefficient a doit être élevé. Et nous avons le même constat avec les courbes précédentes, il faut que *a* soit suffisamment élevé pour que d_{iv} soit rapidement à la distance D voulue.

Il ne faut pas tenir compte des pics qu'il peut y avoir sur le graphique de la distance intra-véhiculaire captée par le capteur à ultrasons car ils sont dû à l'imprécision de celui-ci.

Politique « A deux points »

La troisième étape a consisté de permettre au robot de freiner et d'accélérer en fonction de la distance entre le suiveur et le leadeur afin de la maintenir constante pendant sa course. Pour cela, sa vitesse doit également varier en fonction de sa distance avec le leader détectée par le capteur à ultrasons.

Le pourcentage de la vitesse maximale à appliquer au robot est donc calculer de la façon suivante.

Tout d'abord, on calcule le pourcentage de vitesse pour le freinage pour un point Df :

$$\mathrm{Pf_V}(t+Ts) = max(min(50, af \times (d(t) - Df)),0)$$

$\mathbf{D_f}$: distance de freinage que le suiveur ne peut pas dépasser.
$\mathbf{a_f}$: coefficient de freinage appliqué sur la différence entre la distance leadeur-suiveur et la distance minimum acceptable

On calcule également le pourcentage de vitesse pour l'accélération pour un point Da :

$$\mathrm{Pa_V}(t+Ts) = max(min(50, aa \times (d(t) - Da)),0, Pv(t))$$

$\mathbf{D_a}$: distance d'accélération que le suiveur ne peut pas dépasser.
$\mathbf{a_a}$: coefficient de freinage appliqué sur la différence entre la distance leadeur-suiveur et la distance maximum acceptable

On en déduit enfin le pourcentage de vitesse appliqué au robot :

$$\mathrm{P_V}(t+Ts) = min(\mathrm{Pfv(t + Ts)}, \mathrm{Pav(t + Ts)})$$

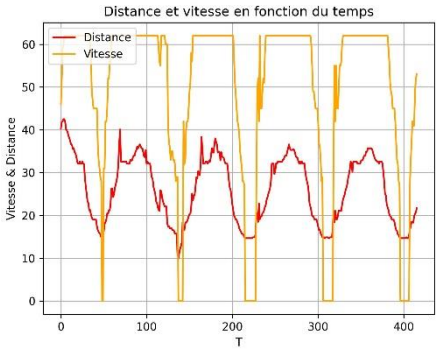
Ainsi, nous avons créé une fonction nous renvoyant un pourcentage de réduction de la vitesse maximale à appliquer sur le robot.

```
def move2Point(self):
        self.speedSlowDown = max(min(50, self.af*(self.previousDistance-self.df)), 0) / 100
        self.speedAccel = max(min(50, self.ac*(self.previousDistance-self.dc)), 0,
self.speedCoeff) / 100
        self.speedCoeff = min(self.speedAccel, self.speedSlowDown)

        self.previousDistance = self.distanceController.getDistance()

        return self.speedCoeff
```

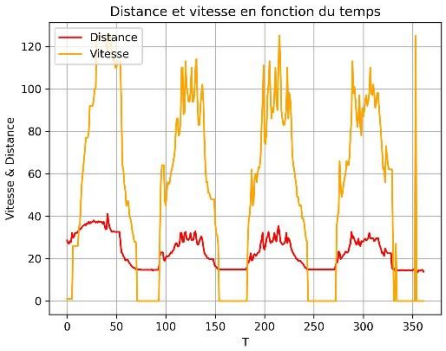
Cette fonction reprend les formules précédemment présentées en divisant les valeurs calculées par 100 de sorte à obtenir un coefficient à multiplier par la vitesse maximale du robot.

Tests réalisés

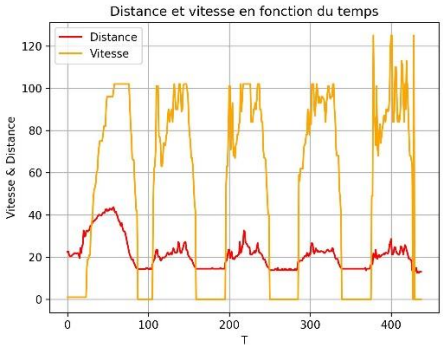
Nous avons ensuite réalisé différents tests afin d'identifier les impacts de variations des valeurs des coefficients et des distances d'accélération et de freinage sur l'évolution de la vitesse v et de la distance inter-véhiculaire d_{iv} au cours du temps. Nous n'avons pas testé différentes valeurs de temps de cycle Ts étant donné que la vitesse du robot car il est plus intéressant d'avoir un temps de cycle court de sorte à traiter les données en temps réel borné à 50%.



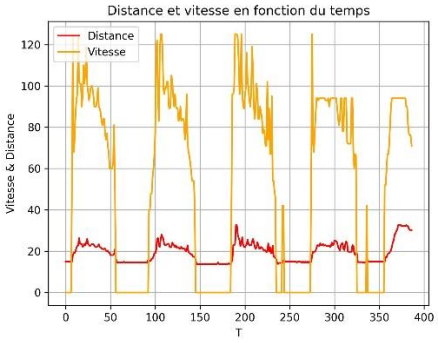
af = 0.2, Df = 100, aa = 0.2, Da = 300



af = 0.2, Df = 100, aa = 0.5, Da = 300



af = 0.3, Df = 100, aa = 0.3, Da = 300



af = 0.3, Df = 100, aa = 0.5, Da = 300

Tout d'abord, nous avons cherché à déterminer l'impact sur la vitesse et la distance inter-véhiculaires des valeurs de coefficients d'accélération a_a et de freinage a_f. Nous avons fixés des valeurs minimales de ces coefficients à 0.2.

Ainsi, pour ces premières valeurs de coefficients, nous constatons que la vitesse du robot se stabilise rapidement sans pour autant atteindre sa vitesse maximale autorisée possible. Également, on peut constater que le robot que la distance avec le leader augmente et diminue lentement en raison des faibles valeurs des coefficient d'accélération et de freinage.

En effet, on constate lorsqu'on augmente le coefficient d'accélération que la vitesse augmente d'avantage et plus rapidement lorsque le suiveur démarre et que le celui-ci perd moins de distance sur le robot leader.

Lorsque la valeur du coefficient de freinage est augmentée, le robot suiveur ralentira plus brusquement à l'approche du leader, ce qui se traduit en effet par une baisse abrute de la vitesse et de la distance inter-véhicules.

Ainsi, les coefficients de freinage et d'accélération permettent respectivement de caractériser le comportement du suiveur lorsque celui-ci va s'éloigner du leader et s'en rapprocher. Une variation du premier entrainera une accélération plus ou moins importante lorsque le suiveur s'éloignera du leader au démarrage de celui-ci. Une variation du second entrainera un freinage plus ou moins important lorsque le robot se rapprochera du leader.

Sur les courbes, un coefficient de freinage élevé entrainera un freinage sec du suiveur à la distance de freinage et donc une diminution rapide de la distance inter-robot et de la vitesse. Un coefficient d'accélération entrainera un une accélération immédiate du suiveur à la distance d'accélération et donc une augmentation rapide de la distance inter-robot et de la vitesse.

Les distances de freinage et d'accélération permettent ainsi respectivement de déterminer à quelle distance du leader le suiveur devra s'arrêter, et la distance du leader à laquelle le suiveur devra accélérer. Plus le suiveur se rapprochera de la distance de freinage, plus celui-ci ralentira. Plus il se rapprochera de la distance d'accélération, plus il accélérera.

Il est ainsi pertinent par rapport aux résultats obtenus de déterminer ces valeurs de coefficients qui permettent un comportement optimal du suiveur :

- Df = 10 cm
- Da = 30 cm
- af = 0.3
- aa = 0.5

Ainsi, comme on peut l'observer sur la dernière courbe, le suiveur accélérera directement dès que la distance avec le leader approchera les 30 cm lorsque ce dernier démarrera. Ensuite, à l'approche du leader, lorsque la distance approchera les 10 cm, alors le suiveur ralentira progressivement.

Communication sans-fil

La dernière étape consiste à réaliser une communication sans fil entre deux robots afin qu'ils avancent à la même vitesse. Pour réaliser cette fonctionnalité, Le leader qui jouera le rôle du serveur devra envoyer sa vitesse au suiveur qui sera le client. Les 2 robots doivent être sur la même borne wifi et doivent être connecté en Bluetooth.

Tout d'abord, c'est le serveur qui doit attendre une connexion avant que le suiveur se connecte ; nous avons donc créé une fonction d'attente *WaitForConnecion*() pour que le client puisse se connecter.

```
    def WaitForConnection(self):
        # The server must be started before the client!
        print('waiting for connection...')
        self.server.wait_for_connection()
        print('connected!')
        return True
```

Nous avons ensuite initialisé le serveur dans le programme du leader pour initialiser une connexion Bluetooth avec le client :

```
server = BluetoothMailboxServer()

def sendData(self, message):
        self.mbox.send(message)
```

Une fois que le leader attend une connexion, nous pouvons lancer le client avec qu'il se connecte au serveur. Pour cela, nous avons créé la fonction *connect()* dans le programme du suiveur :

```
def connect(self):
    print('establishing connection...')
    self.client.connect(self.SERVER)
    print('connected!')
    return True
```

Nous avons ensuite initialisé le client et le serveur dans le programme du suiveur avec d'établir une connexion Bluetooth avec un serveur.

```
    SERVER = 'ev3dev-2'
    client = BluetoothMailboxClient()
```

Dans le programme principal, il faut qu'il y ait un robot joue le rôle du leader (serveur) et un autre robot qui joue le rôle du suiveur (client). Une fois que le leadeur est en attente d'une connexion, le suiveur peut directement se connecter sur celui-ci.
Lorsque la connexion a été effectuée avec succès, le programme suivant s'exécute :

```
     if isClientConnected:
         speed = suiveur.receiveData()
```

```
        if speed == 0:
            speed = 250*suiveur.move()
        robot.drive(speed, angleN)
```

Avec la fonction *receiveData()* qui permet de récupérer les données du serveur :

```
    def receiveData(self):
        return self.mbox.read()
```

Dans ce programme, le client reçoit les informations de vitesse du serveur afin que les robots soit à la même vitesse. En l'absence de message reçu, la politique « tout ou rien » sera appliquée (fonction *move()).*

Nous avons effectué une multitude de tests afin d'établir la connexion Bluetooth entre les 2 robots, cependant après 3 heures d'essai, nous avons décidé d'arrêter.

Conclusion

Pour conclure, nous avons eu l'opportunité durant ce TP d'observer l'effectivité des différentes politiques de suivi d'un robot et d'adaptation de la vitesse en fonction de la distance avec le robot leader.

Nous avons notamment pu constater la facilité d'implémentation et d'exécution de la politique « A un point » qui permet un freinage et une accélération identique et efficace du robot suiveur, que l'on peut facilement identifier sur les courbes tracées.

Enfin, nous avons pu améliorer cet algorithme avec la politique « A deux points » et la possibilité de considérer deux comportements et deux distances différents pour le freinage et l'accélération du suiveur.