Conception d'un robot mobile autonome

Compte rendu

Jocelyn Deleuil & Guillaume Seimandi
Groupe ESE B

Sommaire

1 Introduction **1**

2 Prise en main du robot **2**

3 Stratégies **3**
3.1 Trajectoire vectorielle 3
3.1.1 Hypothèses et tests 3
3.1.2 Solution 5
3.2 Contre-réaction linéaire 7
3.2.1 Principe de fonctionnement 7
3.2.2 Choix des contres-réactions 8
3.2.3 Application 9

4 Conclusion et limites de la stratégie choisie **11**

1 Introduction

L'essor des robots autonomes marque une révolution technologique majeure, influençant de manière significative une multitude de secteurs. De la logistique à l'assistance à la personne, en passant par la surveillance, l'exploration, l'agriculture, le transport urbain et même le football, ces robots, véritables bijoux technologiques, apportent des solutions avant-gardistes qui promettent de révolutionner notre quotidien et notre manière de travailler.

Ce rapport détaille la conception d'un robot mobile autonome, certes à un niveau scolaire, mais permettant d'acquérir les notions fondamentales des systèmes embarqués. Composé de deux roues et de cinq capteurs infrarouges, notre robot n'a rien à envier des machines plus avancées, car il se déplace tout aussi bien dans son environnement en évitant les obstacles avec précision.

Ainsi, nous nous pencherons essentiellement sur la stratégie mise en place pour atteindre cet objectif, en expliquant nos réflexions, nos hypothèses et la manière dont nous sommes arrivés à nos fins : un robot qui a fait un sans faute lors de l'évaluation finale.

2 Prise en main du robot

La première partie de notre projet a été réalisée pendant les heures de cours où notre objectif était de suivre le sujet pas à pas pour progresser dans la programmation de notre robot tout en découvrant les différents composants.

Toutefois, il ne nous semble pas pertinent de détailler l'intégralité de cette phase dans ce rapport. Cette étape initiale visait principalement à nous familiariser avec l'environnement de travail et les diverses parties du robot, en effet nous avons été habillement orienté grâce au sujet, et les programmes nécessaires étaient déjà fournis. Le but de cette première partie n'était pas tant d'obtenir un programme complet, mais plutôt de comprendre le rôle et l'impact de chaque dispositif sur le fonctionnement global du robot.

Timers

Les timers du microcontrôleur dsPIC33EP512GM306 sont essentiels à la gestion du temps et à la coordination des tâches de notre robot. Agissant comme des compteurs précis, ils mesurent des intervalles de temps et déclenchent des événements programmés. Cette précision est cruciale pour synchroniser les fonctions du robot, telles que le mouvement des moteurs (via des signaux PWM) et la lecture des capteurs.

PWM et hacheur

La PWM est essentielle pour le contrôle des dispositifs électroniques dans notre robot. Elle permet de réguler l'intensité du courant dans les moteurs, les LED, et d'autres composants, en ajustant la durée des impulsions électriques. Cette technique assure un contrôle précis de la puissance sans consommation excessive d'énergie. Dans le cas des moteurs, la PWM est combinée à un hacheur de puissance pour ajuster la vitesse, en modulant la durée des signaux électriques.

Capteurs et ADC

Les télémètres infrarouges, couplés aux convertisseurs analogique-numérique (ADC), transforment notre robot en un système autonome capable de détecter les obstacles et de s'adapter dynamiquement à son environnement. Ces capteurs mesurent la distance par réflexion infrarouge, et ces signaux sont ensuite convertis en données numériques par l'ADC, afin d'être traités par notre algorithme de navigation dans le microcontrôleur.

Ce qui est véritablement intéressant, c'est la façon dont nous avons intégré toutes ces nouvelles connaissances dans notre stratégie finale pour optimiser le fonctionnement du robot.

3 Stratégies

Alors qu'une machine à états peut devenir extrêmement complexe avec l'augmentation du nombre d'états et de transitions, un système basé sur des calculs mathématiques reste relativement simple, même lorsqu'il traite des situations élaborées. La complexité ne croît pas de manière exponentielle avec le nombre de scénarios possibles, car le système n'est pas construit sur une série de règles et de conditions prédéfinies, mais sur des principes mathématiques qui peuvent être appliqués de manière uniforme dans diverses circonstances. C'est pour ces raisons que nous avons choisi de mettre de côté la solution par table de vérité, pour ce tourner vers des stratégies plus calculatoires.

3.1 Trajectoire vectorielle

3.1.1 Hypothèses et tests

Dans cette perspective, nous avons entrepris des recherches visant à développer une solution qui utilise des vecteurs, le produit scalaire et la trigonométrie. Nous avons considéré que chaque capteur envoie un signal infrarouge devant lui, pouvant être représenté par un vecteur comprenant une direction, un sens et une norme correspondant à la distance mesurée. Nous avons alors envisagé d'utiliser la somme vectorielle des signaux provenant des 5 capteurs. En effectuant cette somme, le vecteur ayant la plus grande norme aurait une influence plus importante sur le vecteur résultant par rapport à un vecteur plus court. Ainsi, en suivant le vecteur résultant, il serait possible de s'orienter en permanence vers la zone ayant le plus d'espace disponible. C'est à partir de cette hypothèse que nous avons effectuer des différentes simulations sur Matlab afin de prendre conscience visuellement de l'idée que l'on avançait.

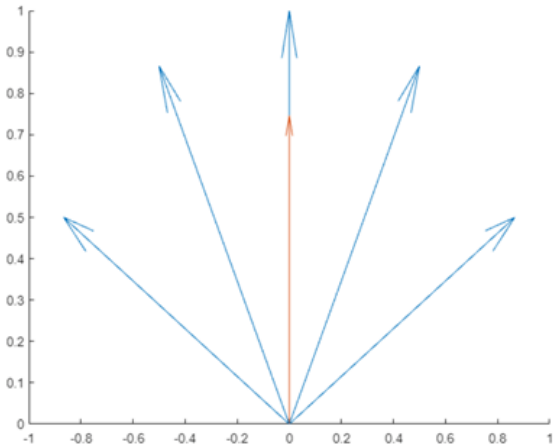

FIGURE 3.1.1 – Vecteurs de chaque capteur sans aucun obstacle

Jusque là, pas de problème, on note que le vecteur résultant, en orange, se dirige vers le centre, ce qui est normal, car les vecteurs ont tous la même norme et se compensent.

Cependant, lorsqu'on simule la présence d'obstacle, le modèle ne fonctionne plus et révèle une défaillance, en effet, les deux capteurs centraux sont obstrués, mais les vecteurs latéraux ne détectent rien, donc les vecteurs se compensent également, le vecteur résultant pointe toujours vers le centre, là où les obstacles se situent en réalité.

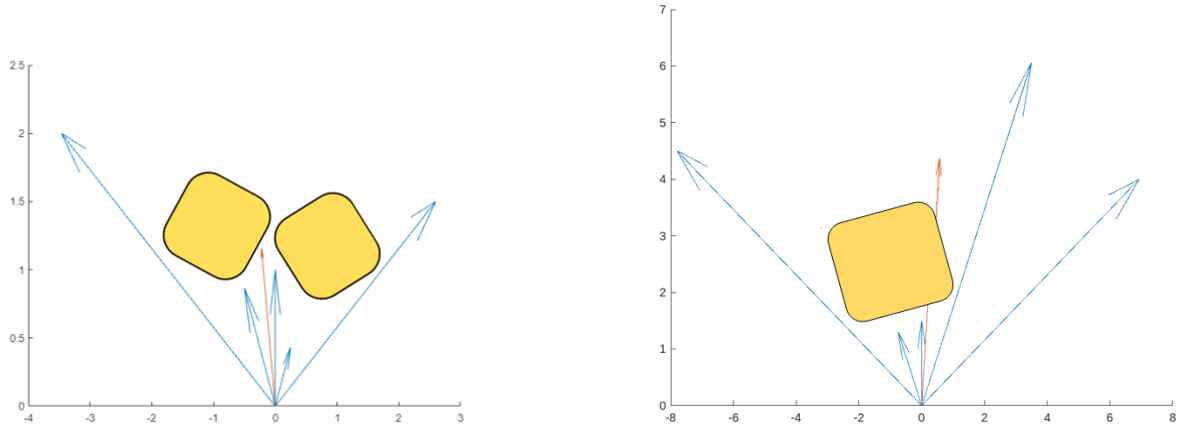

FIGURE 3.1.2 – Simulations en présence d'obstacles

Il fallait donc trouver un moyen de pondérer les poids des différents capteurs de manière à ce que les capteurs extrêmes aient une plus grande influence que les capteurs centraux.

Nous avons donc mis en place une première version où plus on s'écarte vers les extrémités, plus un coefficient était appliqué afin de sur-compenser les capteurs latéraux. Cette solution fonctionne et nous a permis de se sortir de différentes situations mais elle est encore une fois limitée. Sur la figure suivante, on peut voir que le vecteur résultant indique la bonne direction pour éviter l'obstacle, mais le robot frôle les obstacles et se coince dans le cas de parfaite distance égale où les capteurs se compensent mutuellement.

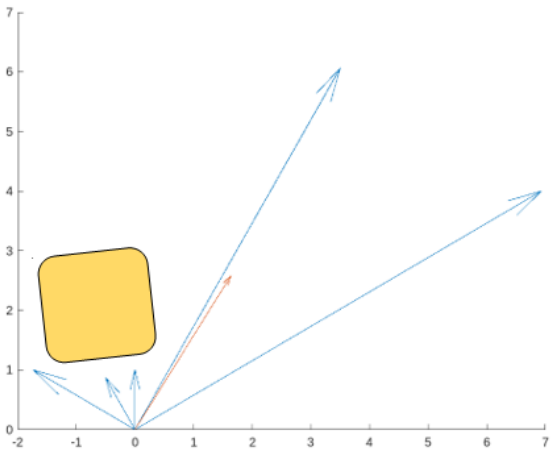

FIGURE 3.1.3 – Pondération des poids des capteurs

Nous avons alors tenté de développer un simulateur en Python, permettant une modification facile des divers poids attribués aux capteurs pour déterminer la configuration optimale. Nous sommes parvenus à recréer l'environnement et le mouvement du robot grâce à la bibliothèque Pygame.

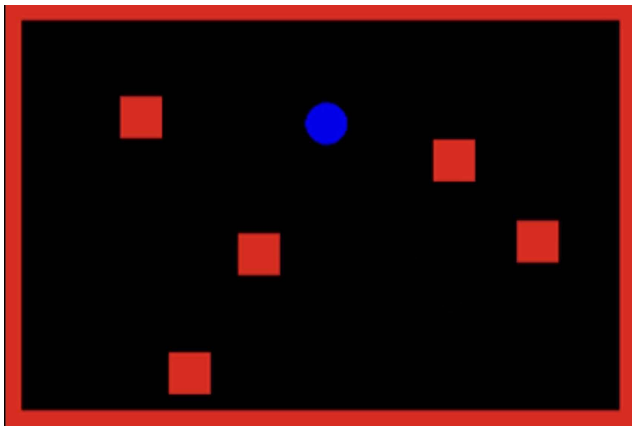

FIGURE 3.1.4 – Simulateur en Python

Cependant, nous n'avons pas plus approfondi, ayant rapidement compris que notre approche basée sur les vecteurs n'était pas la plus efficiente et qu'il n'était pas judicieux de s'y attarder excessivement. Mais ce n'est pas parce que nous avons mis ce simulateur de coté, que nous allons pas revenir dessus plus tard...

3.1.2 Solution

Notre dernière approche a été beaucoup plus performante. Afin de déterminer le vecteur direction le plus fiable, il a fallu ajouter une série de calculs et de tests pour s'orienter dans la bonne direction. C'est ainsi que nous avons implanté non pas un vecteur résultant, mais trois. Le premier est la somme des deux capteurs les plus à gauche, le second est la somme des trois capteurs les plus au centre, et le troisième est la somme des deux capteurs de droite. Ainsi, nous obtenions trois vecteurs directionnels intermédiaires, qui pourront ensuite être comparés.

L'ultime vecteur de direction est donc obtenu en choisissant quelle est le plus long parmi les trois. Cette solution a résolu tous nos problèmes, l'intérêt de cette stratégie réside dans les trajectoires lissées et les mouvements fluides. Elle permet de se diriger constamment vers l'espace ayant le moins d'obstacles, plutôt que d'éviter activement les obstacles.

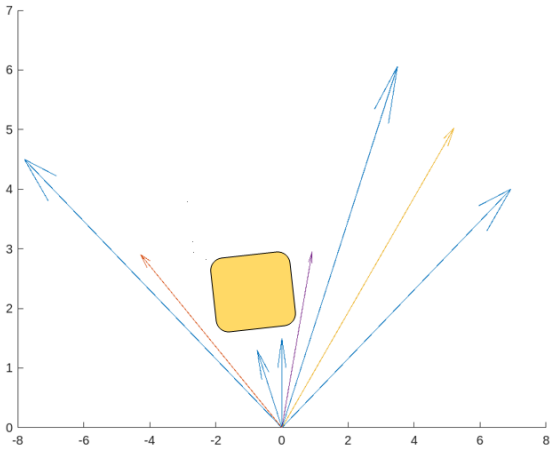

FIGURE 3.1.5 – Vecteurs résultants intermédiaires

Dans le cas ci-dessus, le vecteur final est le jaune, qui indique la zone la plus espacée.

```
// Initialisation des vecteurs uVect et vVect pour stocker les composantes des vecteurs de d
    éplacement
float uVect[3] = {
    robotState.distanceTelemetreMelanchon * 0.5f + robotState.distanceTelemetreGauche *
    0.866f,
    robotState.distanceTelemetreGauche * 0.866f + robotState.distanceTelemetreCentre +
    robotState.distanceTelemetreDroit * 0.866f,
    robotState.distanceTelemetreDroit * 0.866f + robotState.distanceTelemetreLePen * 0.5f
};

float vVect[3] = {
    robotState.distanceTelemetreMelanchon * -0.866f + robotState.distanceTelemetreGauche
    *-0.5f,
    robotState.distanceTelemetreGauche * -0.5f + robotState.distanceTelemetreDroit * 0.5f,
    robotState.distanceTelemetreDroit * 0.5f + robotState.distanceTelemetreLePen * 0.866f
};

// Calcul de la norme pour chaque vecteur
float normVect[3] = {
    sqrt(uVect[0] * uVect[0] + vVect[0] * vVect[0]),
    sqrt(uVect[1] * uVect[1] + vVect[1] * vVect[1]),
    sqrt(uVect[2] * uVect[2] + vVect[2] * vVect[2])
};

// Sélection du vecteur avec la plus grande norme
uint8_t finalVect;
if (normVect[0] > normVect[1]) {
    if (normVect[0] > normVect[2]) {
        finalVect = 0;
    } else {
        finalVect = 2;
    }
} else {
    if (normVect[1] > normVect[2]) {
        finalVect = 1;
    } else {
        finalVect = 2;
    }
}

// Calcul de la direction en degrés à partir du vecteur sélectionné
int dir = (int) (180 / PI) * atan(vVect[finalVect] / uVect[finalVect]);

// Mise à jour de la vitesse des moteurs en fonction de la direction calculée
PWMSetSpeedConsigne(((dir * -VITESSE / 2.0f) / -60.0f) + VITESSE, MOTEUR_GAUCHE);
PWMSetSpeedConsigne(((dir * VITESSE / 2.0f) / -60.0f) + VITESSE, MOTEUR_DROIT);
```

Implémentation logicielle de la stratégie

Cette solution alliant vecteurs et trigonométrie, est efficace mais pas la plus optimale, il n'était pas simple d'agir sur la vitesse et de gérer des cas de navigation plus complexes. Nous avons eu une idée sortant un peu de l'ordinaire, nous sommes arrivés à la mettre en place et la rendre fonctionnelle, pour nous c'était déjà très encourageant pour la suite.

3.2 Contre-réaction linéaire

3.2.1 Principe de fonctionnement

La stratégie dite à contre-réaction linéaire s'inscrit dans la même démarche que la stratégie abordée plus tôt. Cette stratégie utilise la valeur mesurée par chaque télémètre pour déterminer la meilleure commande à transmettre à chacun des deux moteurs. Le principe est plutôt simple : pour chaque mesure de capteur, le robot ajoute à une commande neutre (soit la commande qui permet d'aller droit devant) une contre-réaction proportionnelle à la distance mesurée.

Les fonctions caractéristiques de contre-réactions ne sont pas purement linéaires (voir Figure 2.1.1). En effet, pour le capteur central, le capteur de droite et celui de gauche, cette caractéristique est divisée en deux plages : la plage linéaire et la plage constante qui implique qu'aucune contre-réaction n'est appliquée au-delà d'un seuil. Les caractéristiques pour les capteurs extrêmes suivent le même schéma, mais leur plage linéaire est constituée de deux droites avec des coefficients différents, mises bout à bout (voir Figure 2.1.2). Cette modification a été apportée pour éviter que le robot ne frôle les obstacles au niveau des extrêmes.

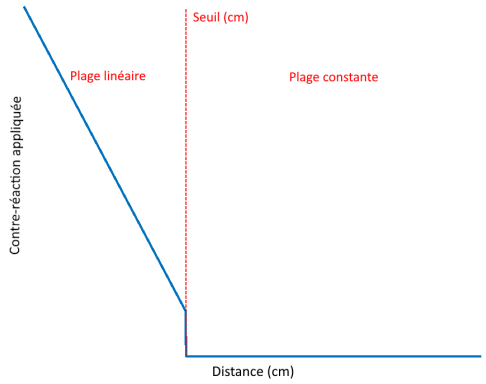

FIGURE 3.2.1 – Représentation de la caractéristique de la contre-réaction centrale, droite et gauche

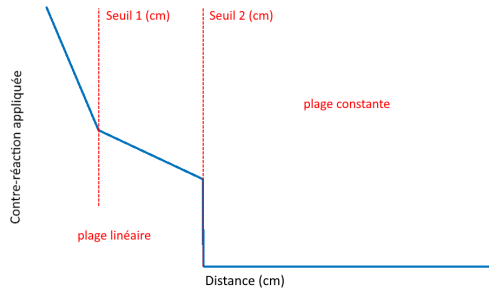

FIGURE 3.2.2 – Représentation de la caractéristique de la contre-réaction aux extrêmes

Il est à noter que ci-dessus, la contre-réaction est exprimée en valeur absolue (voir sous-section Application) avec la plage constante égale à 0.

3.2.2 Choix des contres-réactions

À présent, le principe de fonctionnement étant établi, il faut déterminer les pentes des courbes ainsi, que les seuils à utiliser pour chaque caractéristique. Pour ce faire, des tests ont été réalisés en remplaçant la plage linéaire par une ou plusieurs plages constante afin d'approcher de manière grossière les paramètres susmentionnés. Ces tests ont permis d'obtenir le tableau ci-dessous. Dans ce tableau les points avec les abscisses les plus élevés correspondent aux seuils qui séparent les deux plages des caractéristiques.

Capteurs	Distances (cm)	Contres-réactions	Équations de droite
Centre	30	5	$-1,25.d + 42,5$
	18	20	
Droite et gauche	40	2	$-0,0889.d + 5,5556$
	25	$\frac{10}{3}$	
Extrêmes	40	2	$-0,0889.d + 5,5556$
	25	$\frac{10}{3}$	
	10	4,51	$-0,4481.d + 9,7403$
	5	7,5	

FIGURE 3.2.3 – Points limites et équations de droite des plages linéaire

Ensuite, il est aisé d'appliquer les équations trouvées plus haut aux caractéristiques et le résultat est trois caractéristiques : une pour le centre, une pour la droite et la gauche et une pour les extrêmes.

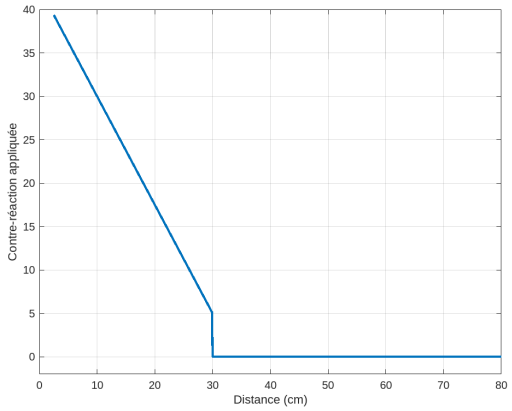

FIGURE 3.2.4 – Caractéristique de la contre-réaction liée au capteur centrale

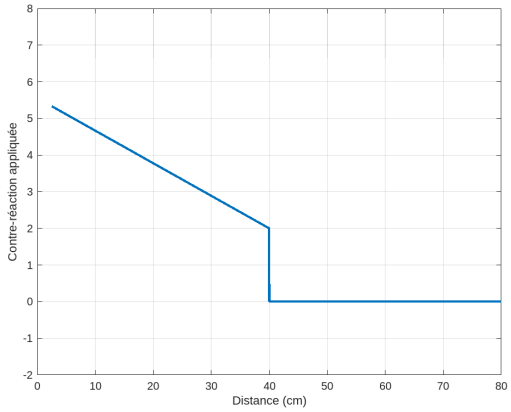

FIGURE 3.2.5 – Caractéristique de la contre-réaction liée aux capteurs droit et gauche

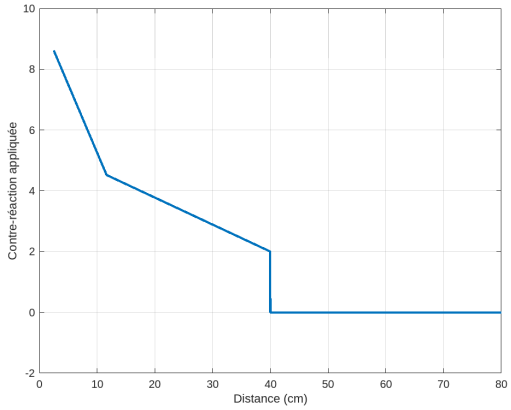

FIGURE 3.2.6 – Caractéristique de la contre-réaction liée aux capteurs extrêmes

3.2.3 Application

Grâce aux caractéristiques de contre-réaction établies, la stratégie peut être implémentée dans le robot. Plusieurs points sont à prendre en compte.

Tout d'abord, lorsque le robot avance en ligne droite, il doit rouler à vitesse maximale. Ceci se traduit ici par l'assertion suivante : "si toutes les contre-réactions sont dans leur plage constante (soit qu'elles valent 0), alors le robot avance à vitesse maximale". Autrement, le robot avance à une vitesse d'exploration inférieure.

De plus, comme expliqué plus haut (cf. Principe), chaque capteur ajoute sa contre-réaction à une commande neutre. Or, le robot possède deux moteurs et par conséquent, le programme doit fournir deux commandes. C'est alors que l'intérêt d'exprimer la contre-réaction en valeur absolue prend tout son sens. Le tableau suivant montre le terme réel qui est ajouté à la commande neutre. En d'autres termes, le programme joue sur le signe de la contre-réaction pour obtenir la commande adéquate : "je vois un obstacle à droite, je tourne à gauche" et inversement. Avec $\delta_{capteur}$, la contre-réaction apportée par un capteur donné.

Capteurs	Contre-réactions relatives appliquées à la commande du moteur droit (%)	Contre-réactions relatives appliquées à la commande du moteur gauche (%)
Centre	$-\delta_{centre}$	$-\delta_{centre}$
Droit	δ_{droit}	$-\delta_{droit}$
Extrême droit	δ_{ex_droit}	$-\delta_{ex\_droit}$
Gauche	$-\delta_{gauche}$	δ_{gauche}
Extrême gauche	$-\delta_{ex\_gauche}$	δ_{ex_gauche}

FIGURE 3.2.7 – Signe de la contre-réaction pour chaque capteur et chaque commande

Finalement, la plage constante de la contre-réaction et testée grâce à un *if* ce qui permet également de simplifier l'implémentation du premier point. La figure suivante montre l'implémentation des différents points abordés dans cette section. Pour des raisons de lisibilité et de redondance, seulement l'implémentation du capteur centrale, et du capteur extrême gauche et de l'envoi de la commande aux moteurs est montré. En effet, le code le seulement changement entre les blocs de codes qui applique les contre-réactions et le signe de celle-ci. (cf. Figure 2.3.1).

```
1  float baseGauche = VITESSE;
2  float baseDroite = VITESSE;
3  int isViteVite = 1;
4
5  if (robotState.distanceTelemetreMelanchon <= 45) {
6      isViteVite = 0;
7
8      if (robotState.distanceTelemetreMelanchon <= 10) {
9          baseGauche += (-0.4481) * robotState.distanceTelemetreMelanchon + 9.7403;
10         baseDroite -= (-0.4481) * robotState.distanceTelemetreMelanchon + 9.7403;
11     } else {
12         baseGauche += (-0.0889) * robotState.distanceTelemetreMelanchon + 5.5556;
13         baseDroite -= (-0.0889) * robotState.distanceTelemetreMelanchon + 5.5556;
14     }
15 }
16 [...]
17 if (robotState.distanceTelemetreCentre <= 40) {
18     isViteVite = 0;
19
20     baseGauche -= (-1.25) * robotState.distanceTelemetreCentre + 42.5 + ((robotState.
       distanceTelemetreMelanchon + robotState.distanceTelemetreGauche) > (robotState.
       distanceTelemetreLePen + robotState.distanceTelemetreDroit) ? VITESSE : -VITESSE);
21
22     baseDroite -= (-1.25) * robotState.distanceTelemetreCentre + 42.5 + ((robotState.
       distanceTelemetreMelanchon + robotState.distanceTelemetreGauche) > (robotState.
       distanceTelemetreLePen + robotState.distanceTelemetreDroit) ? -VITESSE : VITESSE);
23 }
24 [...]
25 if (isViteVite) {
26     PWMSetSpeedConsigne(VITE_VITE, MOTEUR_GAUCHE);
27     PWMSetSpeedConsigne(VITE_VITE, MOTEUR_DROIT);
28     [...]
29 } else {
30     PWMSetSpeedConsigne(baseGauche, MOTEUR_GAUCHE);
31     PWMSetSpeedConsigne(baseDroite, MOTEUR_DROIT);
32     [...]
33 }
```

Implémentation logicielle de la stratégie

Cette figure dense mérite quelques précisions. Dans un premier temps, les lignes 20 et 22 sont très longues. Toutefois, elles exploitent les structures ternaires, ce qui permet de simplifier le code. L'ajout de ces structures permet de ne pas faire reculer le robot tout droit : sans quoi, il se bloque face aux murs. Par ailleurs, les lignes 25 à 31 montre l'envoi des commandes aux moteurs et il montre également la gestion de la vitesse maximale conformément avec l'assertion précédemment énoncée.

Finalement, ce code est exécuté à chaque fin de la séquence d'acquisition des capteurs, soit environ à 40 Hz lorsqu'ils sont exploités à leur vitesse maximale, ce qui est le cas ici.

4 Conclusion et limites de la stratégie choisie

La stratégie de contre-réaction linéaire comme explicitée dans la section précédente permet d'avoir des mouvements relativement fluides et une bonne efficacité dans les configurations type couloir. Cependant, lorsque tous les capteurs voient un obstacle sauf le centre, étant donné que ce dernier est le seul à pouvoir faire reculer le robot, le robot avance. Par ailleurs, le robot à tendance à faire des demi-tours violant lorsqu'il arrive avec une grande vitesse face à un obstacle. Ce comportement est parfois préjudiciable lorsque les obstacles sont très rapprochés. Finalement, l'utilisation de fonction linéaire permet d'avoir une vitesse qui s'adapte aux obstacles à proximité.