Système temps-réel, modélisation réaliste et implantation multi-taches

Julien BONNARDEL

&

Oualid Srhiri

&

Bertrand E. Talaki

9 février 2018

Table des matières

1 Introduction

Afin d'avoir une idée de comment se réalise l'implantation de systèmes multi-tâches, ce projet consiste à implémenter deux tâches sur un Robot Lego : suivre une ligne noire et détecter ainsi qu'éviter un obstacle. Le robot devra faire demi tour à la détection d'un obstacle et suivre la ligne dans l'autre sens.

Il faudra passer par la modélisation, la simulation et la validation avant d'implémenter et d'exécuter ces deux tâches. Il s'agit là des étapes à suivre dans le développement d'un système critique temps réel.

L'implantation des systèmes de contrôle est dite sure lorsqu'elle respecte deux propriétés : temps réel et déterministe. Pour un système déterministe, le résultat d'une simulation doit être connu à n'importe quel moment. Ces deux propriétés sont centrales pour ce projet.

Ce projet nous permet de découvrir les méthodes classiques de conception et d'implantation sures. L'objectif est d'implanter un système stable multi-taches.

2 Modélisation et études théoriques

Notre but dans cette partie est d'arriver à une modélisation réaliste du comportement du robot via une loi de commande stable qu'on discrétisera (pour être utilisable en informatique).

Le Robot utilisé est composé de deux roues (et deux moteurs), deux capteurs de couleur et un capteur d'obstacle. On utilise un contrôleur d'orientation pour ajuster l'orientation du robot et un contrôleur de vitesse pour ajuster les vitesses des moteurs.

2.1 Loi de commande d'orientation

θ représente l'orientation du robot, Vg et Vd sont les vitesses des roues gauche et droite et l représente la distance inter roues, θ' : On modélise alors la position du robot de la façon suivante :

$$\begin{aligned} x' &= \frac{v_g+v_d}{2}\ cos(\theta) \\ y' &= \frac{v_g+v_d}{2}\ sin(\theta) \\ \theta' &= \frac{v_g-v_d}{l} \end{aligned}$$

En considérant la différence de vitesses comme variable de commande : $u_\theta = v_g - v_d$, on remarque qu'on obtient la relation :

$$\frac{\theta'}{u_\theta} = \frac{v_g-v_d}{l(v_g-v_d)} = \frac{1}{l}$$

et en appliquant la transformée de Laplace comme suit :

$$\mathcal{L}(\theta') = \frac{\theta}{s}$$

Ainsi :

$$\frac{\theta}{u_\theta} = \frac{1}{l*s}$$

on obtient la fonction de transfert de θ en fonction de u_θ :

$$H_\theta(s) = \frac{\theta}{u_\theta} = \frac{1}{l*s}$$

Nous contrôlons l'orientation du robot par un contrôleur de type PI(proportionnel intégral) dont la fonction de transfert est :

$$C_\theta(\mathrm{s}) = \frac{k_{i\theta}}{s} + k_{p\theta}$$

où $k_{i\theta}$ est le coefficient de l'action intégrale (pour stabiliser la correction dans le temps) et $k_{p\theta}$ le coefficient de l'action proportionnelle. Ce contrôleur prend en entrée l'écart entre la sortie d'orientation θ et l'orientation souhaitée.

On calcule alors la fonction de transfert du système en boucle fermée (à l'aide de la formule de Black), qui nous permettra d'évaluer la stabilité de notre système :

$$H(s) = \frac{C_\theta(s)*H_\theta(s)}{1+C_\theta(s)*H_\theta(s)} = \frac{\frac{k_p\theta}{l}s+\frac{k_i\theta}{l}}{s^2+\frac{k_p\theta}{l}s+\frac{k_i\theta}{l}}$$

On pose $\alpha = 1/l$.

On obtient alors :

$$k_{i\theta} = \omega^2/\alpha \text{ et } k_{p\theta} = 2\xi\omega/\alpha$$

ξ étant l'amortissement et ω la pulsation du système. Notre système s'identifie alors à un système du second ordre dont les pôles sont à partie réelle $(-\omega\xi)$ négative et assurent ainsi la stabilité. En ce qui concerne les performances (rapidité, dépassement...) il faut choisir les valeurs ω et ξ pour avoir les performances désirées en fonction des formules de dépassement et de temps de réponse (fournies dans le cahier des charges) notamment.

On choisit les valeurs $k_p\theta = 0.4$ et $k_i\theta = 0.2$ après avoir modifié les valeurs et testé avec Simulink, puis à nouveau avec le robot LEGO une fois l'implémentation effectuée.

2.2 Loi de commande de vitesse

En ce qui concerne le correcteur de vitesse, c'est un correcteur proportionnel qui se base sur l'écart entre la correction d'orientation u_θ et la commande de vitesse u_δ de la sorte :

$$v_g = \frac{u_\theta+u_\delta}{2} \; v_d = \frac{u_\delta-u_\theta}{2}$$

Concrètement, si l'on se trouve dans un virage, cela permet d'avoir une vitesse réduite, étant donné que la commande u_θ aura une valeur absolue plus grande.

2.3 Echantillonage

Pour discrétiser notre modèle on utilise la méthode de Tustin avec une période d'échantillonage T, car cette méthode s'avère garder la stabilité (contrairement à la méthode d'Euler en avant) dans l'espace des Z (elle aboutit dans le cercle unitaire), on effectue la transformation suivante :

$$s = \frac{2(z-1)}{T(z+1)}$$

En ce qui concerne le choix de la période d'échantillonnage, nous avons utilisé la règle vue en cours $\omega_c * T$ entre 0.05 et 0.14.

En premier lieu nous avons calculé la pulsation critique du système en boucle ouverte à l'aide des fonctions Matlab suivantes :

$$sysbo = tf([0\ 0.4\ 0.2], [8.3333\ 8.3333\ 0])$$
$$[gpfwc] = margin(sysbo)$$

La fonction matlab margin nous donne la pulsation critique du système à environ 1.

Par la suite nous avons cherché à avoir la période d'échantillonage la plus petite possible en effectuant : $T = \omega_c * 0.05$

Ainsi nous avons obtenu $T = 0.0025$.

2.4 Hurwitz-Routh

Pour entamer l'étude de stabilité continue, nous effectuons un changement de variable bilinéaire sur la fonction de transfert continue (avec T) pour repasser dans le plan gauche et utiliser les mêmes critères que pour un système continu :

$$z = \frac{1+w}{1-w}$$

On obtient ainsi le polynôme caractéristique du système (polynôme du dénominateur) :

$$\pi(w) = \frac{4}{T^2} * w^2 + \frac{2k_p\theta}{TL} * w + \frac{k_i\theta}{L}$$

En utilisant le script matlab fourni, on calcule la matrice d'Hurwitz pour étudier la stabilité du polynôme. En effet, si tous les "mineurs principaux dominants" (ici les determinants de sous matrices carrées) de la matrice sont positifs, alors le système est stable.

On effectue la commande suivante pour déclarer les coefficients du polynôme :

$$p = [\frac{4}{T^2}\ \frac{2k_p\theta}{TL}\ \frac{k_i\theta}{L}]$$

puis on utilise la commande suivante pour calculer la matrice d'Hurwitz :

$$H = hurwitz(p, 2)$$

Avec les valeurs kptheta = 0.4, ki = 0.2, alpha = 8.3333, T = 0.002

On obtient (H(0,0) = 3330) et det(H)=2.53e5.

Ainsi, on a la confirmation que notre système après discretisation est bien stable.

3 Modèle simulink du système suiveur de ligne

Les fondements théoriques étant assimilés, on passe à la création d'un modèle simulink permettant au robot de suivre une ligne.

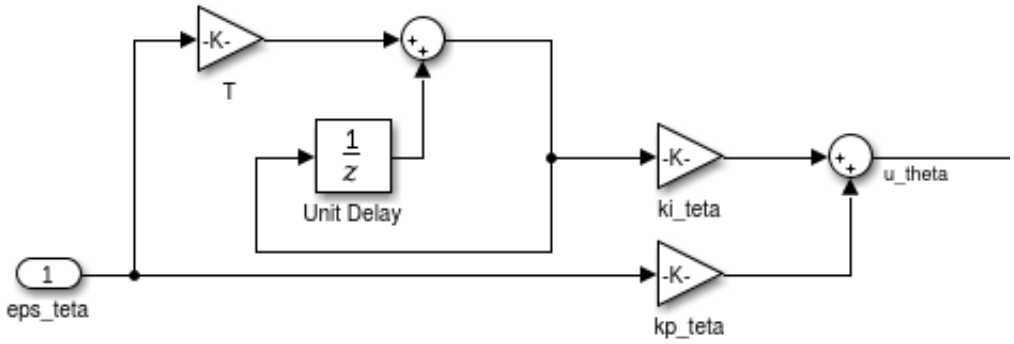

FIGURE 1 – Fonction de transfert des correcteurs.

Nous avons modifié la fonction de transfert des correcteurs dans Simulink en introduisant la méthode de Tustin que nous avons évoqué dans la partie théorique (meilleure que Euler en avant).

De plus pour avoir un fonctionnement optimal, nous avons également dû réaliser un calibrage des capteurs en début de fonctionnement du Robot LEGO. En effet, les circuits pouvant avoir des nuances de noir et blanc et les capteurs n'étant pas identiques à chaque fois, il faut recaliber les capteurs en intéragissant avec l'écran LCD du robot. (cf. Fin du rapport, Figure 11)

Outre l'étude théorique, nous avons également réadapté légèrement les valeurs des différentes variable de contrôle $k_p\theta$, $k_i\theta$ et T, en veillant à rester conforme au cahier des charges (pas de valeur de $T\omega_c$ dans $[0.05, 0.14]$.

4 Modèle simulink du système anti-collision

4.1 Module anti-collision

Tout en conservant les capacités de suiveur de ligne du robot, on implémente un modèle simulink d'anti-collision qui permet au robot d'éviter les obstacles. On choisit la stratégie d'évitement suivante :

— Le robot s'arrête (pas instantanément du fait de la latence des moteurs).
— Puis il fait un demi tour.

Pour conserver le caractère de suiveur de ligne, il faut intégrer la capacité au robot de reprendre la ligne. On suppose que le robot fait un demi tour en partant sur sa gauche lorsqu'il rencontre un obstacle. Autrement dit le capteur gauche du robot va observer une séquence "Blanc-Noir-Blanc". Le robot a donc besoin de mémoriser la séquence de couleur. Pour ce faire on utilise un bloc de mémorisation avec reset qui permet au contrôleur de revenir dans l'état antérieur au mode anti-collision.

Le système fonctionne sous deux modes, détection d'obstacle et suiveur de ligne. A la détection d'un obstacle le robot s'arrête et fait un demi tour. Pour conserver le caractère de suiveur de ligne, il faut rendre le robot capable de revenir au mode suiveur de ligne. Pour cela on suppose que le robot fait un demi tour en partant sur sa gauche lorsqu'il rencontre un obs-

tacle. Le capteur gauche du robot va observer une séquence "Blanc-Noir-Blanc". Il a donc besoin de pouvoir mémoriser cette séquence de couleur. Pou ce faire, on utilise un bloc de mémorisation avec reset qui permet au contrôleur de revenir dans un état antérieur (dans notre cas en mode suiveur de ligne).

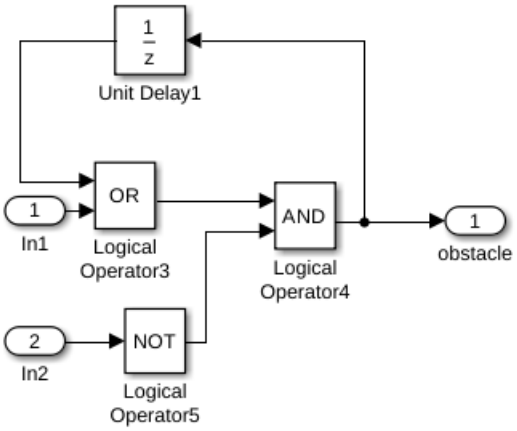

FIGURE 2 – Bloc mémoire avec reset.

Puis on crée le sous-système qui permet de reconnaître la séquence "Blanc-Noir-Blanc" et de savoir si il est dans le mode suiveur de ligne ou dans le mode détection d' obstacle.

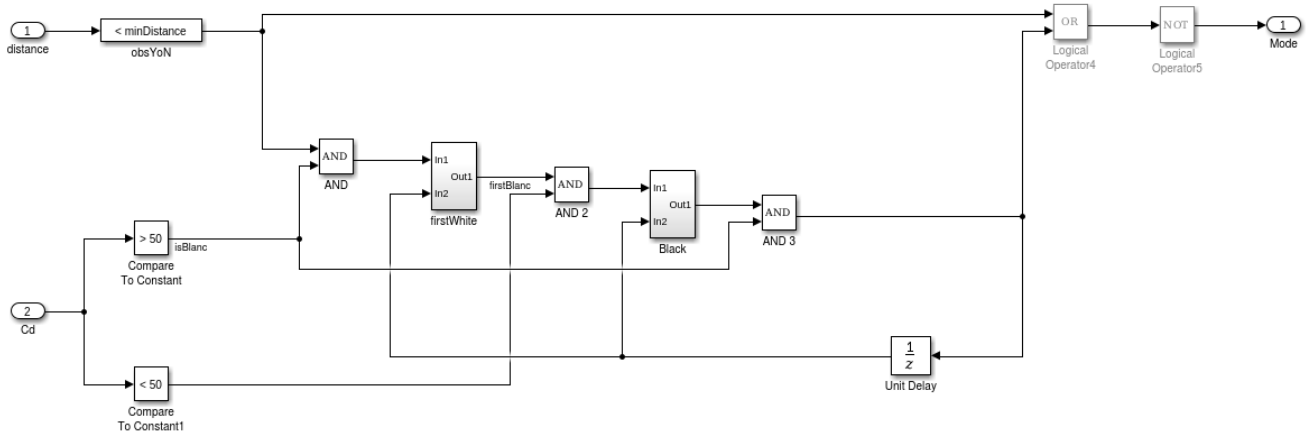

FIGURE 3 – Bloc de changement de mode.

On teste la distance (valeur reçu par le capteur) avec une distance minimale fixée arbitrairement. Si la distance est inférieure à cette distance minimale, le robot passe en mode anti-collision et il effectue un demi tour. Sinon il reste en mode suiveur de ligne.

4.2 Module de comportement

Après avoir finalisé les modules d'anti-collision et suiveur de ligne, il faut modifier le comportement du robot en fonction de son état. Lorsque le robot détecte un obstacle, il effectue un demi tour. Sinon il reste dans le mode suiveur de ligne. Pour cela on utilise un gain négatif pour inverser le sens de rotation d'un de ses moteurs (pour passer ce moteur en marche arrière).

Le bloc du comportement du robot étant pratiquement implémenté, il faut rajouter des

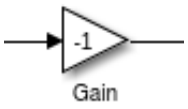

FIGURE 4 – Marche arrière.

switchs pour le changer de mode (suiveur de ligne ou anti-collision). Le choix de la vitesse à 0.4 est purement arbitraire et permet au robot de faire le demi tour à une vitesse acceptable.

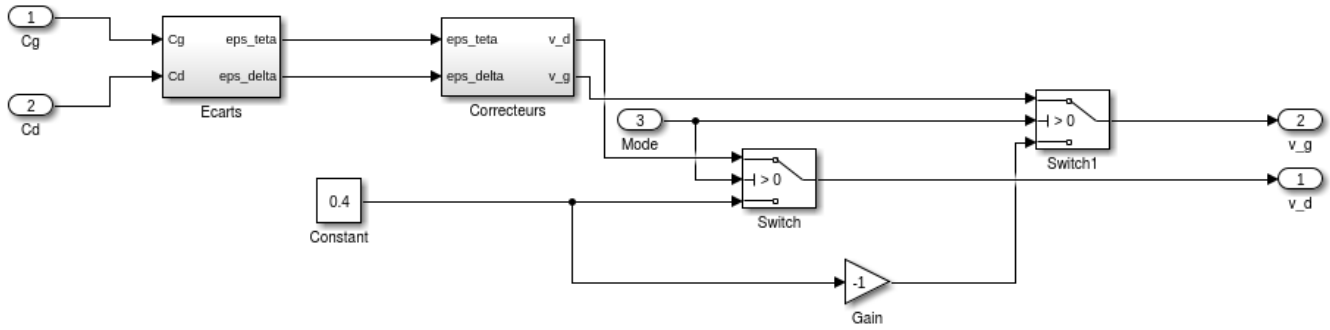

FIGURE 5 – Bloc de comportement.

4.3 Module de contrôle

Le système de contrôle anti-collision et suiveur de ligne du robot Lego est le suivant :

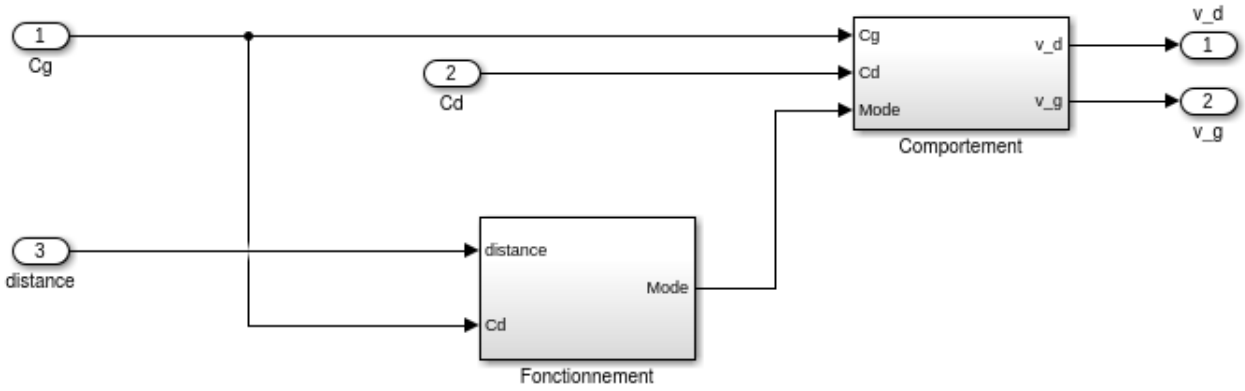

FIGURE 6 – Système de contrôle.

Il se compose donc du bloc "Fonctionnnement" qui permet de savoir si le robot est en suiveur de ligne ou a détecté un obstacle. Lorsque ce mode est connu, il envoie l'information au bloc "Comportement" qui s'occupe soit de faire un demi tour, soit de suivre la ligne.

Pour valider le tout, on simule l'obstacle au sein de simulink à l'aide d'un générateur de créneau (simulation d'un "obstacle" toutes les trois secondes).

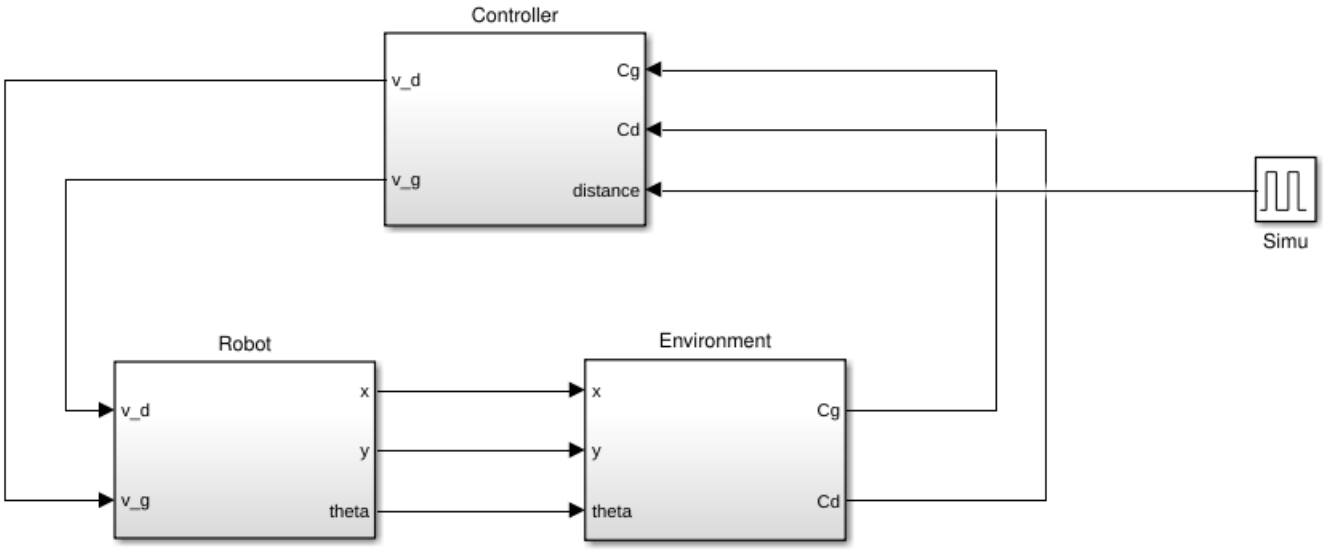

FIGURE 7 – Environnement du robot.

5 Implémentation du système anti-collision

5.1 Mise en place du cadre de travail

Après avoir validé le modèle simulink, on passe à l'implémentation en générant des fichiers exécutables et en complétant le code C (fichier glue.c) pour que le robot soit fonctionnel.

Le modèle respecte bien la séparation du contrôleur en deux sous-systèmes (cf. Figure 6) "Comportement" et "Fonctionnement".

Les fichiers exécutables sont générés en suivant la chaîne de compilation suivante :

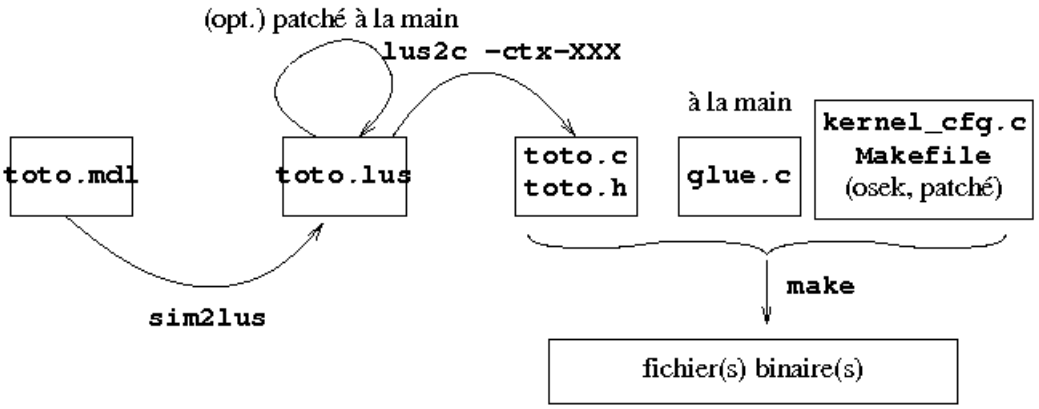

FIGURE 8 – Chaine de compilation.

On crée les fichiers lus des deux sous-système, "Comportement.lus" et "Fonctionnement.lus" en tapant la commande "mdl2lus ../ExRobotAndEnvironmentController.mdl -system Fonctionnement (de même Comportement)".

Puis on génére les fichiers c et h de ces derniers, "Comportement.c Comportement.h" et "Fonctionnement.c Fonctionnement.h" en tapant la commande "lus2c Comportement.lus Com-

portement -ctx-static (de même pour Fonctionnement)".

Pour finir, on importe les fichiers kernel_cfg.c, kernel_id.h et un Makefile que l'on modifie de la manière suivante :

```
1
2 # Application name
3 TARGET=MyProgram
4
5 # Where to find user C code
6 USR_PATH=.
7
8 # -----------
9 # user C code
10 # -----------
11 USR_CFILES=\
12    kernel_cfg.c\
13    Fonctionnement.c\
14    Comportement.c \
15    glue.c\
16
17 include $(MDL2LUS20SEK)/generic.mak
```

FIGURE 9 – Makefile.

Et on édite deux fichiers "comportement_ext.h" et "fonctionnement_ext.h" où l'on définit des constantes pour le bon fonctionnement du robot($k_p\theta$, $k_i\theta$ ainsi que la variable minDistance qui modifie la marge de détection du détecteur d'obstacle et une valeur approchée de π).

5.2 Implémentation du fichier glue.c

Le cadre de travail étant fait. Il suffit de compléter le code du fichier glue.c en utilisant l'API nxtOSEK qui se trouve sur le site "http ://lejos-osek.sourceforge.net/ecrobot_c_api.htm".

— Initialisation et désactivation des capteurs :

```
66 /* Init and terminate OSEK */
67 void ecrobot_device_initialize() {
68    /*
69       HERE: put here specific code that will be executed ONCE
70             when the application starts
71       TYPICALLY: initialization of (light) sensors
72    */
73         ecrobot_init_sonar_sensor(NXT_PORT_S3); //Distance
74         ecrobot_set_light_sensor_active(NXT_PORT_S2); //Gauche
75         ecrobot_set_light_sensor_active(NXT_PORT_S1); //Droite
76 }
77
78 void ecrobot_device_terminate() {
79    /*
80       HERE: put here specific code that will be executed ONCE
81             when the application stops
82       TYPICALLY: finalization of (light) sensors
83    */
84         ecrobot_term_sonar_sensor(NXT_PORT_S3); //Distance
85         ecrobot_set_light_sensor_inactive(NXT_PORT_S2); //Gauche
86         ecrobot_set_light_sensor_inactive(NXT_PORT_S1); //Droite
87 }
```

FIGURE 10 – Initialisation et désactivation des capteurs.

— Initialisation du mode et calibrage des capteurs :

```
102 void usr_init(){
103         // Init du buffer
104         bufferMode[0] = true;
105         bufferMode[1] = true;
106 
107         GetResource(lcd);
108 
109         display_clear(0);
110         display_goto_xy(0,0);
111         display_string("White init");
112 
113         // Init les capteurs sur le blanc
114         while(!ecrobot_is_ENTER_button_pressed()) {
115                 leftW = ecrobot_get_light_sensor(NXT_PORT_S2) + 20;
116                 rightW = ecrobot_get_light_sensor(NXT_PORT_S1) + 20;
117 
118                 display_clear(1);
119                 display_goto_xy(0, 0);
```

FIGURE 11 – Début de la fonction de calibrage.

— Implémentation de la tâche de basse priorité (sous-système Fonctionnement) :

```
306 TASK(LowTask) { // Fonctionnement
307 
308         display_goto_xy(0,0);
309         display_string("Block Mode");
310         display_update();
311 
312          Fonctionnement_I_Cd_n(grade(rightB, rightW, ecrobot_get_light_sensor
    (NXT_PORT_S1)));
313         Fonctionnement_I_distance(checkDistance(ecrobot_get_sonar_sensor(NXT_PORT_S3)));
314 
315         Fonctionnement_step();
316 
317         TerminateTask();
318 }
```

FIGURE 12 – Tâche de basse priorité.

— Implémentation de la tâche de haute priorité (sous-système Comportement) :

```
320 TASK(HighTask) { // Comportement
321         if( count == 0 ){
322                 currentMode = (currentMode + 1) % 2;
323         }
324
325         display_goto_xy(0,0);
326
327         if( bufferMode[currentMode % 2] ){
328                 display_clear(1);
329                 display_goto_xy(1,6);
330                 display_string("Go ahead");
331         } else {
332                 display_clear(1);
333                 display_goto_xy(1,3);
334                 display_string("Obstacle");
335                 display_goto_xy(1,7);
336                 display_string("Turn !");
337         }
```

FIGURE 13 – Début de la tâche de haute priorité.

— Implémentation des fonctions de sorties de vitesse moteur en fonction de la sortie du système de contrôle :

```
258 /*-------------------------------
259   Speed and mode Functions
260 -------------------------------
261 - Control engine right and left
262 - Change mode when there is an obstacle
263 -------------------------------*/
264
265 void speedMotor(U8 port_id, _real speed, int place){
266         int speed_cast_to_int = (int) (speed * 100);
267
268         if( speed_cast_to_int > speedMax ){
269                 speed_cast_to_int = speedMax;
270         }
271
272         if ( speed_cast_to_int < -speedMax ){
273                 speed_cast_to_int = -speedMax;
274         }
275
276         display_goto_xy(0, place);
277         display_int(speed_cast_to_int,3);
278         display_update();
279
280         return ecrobot_set_motor_speed(port_id, speed_cast_to_int);
281 }
282
```

FIGURE 14 – Fonction de vitesse.

```
283 void Comportement_0_v_d(_real speed){
284         speedMotor(NXT_PORT_A, speed, 1);
285 }
286
287 void Comportement_0_v_g(_real speed){
288         speedMotor(NXT_PORT_B, speed, 2);
289 }
290
291
292
293 void Fonctionnement_0_Mode(_boolean mode){
294         bufferMode[(currentMode + 1) % 2] = mode;
295 }
```

FIGURE 15 – Vitesse des moteur .

Une fois ces étapes effectuées, on compile avec la commande "make", on connecte le robot par USB au pc et on tape la commande "t2n -put MyProgram.rxe" pour envoyer le code dans le robot. Une fois le robot fonctionnel, on effectue des tests sur différentes pistes (nuances différentes de noir, ...) pour modifier les valeurs choisies (notamment la vitesse des moteurs, distance minimale du capteur d'obstacle, ...) pour avoir un fonctionnement acceptable.

6 Conclusion

Notre projet robot a permit tout d'abord une implantation mono-tache (suiveur de ligne). Pour cela, nous avons envisagé un modèle réaliste, en prenant soin de comprendre et d'appliquer les concepts théoriques vus en cours.

La complexité du système a augmenté lorsqu'on a ajouté la détection d'obstacles, le faisant ainsi évoluer en un système multi-tâches. La décomposition du contrôleur en deux sous-systèmes a permis de répondre aux problématiques temps réels. Un contrôleur de planification est alors utilisé pour définir la tâche prioritaire et la tâche à exécuter.