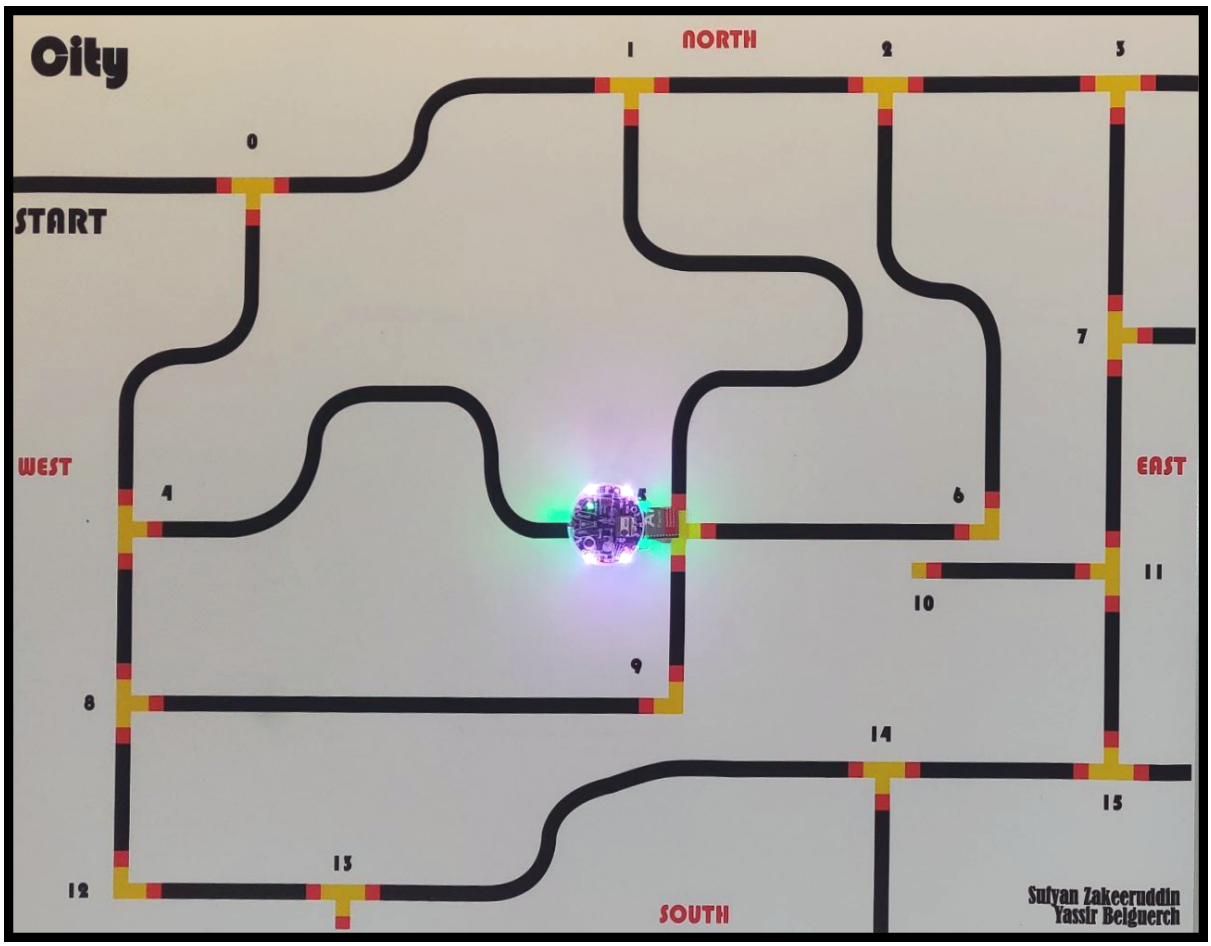
École Polytechnique fédérale de Lausanne

MICRO-315 - Mini-projet - 16 Mai 2022

CityBot



Groupe N° 33 : Yassir BELGUERCH
Sufyan Shaik Mohammed ZAKEERUDDIN

Table des matières

1 Introduction

Dans le cadre du cours « Systèmes embarqués et robotique » en 3ème année de Microtechnique, nous avons réalisé un mini-projet en langage C sur le robot e-puck2. La donnée du projet est ouverte, à condition de s'intégrer au système d'exploitation ChibiOS/RT, d'utiliser les moteurs, un capteur de distance ainsi qu'un autre capteur étudié durant les travaux pratiques.

Dans notre projet, à travers une ville 2D dessinée sur papier A0 et constituée de différents croisements, le robot devra se rendre d'un croisement initial à un croisement final sur le principe d'un «line follower» au moyen de sa caméra. Le chemin emprunté sera le plus court, calculé à l'aide de l'algorithme de Dijkstra. Le robot effectue un aller-retour. Les capteurs de distance infrarouges et Time of Flight sont utilisés pour détecter puis contourner un éventuel obstacle positionné sur son chemin.

2 Setup

2.1 Le miroir

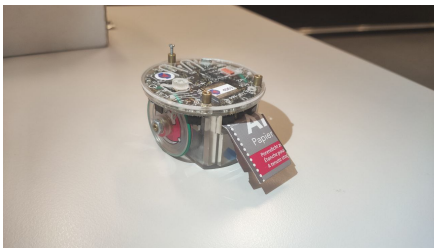


Figure 1 – Vue d'ensemble de l'e-puck2, avec un miroir (collé sur un carton) placé entre le ToF et la caméra.

Figure 2 – Miroir utilisé. On peut voir que le carton est penché de telle sorte à ce que la caméra soit bien visible et rentre dans l'angle du miroir.

Afin de réaliser un line follower, il nous fallait placer un miroir afin que la caméra voit la ligne noire à suivre. Le miroir est collé sur un bout de carton. Néanmoins, plusieurs contraintes se sont avérées :

- Le miroir doit être suffisament fin. Un miroir trop épais fait que la trajectoire de la caméra venait se fondre dans l'épaisseur du miroir (et non sur sa surface réfléchissante).
- Un miroir suffisament léger pour que le carton ne se penche pas trop sous l'effet du poids du miroir et soit ajustable.
- Comme nous utilisons aussi le ToF, il était difficile d'utiliser les vis sur l'interface en verre du robot pour venir visser un miroir. En effet, le ToF deviendrait inutilisable. Ainsi, notre setup carton + miroir doit pouvoir se glisser dans la fine fente se situant entre le ToF et la caméra.

2.2 La ville

La ville a été dessinée avec des lignes d'épaisseur 15 mm. Nous avons fait au préalable des tests avec des lignes d'épaisseur 10 mm et 20 mm ; les deux types résultaient en un bon fonctionnement, donc nous avons fait le choix de prendre une épaisseur entre les deux.
La seule contrainte était de ne pas avoir une ligne de trop grande épaisseur afin qu'elle ne couvre pas tout le champ de la caméra.

Nous avons décidé de l'imprimer sous format A0 afin de pouvoir dessiner le plus de chemins

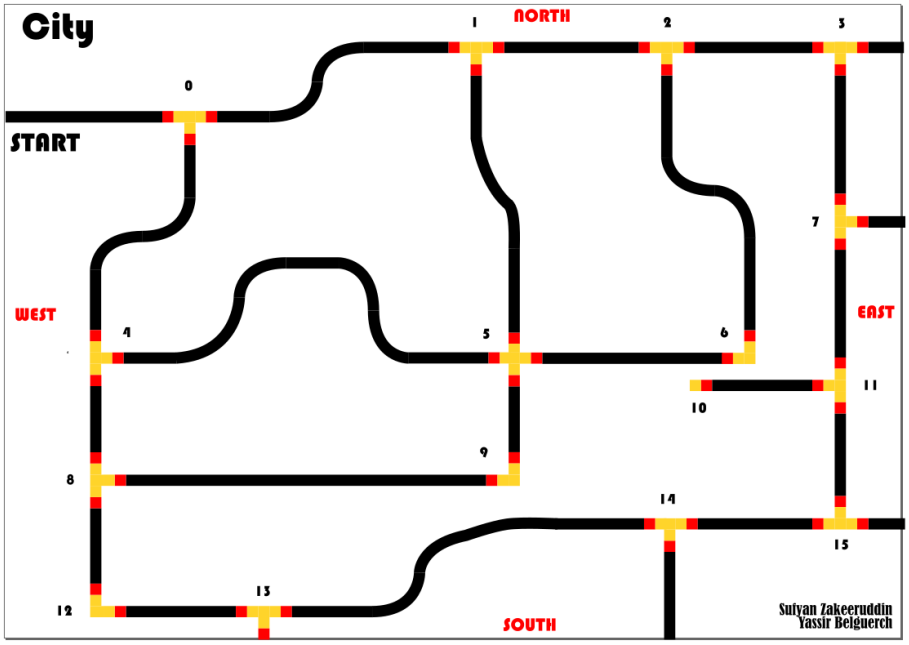


FIGURE 3 – Voici la ville, qui a été imprimée en format A0 (841 x 1189 mm). Le noeud "0" est celui où le robot débute son chemin. A chaque croisement (i.e. à chaque fois que le robot voit un carré rouge), une instruction est donnée afin de savoir dans quelle direction avancer, selon le plus court chemin. Le robot peut circuler entre les 16 noeuds indiqués sur l'image.

divers et variés possibles. On peut ainsi voir différentes lignes courbées que le robot est capable de suivre.

3 Principe de fonctionnement

3.1 Modélisation de la ville

Nous détaillons la modélisation de la ville dans le paragraphe qui suit.

3.1.1 Croisements

Chaque croisement est modélisé, à travers l'usage d'un "struct", comme un noeud, chacun possédant un identifiant UID. La ville est modélisée à travers 16 noeuds, chacun possèdant 2 tableaux avec les noeuds voisins et leurs distances respectives.

Après que l'algorithme de Dijkstra ait calculé le plus court chemin (cf. sec. 3.1.2), l'ensemble des noeuds à parcourir se trouve dans un tableau node_path. Afin de savoir dans quelle direction tourner à un croisement ($right, left, forward$), on compare l'UID du noeud actuel avec celui du prochain noeud en se servant du fait que la ville est un quadrillage carré. Dans notre cas, comme nous avons 16 noeuds, city_line_size = 4. Par exemple, si le prochain noeud se trouve à la ligne suivante, l'UID correspondant sera celui du noeud actuel additionné avec city_line_size. (cf. fig. 4)

Il convient de souligner que notre modélisation (quadrillage) permet au final d'implémenter n'importe quel type de ville. La ville dessinée n'est pas obligée de respecter un positionnement en quadrillage carré ; c'est uniquement l'agencement des UID qui doit le respecter. Ainsi, la ville dessinée (cf. fig. 3) n'est qu'un exemple parmis tant d'autres d'une implémentation possible.

3.1.2 Calcul du plus court chemin : principe de l'algorithme de Dijkstra

Après la lecture de la ville et à partir du croisement de départ et d'arrivée attribué, le programme calcule le plus court chemin associé à l'aide de l'algorithme de Dijkstra.
Dans la structure déclarée du noeud, 3 variables sont uniquement utilisées pour Dijkstra : *in*, *access* et *parent*. Le premier est un booléen indiquant si le noeud en question devra être traité

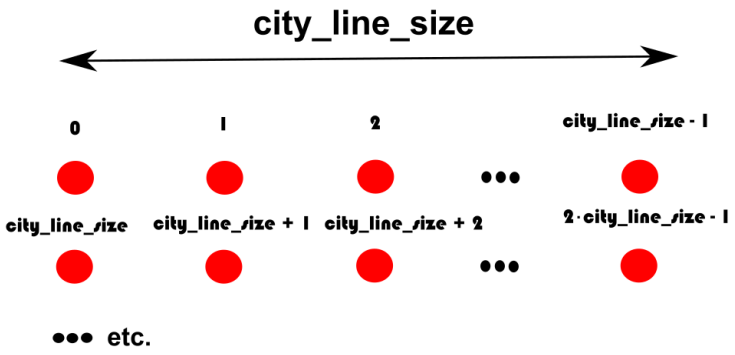


FIGURE 4 – Quadrillage de la ville. Les numéros indiqués représentent le UID des noeuds respectifs

par l'algorithme ou non. Le deuxième indique le temps de parcours accumulé depuis le noeud de départ. Le troisième indique le noeud voisin lorsqu'on remonte le chemin le plus court.

L'algorithme entretient une file d'attente (*queue*) itérativement mise à jour avec les noeuds les plus proches aux premiers indices. Un noeud n'ayant pas encore été traité (i.e. avec *in* à *true*) sera étudié en explorant ses voisins pas encore traité et en mettant à jour leur valeur *access*. Ce travail est effectué sur chaque noeud jusqu'à ce qu'ils aient tous été traité. L'algorithme prend alors fin et le plus court chemin d'un noeud A à un noeud B peut être retracé en remontant les *parents* depuis B.

Pour suivre le chemin correspondant, l'UID des noeuds le constituant sont placés dans un tableau *node_path* pouvant comporter 10 UID au maximum. Le noeud d'arrivée est le 1er élément, son parent est le suivant, et ainsi de suite. Les éléments après le noeud de départ dans le tableau sont des "-1", afin que le tableau soit rempli.

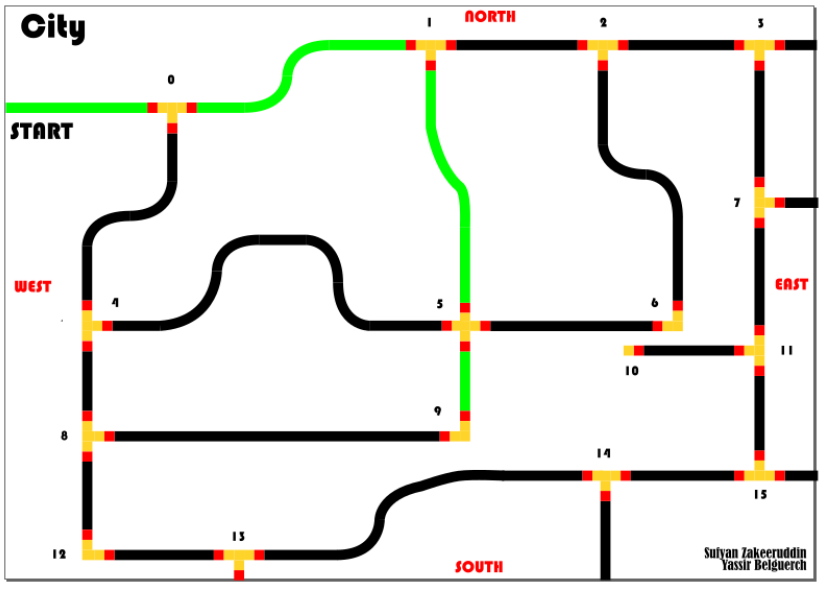


FIGURE 5 – Exemple de l'algorithme de Dijkstra. Il y a plusieurs chemins possibles pour arriver au noeud 9 : 0-4-8, 0-4-5, 0-1-5, 0-1-2-6-5. L'algorithme trouve alors que le 3ème chemin est le plus rapide.

3.2 Suivi du chemin : utilisation de la caméra

3.2.1 Détection ligne noire

Pour suivre le chemin en *freepath*

La détection de la ligne noire se fait à peu près comme vue dans le TP4, nous utilisons

aussi un simple régulateur P pour les roues qui suffit largement pour suivre les courbes avec un assez bon rejet des perturabations.

Pour reprendre le droit chemin après contournement d'obstacle

En arrivant sur la ligne noire, après contournement de l'obstacle, la caméra du robot ne verra que du noir. Ainsi, pour la détecter il s'agit simplement de faire la moyenne sur toute l'image afin de la comparer à un certain seuil. C'est naturellement un seuil relativement bas (l'intensité du noir étant faible) que nous avons déterminé par expérimentation. (figure 6)
Cette détection correspond au *blackline measurement* dans la FSM à la figure 9.

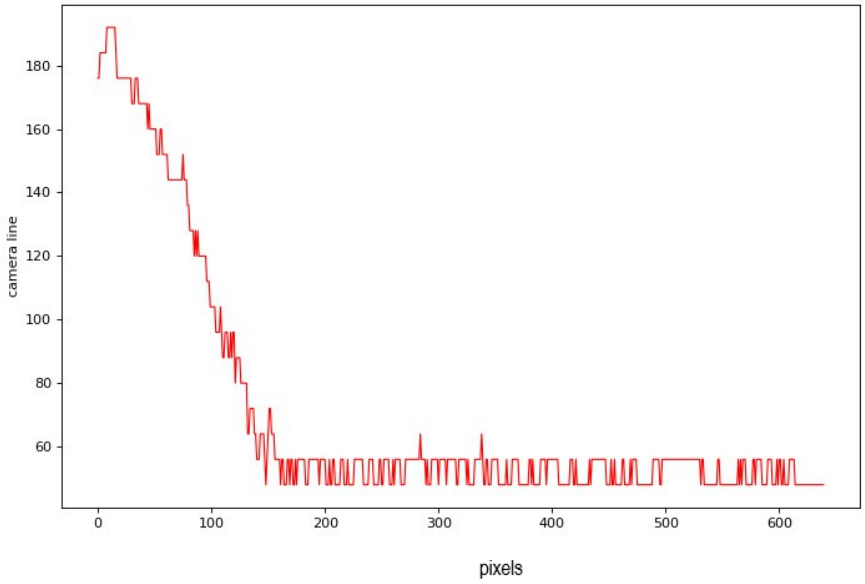


Figure 6 – Détection de la ligne noire après contournement de l'obstacle. On peut néanmoins voir la présence du bruit sur la gauche du graphe

3.2.2 Détection du rouge (croisement)

Pour détecter un red stop, on se sert d'abord du channel **bleu** (figure 7) de la caméra. Ce dernier nous permet de repérer la ligne rouge à la manière d'une ligne noire comme on peut le voir sur la figure. Nous avons donc les pixels de début et de fin de la ligne.
On réalise ensuite une moyenne de l'intensité **rouge** (figure 8) uniquement sur les pixels de la ligne ainsi qu'une moyenne sur l'ensemble des pixels.
Si on arrive à repérer une ligne avec le channel **bleu**, **ET** que la moyenne de cette ligne est plus grande que la moyenne générale pour le **rouge**, cela signifie que l'on se situe sur une **ligne** et qu'elle est **rouge**.

3.3 Suivi de la ligne noire : utilisation des moteurs

Un régulateur P est utilisé afin de s'assurer que le robot suive correctement la ligne, tel qu'on l'a vu en TP4.

3.4 Détection et contournement d'obstacles

Afin de détecter un obstacle frontalement sur le chemin de l'e-puck2, le Time of Flight est utilisé. Puis, pour le contourner, ce sont les capteurs infrarouges qui sont utilisés. Le contournement se fait dans un sens anti-horaire, et donc les capteurs IR2 et IR3 sont utilisés (respectivement à la nomenclature sur l'interface en verre de l'e-puck2).

Après avoir détecté l'obstacle avec le ToF, l'e-puck2 effectue une rotation de 180° afin de s'approcher de l'obstacle **en reculant** (et être suffisament proche pour les détecteurs IR) sans que le miroir ne percute l'obstacle. Il tourne ensuite de 90° afin de débuter le contournement de l'obstacle. Le robot déclenche un mode "*obstacle_around* pour une distance détectée du ToF

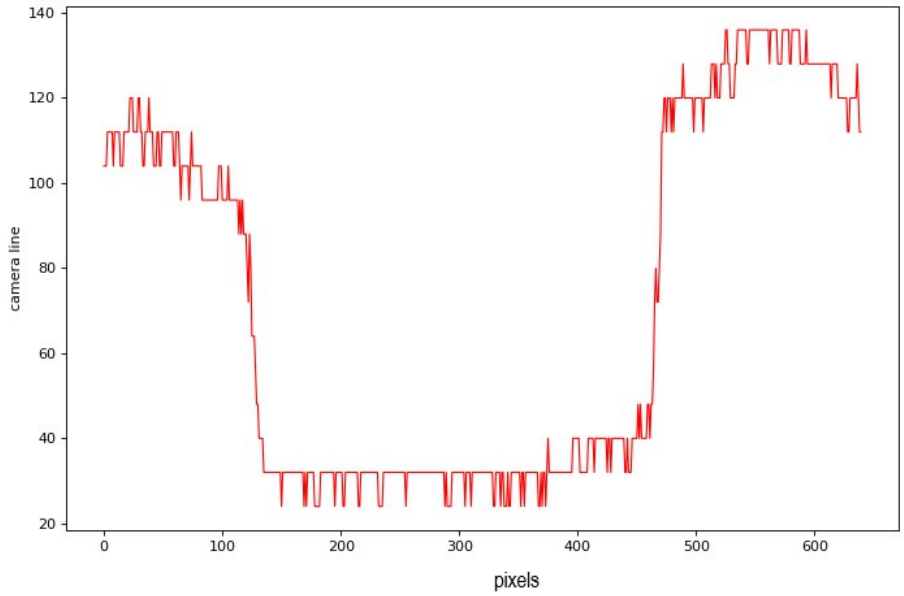


Figure 7 – Détection du rouge, vue du channel bleu

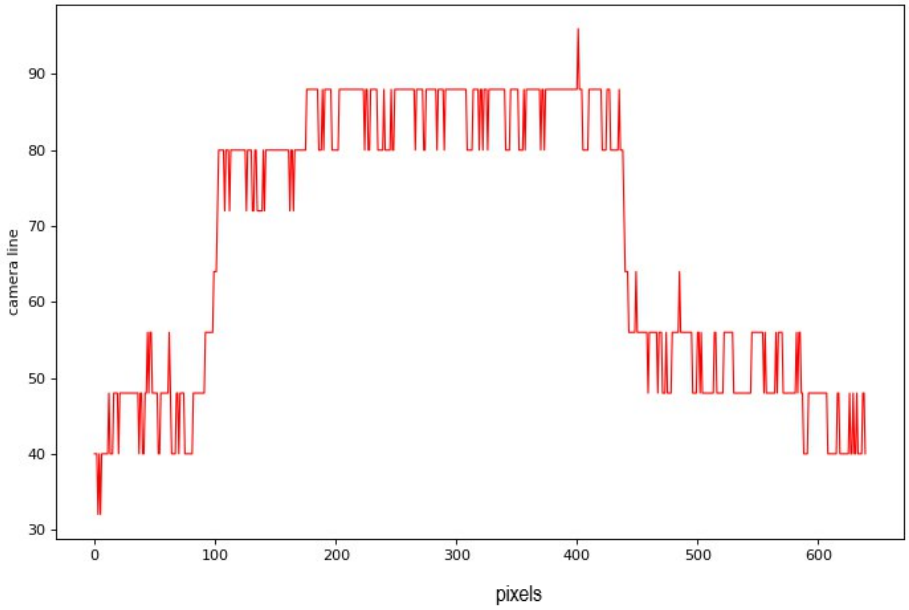


Figure 8 – Détection du rouge, vue du channel rouge

inférieure à 7 cm (caractérisé par un #define OBSTACLE_DIST dans le code).

Pour respecter une certaine distance de sécurité lors du contournement de l'obstacle, un régulateur PID est utilisé. Après avoir commencé par utiliser un régulateur PI, nous avons remarqué que le robot contournait avec bien trop de vibration et des overshoots très grands. L'introduction d'un dérivateur a permis de considérablement le réduire. Les différentes valeurs ont été tunés itérativement afin d'obtenir le régulateur optimal.

L'utilisation des deux capteurs infrarouge est primordiale, en effet, un seul ne permettrait pas de contourner des objets qui ne sont pas ronds car le régulateur a pour effet de le faire trop tourner et donc heurter l'obstacle.

L'implémentation du PID a été faite en utilisant nos notions vues au cours de "Automatique et commande numérique" du BA5 de Microtechnique. [Référence, 1] La commande numérique

dérivative est donnée par :

$$u_d[k] = \frac{T_f u_d[k-1] + K_d(e[k] - e[k-1])}{T_s + T_f}$$

4 Structure du code

4.1 Modules hiérarchiquement haut

4.1.1 module path_regulator

Ce module fondamental dirige le suivi de la ligne du robot. Il implémente une FSM dans le cas du positionnement d'un obstacle (cf. fig. 9). Ce module appelle également les fonctions liées à l'orientation et des directions que le robot doit prendre (cf. sec. 4.2.2).

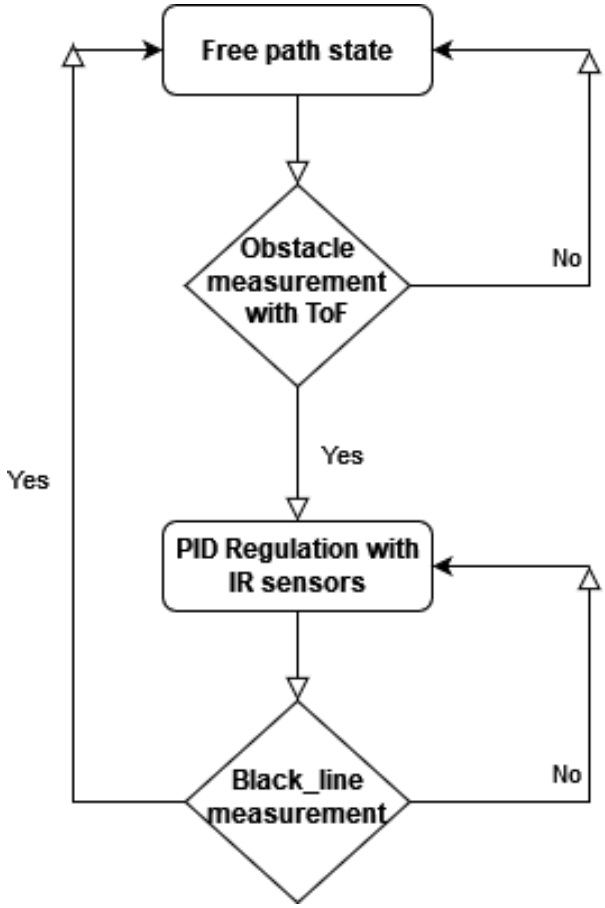


Figure 9 – Ce schéma illustre la FSM (Finite-State Machine) implémentée dans le module path_regulator

Nous avons ainsi deux états qui sont accédés via un enum : le $free\_path$, et le *obstacle* state. Le deuxième s'enclenche dès que le ToF détecte un objet, et change à $free\_path$ après contournement de l'objet et que le robot redédecte la ligne noire, respectivement à ce qui a été décrit en section 3.2.1.

4.1.2 module lecture

Ce module est essentiel dans la gestion du code purement software de notre projet. C'est celui qui permet d'appeler l'algorithme de Dijkstra, selon un noeud de départ donné, et qui en déduit le schéma le plus court pour un noeud d'arrivée. Un paramètre booléen donné en argument de la fonction principale permet de déterminer si l'algorithme est calculé pour l'aller ou pour le retour.
Des fonctions "getters" sont présentes afin de respecter le principe d'encapsulation.

4.2 Modules hiérarchiquement bas

4.2.1 module process_image

Ce module contient les fonctions qui traitent l'image envoyée par la caméra au processeur. Ce dernier est donc responsable de tous les aspects de détection de lignes, de couleurs ainsi que le changement des états des variables permettant la gestion de la direction et des moteurs du robot à un plus haut niveau.

4.2.2 module crossroad

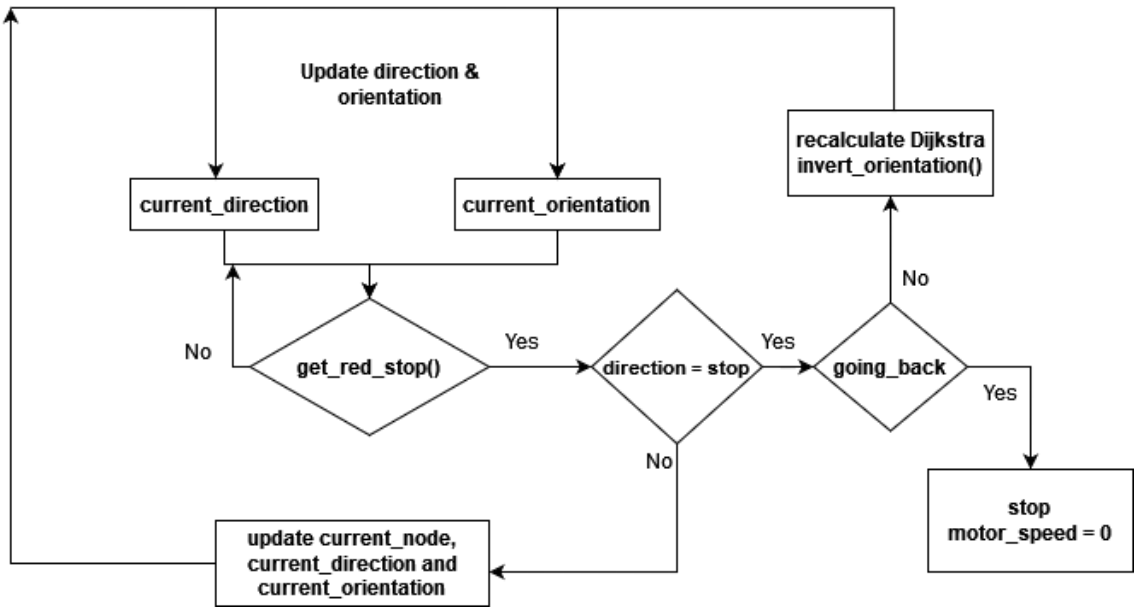


Figure 10 – Ce schéma illustre la FSM (Finite-State Machine) implémentée dans le module crossroad

Dans le module crossroad, une FSM (figure 10) a été implémentée au travers d'un switch-case qui, selon l'orientation actuelle du robot et la valeur du prochain noeud, prend la décision de soit tourner à droite, gauche, ou continuer tout droit.
Nous avons ainsi 3 états qui importent sur la prise de décision du robot : le noeud actuel, l'orientation actuelle, et la direction actuelle.
A mentionner l'usage d'une variable booléenne *going_back*. Celle-ci détermine, si le robot effectue le chemin aller ou retour.

4.2.3 module Noeud

Ce module contient la déclaration d'un Noeud (avec ses paramètres *uid*, *tab_liens*, *tab_liens_dist*, ainsi que ceux utilisés pour l'algorithme de Dijkstra.
Il contient également l'implémentation de ce dernier, avec toutes les fonctions qu'il utilise (telles que *find_min_access*, *sort_queue*, etc.). Ce module est donc essentiellement un module de bas niveau et qui sert au module lecture afin de donner des informations de base au robot (chemin à parcourir, distance, etc.)

4.2.4 module pid_regulator

Ce module implémente le régulateur PID utilisé pour le contournement de l'obstacle (cf. sec. 3.4).

4.2.5 module City_intialization

Ce module est le module le plus bas dans ce qui s'adresse aux noeuds et à l'algorithme de Dijkstra. Il initialise tous les noeuds de la ville avec leur paramètres associés (UID, liens, distance, etc.).
Ils sont placés dans un tableau de pointeurs sur Noeud appelé *node_list*. L'usage de pointeurs est essentiel dans ce module ainsi que dans noeud.c. En effet, cela nous permet tout d'abord

de modifier directement les paramètres associés à chacun des noeuds de la ville. De plus, cette pratique s'apparente à un "passage par référence" (par opposition à un passage par copie), ce qui permet d'éviter les multiples copies du tableau *node_list* lorsque sont appelées les différentes fonctions utilisés par Dijkstra, ce qui résulterait en un temps de consommation non-optimal.
Ce module déclare les noeuds de départ et d'arrivée comme constantes "define". Ainsi, si l'on veut faire des essais avec des noeuds d'arrivée différents, il faut redémarrer le programme.

4.2.6 module motors_custom

Ce module contient les différentes fonctions de base s'opérant sur le moteur de l'e-puck2. (avancer d'une certaine distance, rotation, etc.)

4.3 Organisation des threads

Notre projet fonctionne avec les threads suivants :

- Les threads de haute priorité et qui sont responsables du traitement d'image ProcessImage et CaptureImage ayant une priorité de NORMALPRIO+1
- Le thread PathRegulator qui s'occupe de la mise à jour de l'état des moteurs à NORMALPRIO.
- Le thread LedToggle qui s'occupe d'allumer et d'éteindre les LEDs qui est aussi à NORMALPRIO, ces deux derniers threads ne rentrent pas en conflit pour notre application car le premier est à une fréquence de environ 10 ms alors que le deuxième est à 300 ms. De plus ce thread sur les LEDs consomme très peu de ressources, son execution est très rapide et ne perturbe pas significativement le déroulement du thread PathRegulator.
- Le thread responsable de la mesure des capteurs IR utilisées pour le contournement d'obstacles IR2 et IR3, qui est à NORMALPRIO
- Le thread responsable de la détection d'obstacles avec le ToF qui est NORMALPRIO+10

La gestion de la priorité des threads n'a pas été d'une grande difficulté car nous avons trois threads principaux, dont deux responsables de la capture de l'image utilisée par PathRegulator pour piloter le robot, d'où une priorité plus basse pour ce dernier. L'ajout des autres threads n'ayant pas impacté le bon fonctionnement du projet de par leur faible fréquence d'appel et leur execution rapide, nous n'avons pas eu à ajouter d'autres niveaux de priorités.
Nous avons du augmenter la working area du thread s'occupant de l'image car nous traitons les informations des deux channels rouge et bleu.

5 Limitations

5.1 Caméra

La caméra étant un périphérique dépendant grandement de l'environnement, il a été très difficile d'effectuer les ajustements pour le suivi de ligne et surtout pour le repérage de la ligne rouge. Bien que théoriquement notre manière de repérer une ligne rouge est correcte, il a fallu ajouter un coefficient par lequel est multipliée la moyenne générale. Ceci est dû à l'intensité qui diminue vers les pixels situés au bord de l'image comme on peut le voir sur les différentes figures. Ainsi, la constante COEFF_MEAN_RED est extrêmement dépendant de l'environnement.

De plus le robot ne peut pas suivre la ligne si :

- L'angle qu'il fait avec la ligne est plus de $\pm 55°$
- Il est décentré de ± 3.5 cm par rapport à la ligne
- La luminosité ambiante n'est pas suffisante

5.2 Contournement d'obstacles

Quand le robot croise un obstacle, il a besoin d'un espace équivalent à la taille de l'obstacle plus 12cm, pour effectuer ce contournement. De plus, pendant que ce dernier contourne un

obstacle, il ne doit pas rencontrer une ligne noire autre que celle par laquelle il a commencé. Etant donné que la taille de notre ville est limitée par la taille d'une feuille A0, nous avons dû trouver un compromis entre le nombre de noeuds la constituant (16) et le nombre de liens dans lesquels il pouvait y avoir un contournement d'obstacles (4).

5.3 Capteurs IR

Afin de garder une distance consistante avec les différents obstacles à chaque contournement, la couleur des obstacles doit être blanche du fait de la nature des capteurs infrarouge, ces derniers ne doivent pas être trop "discontinus" pour permettre aux roues de suivre.
De plus en fonction de l'environnement les valeurs mesurées par le capteur peuvent être grandement différentes ce qui fait qu'il faut ajuster les valeurs du régulateur PID.

5.4 Interaction avec utilisateur

Dans l'idéal, il aurait été souhaitable d'avoir une interface utilisateur par laquelle l'utilisateur décide du noeud de départ, d'arrivée, vitesse du robot, etc. Le manque de temps ne nous l'a pas permis.
Nous avons donc tout déclaré dans le fichier city_initialization.h dans lequel tous les noeuds, leurs liens ainsi que le noeud de départ et d'arrivée sont déclarés.

6 Conclusion

Ce mini-projet aura été une expérience extrêmement enrichissante. Nous avons réussi à satisfaire les objectifs que nous nous étions fixés au début. A savoir, créer un code permettant au robot de se déplacer dans une ville dessinée par nous-même.

D'un point de vue organisationnel, excepté durant la semaine de Pâques, nous avons presque tout du long travaillé et écrit le code ensemble. Cette manière d'opérer nous a convenu, et Github nous était utile uniquement comme stockage de données. Nous sommes ainsi conscient qu'une telle méthode de travail ne serait pas possible ni efficace dans le domaine professionnel, où une interface telle que Github deviendrait un outil très puissant pour collaborer à distance.

Ce mini-projet offre la possibilité, avec plus de temps, d'aller plus profondément dans le code et d'implémenter d'autres fonctionnalité : par exemple, une interface utilisateur lui permettant de sélectionner le noeud de départ/arrivée. Le fait d'avoir écrit un code permettant au robot de s'adapter à tous types de ville élargit le champ de possibilités.

7 References

1 Cours de "Automatique et commande numérique" de Prof. Alireza Karimi, enseigné durant le semestre d'automne 2021.

2 Travail pratique n°4 du cours

3 Projet de programmation orienté objet du BA2