Compte rendu Wifibot
Robotique Mobile

Audin Lucas Almamy, Kobrossly Rayan

4A IA2R TD8

Introduction

Ce TP de robotique mobile porte sur la prise en main et le contrôle du Wifibot, un robot à quatre roues équipé d'une Raspberry Pi. L'objectif était de développer une application graphique (GUI) en C++ capable de piloter le robot à distance via le réseau Wifi (protocole TCP/IP).

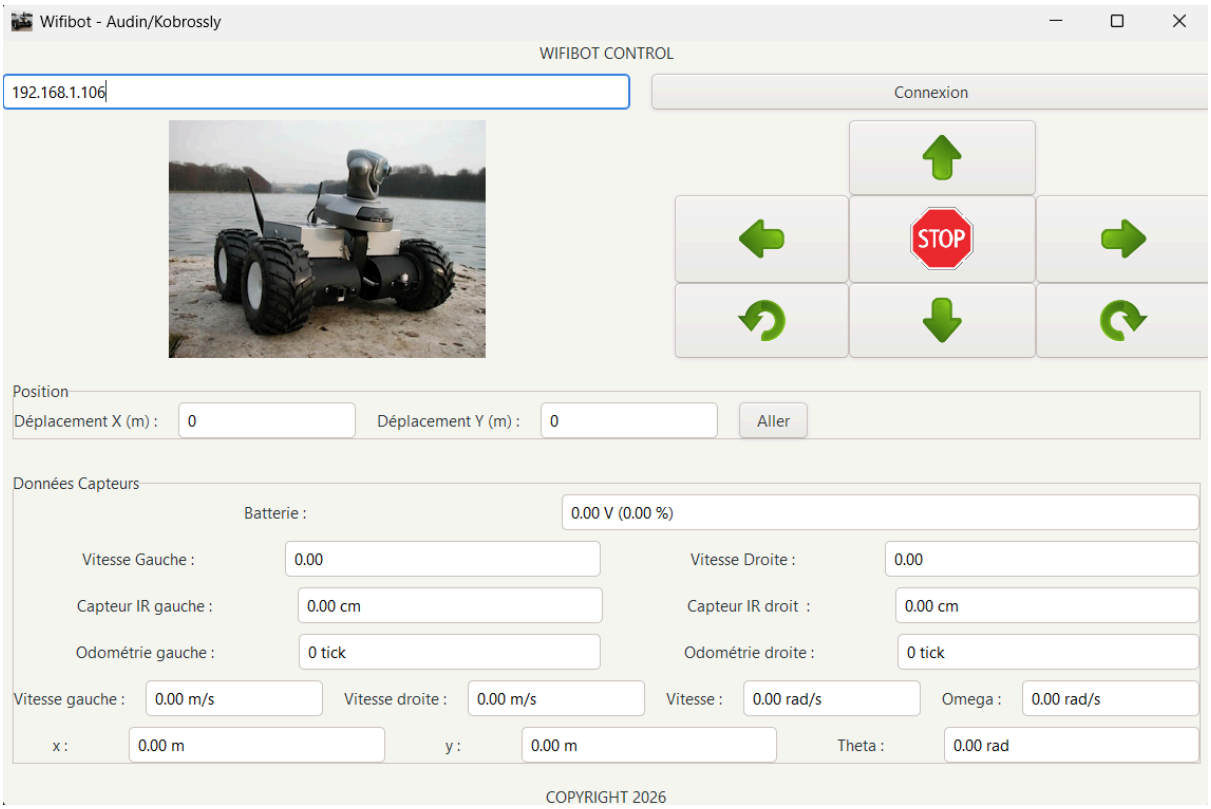
Le travail ne s'est pas limité à envoyer des commandes de vitesse, comme nous avions fait en 3A, il a fallu utiliser un protocole de communication pour recevoir les données de ses capteurs proprioceptifs et exteroceptifs comme le niveau de batterie, les codeurs incrémentaux ou les capteurs infrarouges. Nous avons également mis en œuvre une solution d'odométrie, permettant au robot de calculer sa propre position (x, y) en temps réel grâce aux codeurs de ses roues, afin de réaliser des déplacements précis.

Présentation générale

Pour organiser le code du projet, nous avons opté pour une approche modulaire qui sépare bien les différentes responsabilités. La pièce maîtresse est la classe Wifibot (dans wifibot.cpp), qui agit véritablement comme le cerveau du robot : c'est elle qui centralise les données, calcule la position par odométrie, assure le déplacement du robot et gère les boucles d'envoi et de réception. Plutôt que de gérer la connexion réseau directement dedans, nous avons délégué toute la partie TCP/IP (protocole Ethernet) bas niveau au fichier socket.cpp, ce qui permet de garder le code métier propre et lisible. La carte wifi du wifibot assure la liaison sans fils au système.

La gestion des commandes moteurs passe par la classe Order, qui sert à encapsuler les consignes de vitesse gauche et droite pour éviter les erreurs de manipulation directes sur les valeurs brutes. Le point technique le plus important reste l'utilisation des threads. En effet, pour éviter que l'interface graphique (gui.cpp) ne se fige à chaque échange réseau, nous avons mis en place des processus parallèles pour l'émission et la réception des trames. Cela garantit que l'application reste fluide et réactive pour l'utilisateur, tout en traitant les données de télémétrie en arrière-plan.

Voici l’interface graphique actuelle (qui utilise la librairie gtkmm) qui contient un espace pour saisir l’adresse IP du Wifibot et s’y connecter, une sorte de ‘manette’ pour contrôler manuellement le robot, une boîte qui sert à déplacer le robot vers une position précise relative à sa position initiale lors de l’ouverture de connection et finalement une section pleine d’informations sur les capteurs infrarouge, la batterie, l’odométrie, position et vitesse.



Réceptions des données

(01) Informations transmises par le robot

```
bufsend[0]=(-speedlab);
bufsend[1]=(-speedlab >> 8);//speed is a short and it is tics /50ms or 10ms
bufsend[2]=(unsigned char)(tmpadc2 >> 2);//Bat Volt:10.1V 1.28V 404/4->101
bufsend[3]=(unsigned char)(tmpadc4 >> 2);//3.3v->255 2v-> 624/4 -> 156
bufsend[4]=(unsigned char)(tmpadc3 >> 2);//3.3v->255 2v-> 624/4 -> 156
bufsend[5]=bufposition[0];//Acumulated odometrie is a float
bufsend[6]=bufposition[1];//12ppr x 4 x 51 gear box = 2448 tics/wheel turn
bufsend[7]=bufposition[2];
bufsend[8]=bufposition[3];
bufsend[9]=(speedlab2);
bufsend[10]=(speedlab2 >> 8);
bufsend[11]=(unsigned char)(tmpadc0 >> 2);
bufsend[12]=(unsigned char)(tmpadc1 >> 2);
bufsend[13]=bufposition2[0];
bufsend[14]=bufposition2[1];
bufsend[15]=bufposition2[2];
bufsend[16]=bufposition2[3];
bufsend[17]=0;//robot current // *0.194-37.5 = I in Amp / * 10 for the GUI
: 10 -> 1 A (ACS712 30Amp chip)
bufsend[18]=14; //firmware version
bufsend[19]=crc16 low;
bufsend[20]=crc16 high;
```

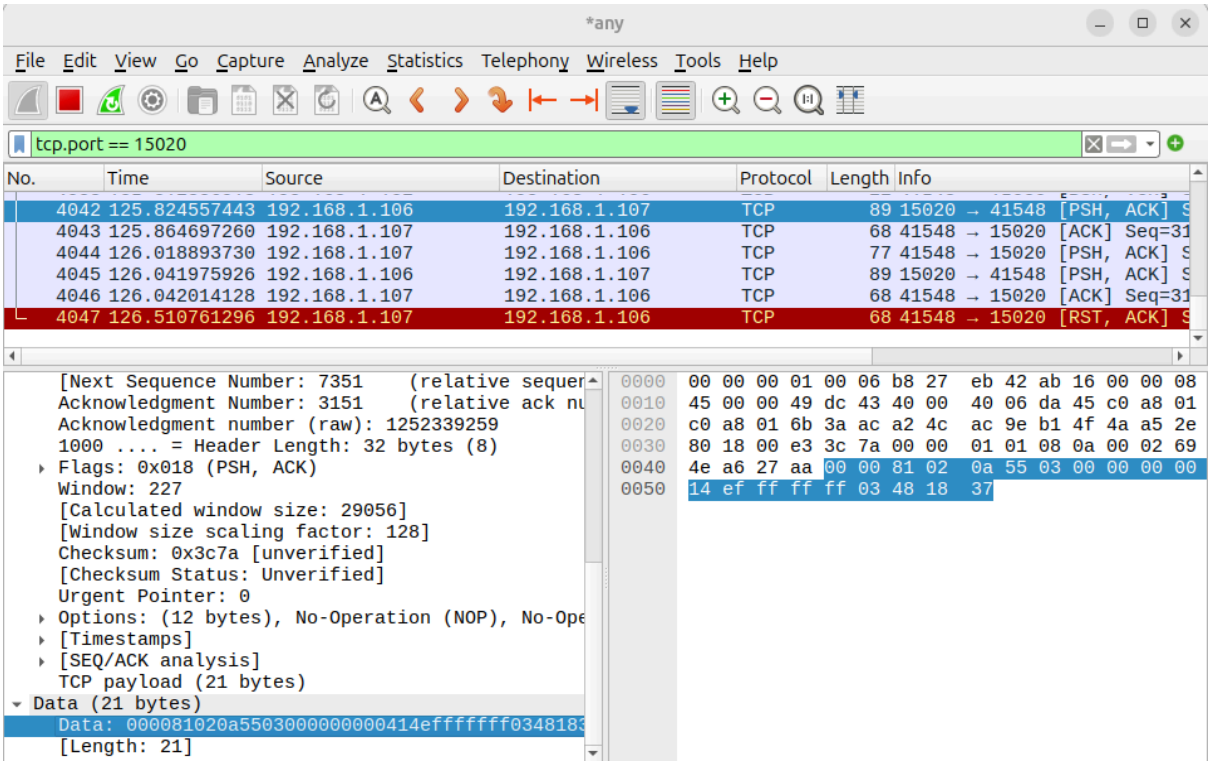
D'après l'analyse du protocole de communication définie dans le document *Protocoles_Wifi*, le robot transmet périodiquement une trame de 21 octets contenant les informations suivantes:

- ➔ **Vitesse des moteurs (Gauche et Droite) :** Codée sur 2 octets pour chaque côté. (Octets 0 1 et 9 10)
- ➔ **Niveau de la batterie :** 1 octet (Octet 2).
- ➔ **Capteurs Infrarouges (IR) :** 4 valeurs analogiques (converties en numérique sur 1 octet chacune en utilisan un convertisseur analogique numérique).

  Dans notre Wifibot actuel que deux des quatres capteurs sont implémentés et donc après quelques essaies, nous avons remarqué que le capteur infrarouge gauche correspond à l'octet 3 tandis que le capteur droit correspond à l'octet 11. (les octets 4 et 12 seront inutiles et le câble physiquement connecté au vide joue le rôle d'une mini antenne)

- ➔ **Odométrie (Codeurs) :** 4 octets pour la roue gauche et 4 octets pour la roue droite, indiquant le nombre de tics comptabilisés depuis le démarrage. La valeur du codeur est donc codée sur 64 bits.
- ➔ **Valeur du courant et version firmware :** ces deux octets 17 et 18 ne seront pas utiles pour notre manipulation du Wifibot.
- ➔ **CRC :** Les deux derniers octets servent de vérification CRC 16-bits.

Le logiciel Wireshark sera utilisé pour visualiser les différents paquets envoyées et reçus par le Wifibot. Le Wifibot d'adresse IP *192.168.1.106* envoie les trames à l'ordinateur connecté, d'adresse *192.168.1.107* et vice versa.



Au repos, le robot n'avance pas et donc les octets 0,1,9 et 10 indiquent *0x00*. Le niveau de batterie est envoyé comme *0x80* ou 128 en décimal, ce qui correspond à 12.8V (Le robot est à charge pleine), les capteurs IR donne *0x02* et *0x14* une valeur binaire. Cette valeur correspond à une tension et dans les parties qui suivent, nous montrerons comment transformer celle-ci en une distance. La valeurs des codeurs incrémenteux correspondent aux valeurs que le Wifibot mesure et donc par la suite nous manipulerons ces valeurs pour avoir des valeurs plus concrètes (remises à 0 dès que la connection est établie).
Pour bien visualiser qui correspond à quoi, nous mettons le robot en mouvement et nous observons les différentes valeurs des octets de la trame changer (vitesse, valeur des codeurs, capteurs infrarouges)

Pour plus de précision, des outils en ligne permettent de comparer les valeurs des 21 octets afin de visualiser les modifications.

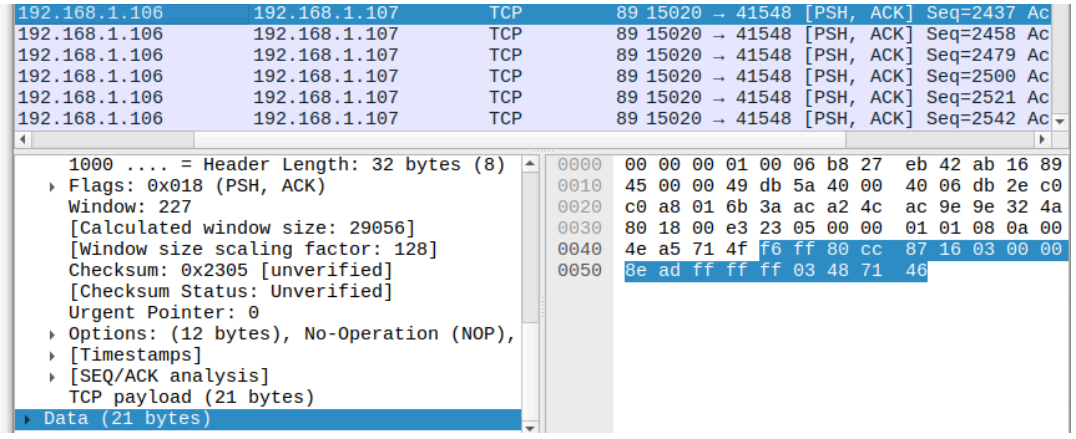
```
1 00 00 81 02 0a 55 03 00 00 00 00 04 14 ef ff ff ff 03 48 18 37
```

```
1 f6 ff 80 cc 87 16 03 00 00 f6 ff c6 8e ad ff ff ff 03 48 71 46
2
```

Nous remarquons encore sur wireshark, les trames envoyées depuis l'ordinateur vers le wifibot, c'est les 9 trames de données envoyées par la méthode *run_send()*.

(02) Thread de réception

La classe Wifibot intègre un mécanisme de multithreading pour éviter de bloquer l'interface graphique. Le thread d'envoi de données a été abordé l'année dernière. La méthode *connect()* lance un thread dédié via *m_p_thread_receive* qui exécute la boucle *run_receive()*. Cette boucle lit le port série (TCP 15020) en continu *(m_socket.receive)* et ne traite les données que si le CRC16 est valide, sinon le paquet est ignoré, garantissant ainsi l'intégrité et la validité des données de la trame reçue.



Nous recevons le tableau char* contenant les 21 octets. Nous n'enregistrons que les octets utiles à notre manipulation.
Les octets 0 et 1 seront combinés avec l'opérateur ou logique '|' (le bit 1 décalé vers la gauche) pour avoir un short de 2 octets. Procédure identique pour les bits 9 et 10.

Les bits 5,6,7,8 seront combinés ensemble pour avoir une variable de 4 octets. (De même pour 13,14,15,16.

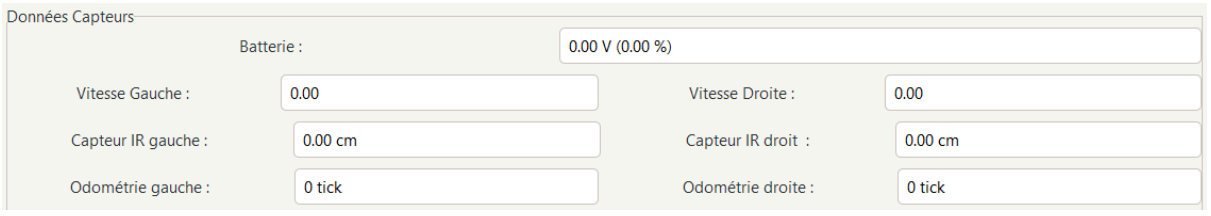
Veuillez consulter le code de la méthode run_receive() pour plus d'informations. Les conversions des valeurs binaires en données manipulables seront expliquées dans les parties des capteurs proprioceptifs et exteroceptifs.

(03) Visualisation de la batterie

Avant que le wifibot envoie son paquet, il décale d'un bit vers la droite et donc il divise la valeur en décimal par 2, si le max était *0xFF* pour 255, il devient *0x80* pour 128. Depuis la doc du wifibot, la tension à charge pleine du robot est de 12.8V. Il suffit donc de diviser par 10 la valeur reçue par le socket. La valeur de la tension et le pourcentage de la batterie seront stockés sous forme d'un nombre flottant.

Pour l'interface graphique (*gui.cpp*), un signal timeout (*timer*) déclenche périodiquement la mise à jour de l'affichage (*Widget Gtk::Entry ou Labe*l) avec cette valeur stockée dans l'objet Wifibot.
*Glib::signal_timeout().connect(sigc::mem_fun(*this, &Gui::time_out), 400);* permet de mettre la méthode à jour toutes les 400ms.
Toutes les mises à jour des éléments (capteurs, positions…) seront dans cette méthode *timeout*. La méthode *std::tostring()* transforme le flottant en texte mais laisse par défaut 6 places décimales. Nous définissons alors une méthode *format_nb()* qui elle utilise *std::stringstream* pour saisir une précision de 2 chiffres après la virgule.

Une fois la valeur de la batterie affichée, nous pourrons réaliser des manipulations sur les autres octets reçus.



Capteurs extéroceptifs

(04, 05, 06) Nature du capteur

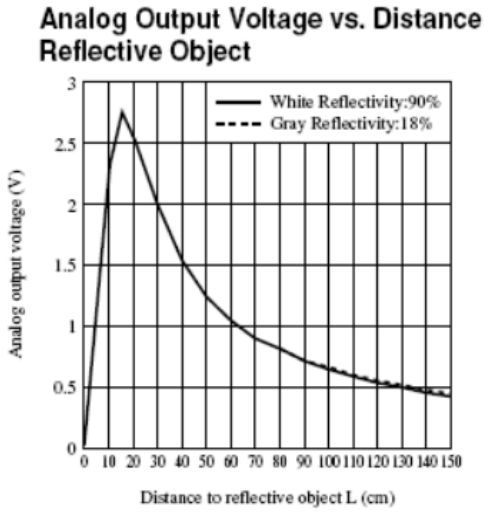
Le robot utilise des capteurs de distance optiques infrarouges (IR). La datasheet identifie le modèle comme étant le Sharp GP2Y0A02YK. D'après la documentation, la plage de détection efficace est de 20 cm à 150 cm. Le signal de sortie est une tension analogique qui varie de manière non-linéaire (fonction puissance quand la distance est supérieure à 20 cm) en fonction de la distance (inversement proportionnelle). La courbe caractéristique montre une tension élevée pour les objets proches (~2.5V à 20cm) et faible pour les objets lointains (~0.4V à 150cm). Les distances plus petites que 20 cm ne seront pas prises en compte.

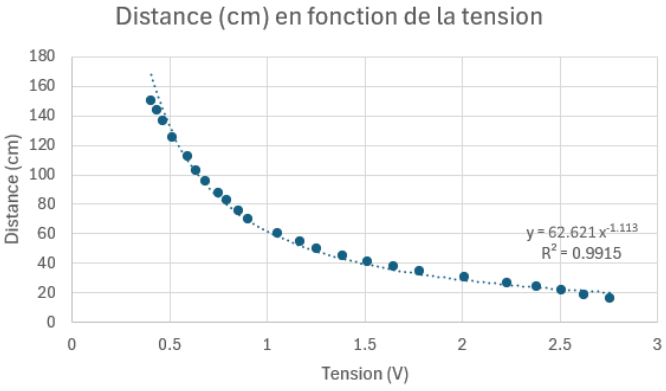
(07) Information reçue et conversion en distance

La trame du robot contient la valeur brute de la conversion Analogique-Numérique (ADC) sur 1 octet (valeur entre 0 et 255). 255 représente une tension maximale de 3.3V et 0 représente donc 0V. La tension est donc obtenu par une simple règle de 3 *(float)(buf[3] / 255.0f) * 3.3f;* (pour le capteur gauche par exemple). Cette valeur de tension est stockée en flottant et sera manipuler par la suite.

(08) Équation de la distance et implémentation

La documentation technique nous donne la courbe de variation de tension en fonction de la distance :





Après avoir utilisé PlotDigitizer et exporté les données vers Excel, nous inversons les axes x et y. Nous visualisons alors la variation de distance en fonction de la tension. Nous pouvons utiliser une courbe de tendance pour avoir la formule exacte.

L'équation utilisée dans le code pour linéariser la réponse du capteur et obtenir la distance en cm est :

$$\text{Distance} = 62.621 \times \text{Tension}^{-1.113}$$

Cette formule est issue d'une régression (power regression). Le coefficient de détermination est égal à 99.15% et donc le modèle (puissance) explique 99% des variations de la distance.

*position_g = 62.621*pow(tension_un,-1.113);*

La méthode de récupération, ou plutôt les lignes de code, sont implémentées dans run_receive. À chaque réception de trame :

1. On récupère les octets du capteur 3 et 11.
2. On le convertit en tension (règle de 3).
3. On applique la formule de puissance ci-dessus pour obtenir les distances gauches et droites *position_g* et *position_d*

Ces distances sont en centimètre, et si plus tard une localisation avec balises ou un filtre de Kalman seront implémentés, il faudra la convertir en mètre.

Capteurs Proprioceptifs

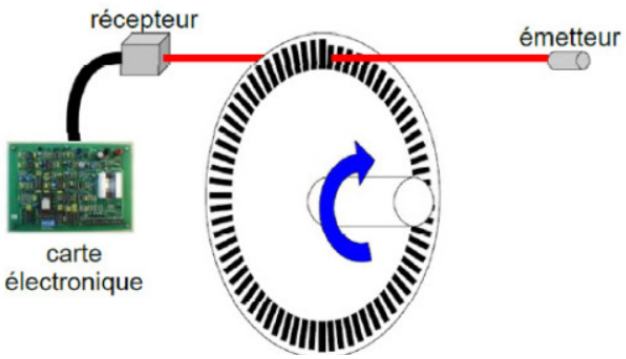
(11) Type de capteur et résolution

Les capteurs proprioceptifs utilisés pour mesurer le déplacement sont des codeurs incrémentaux à effet Hall montés sur les moteurs.

La résolution indiquée est de 336 tics par tour de roue. Cette valeur sera donc implémentée dans notre code sous forme de constante *TICK_IN_REVOLUTION*.

Un codeur incrémental est un capteur électromécanique qui convertit un mouvement de rotation en une suite d'impulsions électriques. En comptant ces impulsions, il est possible de déterminer la variation de position, la vitesse de rotation et le sens de rotation d'un axe, sans fournir de position absolue.

Voici un schéma représentatif d'un codeur incrémental :



(12) Informations utiles pour la position

Pour déterminer la position (x, y) et l'orientation theta θ du robot, nous avons besoin de la différence de tics mesurée entre l'instant t et t-1 pour la roue gauche et la roue droite. Cela permet de calculer la distance parcourue par chaque roue et l'orientation globale du Wifibot. Dans la partie d'odométrie qui suit, les vitesses linéaires et angulaire seront calculées ainsi que les positions du robot à l'aide des valeurs des codeurs incrémentaux..

(13) Dimensions du robot

Le diamètre de la roue a été mesuré pour avoir 14 cm. Cette valeur sera définie dans le fichier d'en-tête *wifibot.cpp* comme une constante *DIAMETRE_ROUE = 0.14* (Implémentation en mètres).

La distance parcourue en un tour de roue est donc son périmètre $P = \pi * D \approx 0.44$ *mètres*. Cette information sera utilisée par la suite.

La distance entre les deux roues 2*L est mesurée encore (pour l'odométrie par la suite). Cette distance est égale à 0.33m. On définit donc L comme égal à 0.165m.

3.4 Odométrie

(14) Modèle cinématique

En modèle continu (analog), voici les différentes équations à implémenter :

$$\begin{cases} v_d(t) = \dot{\varphi}_d(t) r \\ v_g(t) = \dot{\varphi}_g(t) r \end{cases} \qquad \begin{cases} \theta(t) = \int_0^t \omega(t)\,dt \\ x(t) = \int_0^t v(t)\cos\theta(t)\,dt \\ y(t) = \int_0^t v(t)\sin\theta(t)\,dt \end{cases}$$

$$\begin{cases} v(t) = \dfrac{v_d(t) + v_g(t)}{2} \\ \omega(t) = \dfrac{v_d(t) - v_g(t)}{2L} \end{cases}$$

Pour calculer les positions x et y, il faudra avant calculer l'angle Θ, les vitesses linéaire et angulaire et les vitesses gauche et droite.
Dans le monde digital, une discrétisation sera implémentée. Le Wifibot est un robot à roues différentielles. Son modèle cinématique se décrit ainsi comme suivant :

$$\begin{cases} v_k^d = \dfrac{\varphi_k^d - \varphi_{k-1}^d}{\Delta t} r \\ v_k^g = \dfrac{\varphi_k^g - \varphi_{k-1}^g}{\Delta t} r \end{cases} \quad \begin{cases} v_k = \dfrac{v_k^d + v_k^g}{2} \\ \omega_k = \dfrac{v_k^d - v_k^g}{2L} \end{cases} \quad \begin{cases} \theta_k = \theta_{k-1} + \omega_k \Delta t \\ x_k = x_{k-1} + v_k \cos(\theta_k) \Delta t \\ y_k = y_{k-1} + v_k \sin(\theta_k) \Delta t \end{cases}$$

À chaque itération k, les vitesses gauches et droites sont calculées à partir de l'angle de rotation de chaque roue (en radian). Puis la vitesse globale est calculée (moyenne des deux vitesses) et la vitesse angulaire. Pour avoir calculé les positions x et y à l'itération k, il faudra juste avant calculer ϑ_k. Vu le modèle différentiel, il faudra garder en mémoire les valeurs des variables calculées à l'itération k-1. D'où le but de développer un algorithme en C++.

Pour calculer l'angle de rotation de chaque roue, nous utilisons la règle de 3 suivante :

$$\phi_d = \frac{2 \cdot \pi \cdot odo_d}{TICK\_IN\_REVOLUTION}$$

où la constante est le nombre de tics total par rotation (336). odo_d représente la valeur du codeur incrémental mesuré. Au début de la connexion socket, les premières valeurs des codeurs reçues sont sauvegardées sous une variable initiale (*init_odo_g* et *init_odo_d*) et donc chaque valeur mesurée par la suite sera soustraite de cette valeur. Quand le robot avance, cette valeur ne cesse d'augmenter (analogie à une position), d'où nous implémentons la dérivée de la fonction pour trouver la vitesse.

(15) Algorithme de calcul de position (x,y)

L'algorithme, implémenté dans *update_position()* suit la logique suivante :

Pour suivre la position du robot en temps réel, la méthode commence par lire les valeurs des codeurs sur les roues gauche et droite. On récupère le nombre de tics et on le convertit en radians, sachant qu'un tour complet correspond à 336 incréments.

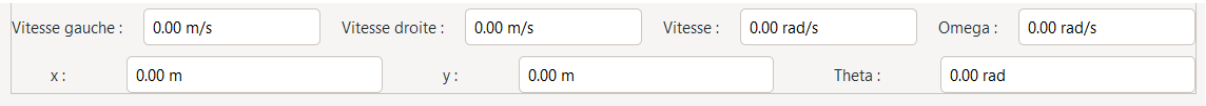
Ensuite, on calcule la vitesse de chaque roue en regardant de combien elle a tourné depuis la dernière boucle. En multipliant cette variation par le rayon de la roue et en divisant par le temps écoulé, on obtient la vitesse linéaire instantanée de chaque côté.

La vitesse d'avancement est simplement la moyenne des vitesses des deux roues, tandis que sa vitesse de rotation dépend de la différence entre elles, rapportée à la largeur du châssis.Enfin, pour trouver la nouvelle position (x, y) et l'orientation ϑ, on procède par intégration (méthode d'Euler). On ajoute le petit déplacement calculé à

la position précédente, en projetant la distance parcourue selon l'angle actuel. Une dernière étape permet de recadrer l'angle entre -π et π pour rester cohérent.

Par contre, durant les manipulations sur le Wifibot, nous avons remarqué que l'angle θ est toujours connu à +-0.1 rad. D'où les imprécisions dans la méthode de *deplacement_global*.

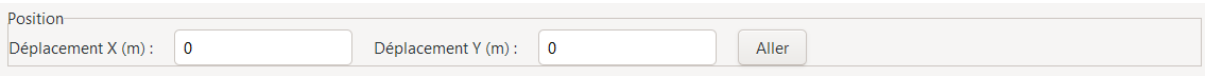
Avant de sortir de la boucle, la variable *odo_prev* devient égale à la *odo* actuelle, ainsi nous avançons sur une itération k+1.



(16) & (17) Déplacement contrôlé

Le code implémente une méthode *go_to(dx, dy)* et *deplacement* qui asservit le robot pour atteindre une cible. *go_to* vérifie qu'aucun thread de déplacement n'est déjà déployé, met le drapeau *en_mouvement* à vrai et crée un thread associé à la méthode déplacement (tout en passant en entrée du thread la position cible). Si un thread de déplacement existe mais le robot n'est pas en mouvement (la position est déjà atteinte), il faudra le supprimer. On vérifie que le thread peut être joint (*if (m_p_thread_goto->joinable()) m_p_thread_goto->join();*) avant de supprimer pour éviter un ressource leak, sinon le programme plante (*std::terminate*).

De plus le boolean en mouvement est enregistré comme attribut *std::atomic<bool> en_mouvement;* , ceci évite que si les deux threads accèdent à la mémoire exactement au même moment, il y a chance que la valeur du boolean soit corrompue. Vu que *updatePostion* et *deplacement* peuvent accéder aux coordonnées x,y,θ au même moment exact, dans une autre version de notre code, nous pourrions utiliser *std::atomic<float>* pour x, y et θ.



L'odométrie permet au robot de s'arrêter lorsque la distance calculée correspond à la consigne (ex: 5 mètres). La précision de cette méthode dépend fortement de l'absence de glissement des roues (patinage), car l'odométrie est une méthode d'estime qui accumule les erreurs au cours du temps.

En deuxième temps, nous pourrons se déplacer en x et en y et donc on oriente le robot suivant l'angle de *arctan(y/x)* et on avance droit vers la position cible. L'erreur accumulée est plus grande (rotation et mouvement) qu'un mouvement linéaire. Les deux méthodes *deplacement* et *deplacement_global* utilisent le principe d'overshoot mais l'implémentation est un peu différente : la première utilise des booleens *position_atteinte* pour sortir de la boucle tandis que l'autre calcule la distance

restante et ne sort de la boucle que quand la distance restante atteint son minimum (et sort de la boucle quand elle réaugmente).

Conclusion

En conclusion, ce TP nous a permis de valider les acquis de 4A en robotique mobile et en programmation C++. Contrairement au projet précédent réalisé en 3A, nous avons ici mis en place une architecture complète capable non seulement d'envoyer des ordres, mais surtout de traiter les retours du robot en temps réel grâce à la gestion des threads. Nous avons appris encore que supprimer un thread nécessite une attention, sinon C++ plante le programme. La mise en œuvre de l'odométrie a été l'étape la plus délicate. Si notre méthode *updatePosition* nous permet bien d'estimer la position (x,y) et d'atteindre une cible via la fonction *go_to*, nous avons pu observer concrètement le problème de la dérive. Comme l'odométrie est une méthode d'estime, le moindre patinage des roues s'accumule et crée un décalage entre la position calculée et la réalité. Pour aller plus loin et avoir un robot parfaitement autonome, il faudrait coupler cette estimation avec d'autres capteurs (capteurs infrarouges) et réaliser un filtre de Kalman pour corriger l'erreur au cours du temps.