Projet
conception d'un robot mobile

Professeur : Valentin Gies

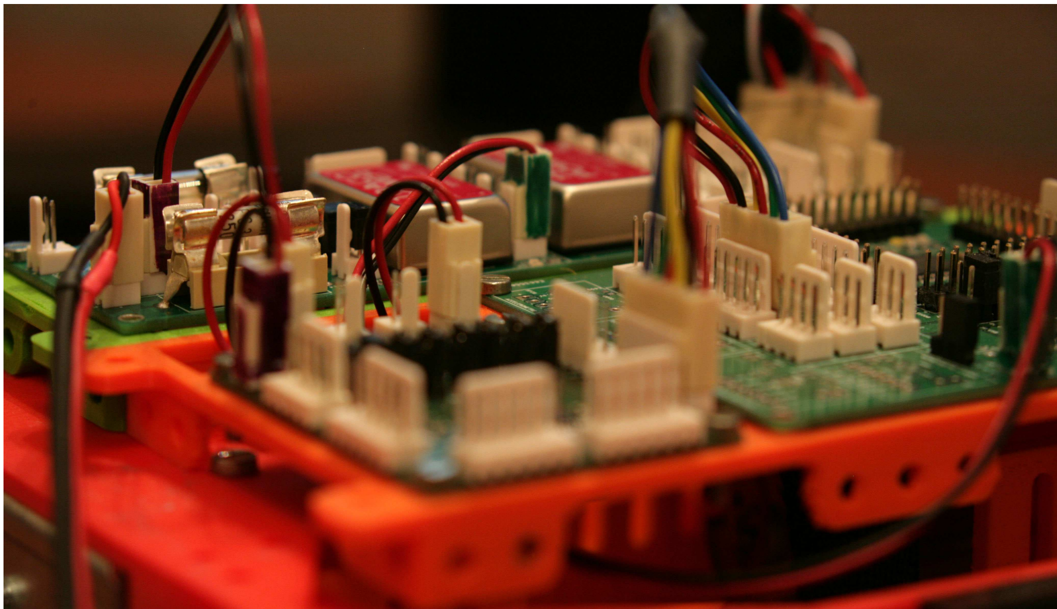

Table des matières

1 A la découverte de la navigation par odométrie **2**
1.1 Mise en oeuvre du module QEI 2
1.2 Détermination des déplacements du robot et supervision 2
1.3 Asservissement polaire en vitesse du robot 4

2 Gestion des tâches - pilotage avancé **8**
2.1 Un exemple de tâche : l'asservissement en position du robot 8

1 A la découverte de la navigation par odométrie

Dans cette partie, vous allez apprendre à utiliser les modules *QEI* de gestion des encodeurs optiques à quadrature de phase. Ce encodeurs optiques sont constitués de roues codeuses perforées de trous.

Le but de cette partie est de :
- Mettre en oeuvre le module *QEI* dans un premier temps.
- Déterminer les déplacements du robot et les afficher sur une carte en temps réel.
- Mettre en oeuvre un asservissement de type *P*, *PI* ou *PID*.

1.1 Mise en oeuvre du module QEI

Le microcontrôleur utilisé possède deux modules QEI capables de gérer les signaux des encodeurs optiques à quadrature de phase placés sur des roues indépendantes. Leur role est de compter ou de décompter les impulsions du codeurs selon que la roue avance ou recule. Cette fonction basique pourrait être réalisée en mode interruption sur le microcontrôleur mais le flux étant très important, le taux d'occupation du processeur serait élevé. L'utilisation d'un module hardware permet donc de décharger le processeur de cette tache. Pour cela il faut configurer le module et les pins remappables correspondantes.

⟹ Ajouter au code de paramétrage des pins remappables situé dans *"IO.c"* le code suivant :

```
    //******************** QEI *****************
    _QEA2R = 97;     //assign QEI A to pin RP97
    _QEB2R = 96;     //assign QEI B to pin RP96

    _QEA1R = 70;     //assign QEI A to pin RP70
    _QEB1R = 69;     //assign QEI B to pin RP69
```

⟹ Ajouter dans un fichier *"QEI.c"* les fonctions suivantes permettant d'initialiser les deux modules QEI. Ajouter le header correspondant et appeler ces fonctions depuis le main avant la boucle infinie.

```
void InitQEI1()
{
    QEI1IOCbits.SWPAB = 1; //QEAx and QEBx are swapped
    QEI1GECL = 0xFFFF;
    QEI1GECH = 0xFFFF;
    QEI1CONbits.QEIEN = 1; // Enable QEI Module
}

void InitQEI2(){
    QEI2IOCbits.SWPAB = 1; //QEAx and QEBx are not swapped
    QEI2GECL = 0xFFFF;
    QEI2GECH = 0xFFFF;
    QEI2CONbits.QEIEN = 1; // Enable QEI Module
}
```

1.2 Détermination des déplacements du robot et supervision

A ce stade, les modules QEI sont actifs en tache de fond, mais les valeurs du QEI ne sont pas encore récupérées par le robot.

⟹ Ajouter à *"QEI.c"* une fonction *QEIUpdateData* permettant de récupérer les données issues du QEI. Cette fonction sera appelée à intervalle régulier, par exemple à 250 Hz sur l'interruption du timer 1. Elle contiendra le code suivant :

```
#define DISTROUES 281.2
void QEIUpdateData()
{
    //On sauvegarde les anciennes valeurs
```

```
    QeiDroitPosition_T_1 = QeiDroitPosition;
    QeiGauchePosition_T_1 = QeiGauchePosition;

    //On éractualise les valeurs des positions
    long QEI1RawValue = POS1CNTL;
    QEI1RawValue += ((long)POS1HLD<<16);

    long QEI2RawValue = POS2CNTL;
    QEI2RawValue += ((long)POS2HLD<<16);

    //Conversion en mm (éérgl pour la taille des roues codeuses)
    QeiDroitPosition = 0.01620*QEI1RawValue;
    QeiGauchePosition = -0.01620*QEI2RawValue;

    //Calcul des deltas de position
    delta_d = QeiDroitPosition - QeiDroitPosition_T_1;
    delta_g = QeiGauchePosition - QeiGauchePosition_T_1;
    //delta_theta = atan((delta_d - delta_g) / DISTROUES);
    delta_theta = (delta_d - delta_g) / DISTROUES;
    dx = (delta_d + delta_g) / 2;

    //Calcul des vitesses
    //attention à remultiplier par la éfrquence dé'chantillonnage
    robotState.vitesseDroitFromOdometry = delta_d*FREQ_ECH_QEI;
    robotState.vitesseGaucheFromOdometry = delta_g*FREQ_ECH_QEI;
    robotState.vitesseLineaireFromOdometry =
        (robotState.vitesseDroitFromOdometry + robotState.vitesseGaucheFromOdometry)/2;
    robotState.vitesseAngulaireFromOdometry = delta_theta*FREQ_ECH_QEI;

    //Mise à jour du positionnement terrain à t-1
    robotState.xPosFromOdometry_1 = robotState.xPosFromOdometry;
    robotState.yPosFromOdometry_1 = robotState.yPosFromOdometry;
    robotState.angleRadianFromOdometry_1 = robotState.angleRadianFromOdometry;

    //Calcul des positions dans le referentiel du terrain
    robotState.xPosFromOdometry = ...
    robotState.yPosFromOdometry = ...
    robotState.angleRadianFromOdometry = ...
    if(robotState.angleRadianFromOdometry > PI)
        robotState.angleRadianFromOdometry -= 2*PI;
    if(robotState.angleRadianFromOdometry < -PI)
        robotState.angleRadianFromOdometry += 2*PI;
}
```

$\Longrightarrow$ Expliquer dans la détail ce que fait le code précédent et complétez les parties manquantes signalées par des ... Pensez à justifier la présence des multiplications par la fréquence d'échantillonnage dans le calcul des vitesses de déplacement. Il vous faudra également ajouter à la structure robotState les grandeurs utilisées dans le code, elle sont de type *double*.
A présent, votre code est capable de calculer en temps réel la position du robot relative à sa position de départ.

Il reste à transmettre ces informations à l'interface de supervision.

$\Longrightarrow$ Implanter dans le fichier "QEI.c" une fonction permettant de réaliser la transmission des données de positionnement avec leur horodatage. Elle ressemblera au code suivant, en remplaçant les ... par le code qui convient. Pour la transmission de la vitesse linéaire du robot et de la vitesse angulaire, veiller à les multiplier au préalable par 1000 de manière à permettre une transmission des décimales. La valeur reçue sera divisée par 1000 dans l'interface en C# qui vous est fournie.

```
#define POSITION_DATA 0x0061
void SendPositionData() {
    unsigned char positionPayload[24];
```

```
    unsigned int i;
    for (i = 0; i < 4; i++) {
        positionPayload[3 - i] = (unsigned char) (timestamp>>(8 * i));
        positionPayload[7 - i] = (unsigned char) ... ; //Transmission de xPosFromOdometry
        positionPayload[11 - i] = (unsigned char) ... ; //Transmission de yPosFromOdometry
        positionPayload[15 - i] = (unsigned char) (((long) (RadianToDegree(robotState.angleRadianF
        positionPayload[19 - i] = (unsigned char) ... ; //Transmission de vitesseLineaireFromOdom
        positionPayload[23 - i] = (unsigned char) ... ; //Transmission de vitesseAngulaireFromOdon
    }
    Uart1EncodeAndSendMessage(POSITION_DATA, 24, positionPayload);
}
```

⟹ Appeler la fonction *SendPositionData* depuis l'interruption du Timer 1, en veillant à **sous-échantillonner les envois** de manière à ne pas envoyer plus de *10* messages par seconde pour éviter de saturer la liaison UART. Si tout se passe bien, vous devriez voir apparaitre les informations de positionnement sur l'interface du robot. Valider avec le professeur le bon fonctionnement de l'ensemble.

1.3 Asservissement polaire en vitesse du robot

Dans cette partie vous allez asservir votre robot en vitesse. Cet asservissement peut être réalisé roue par roue avec un correcteur de type PI ou PID, de manière à assurer une erreur statique nulle en vitesse comme présenté à la figure 1.

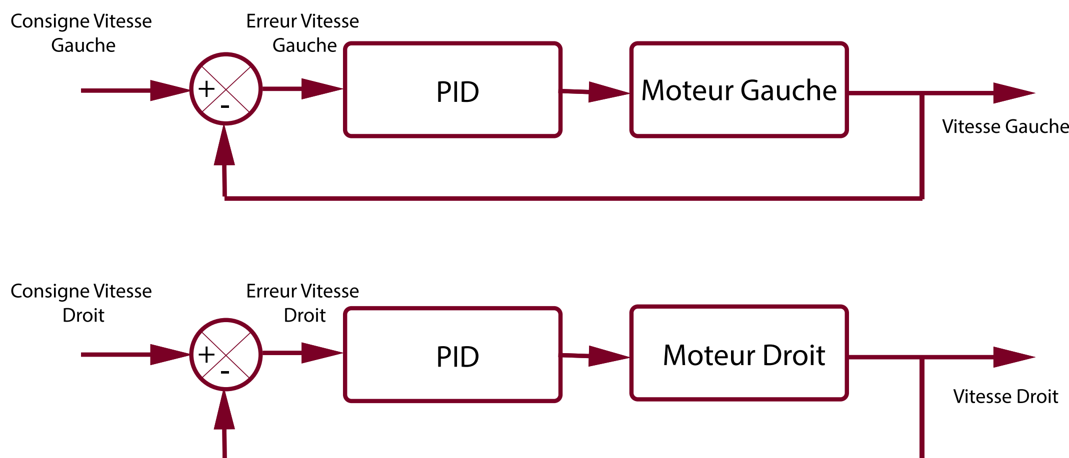

Figure 1 – Asservissement PID classique

Ce type d'asservissement est performant car il permet d'assurer que les deux roues tourneront à la même vitesse si on leur donne une consigne identique, le robot roulera donc en ligne droite en régime permanent, ce qui n'est pas le cas sans asservissement. Toutefois, l'inconvénient de cette méthode est d'asservir les moteurs indépendamment l'un de l'autre : si un moteur démarre plus lentement que l'autre en raison de frottements ou d'une charge mal répartie, alors il va prendre du retard avant d'arriver à sa vitesse de consigne. Ce retard se traduira par une erreur sur le cap du robot.
Cette erreur de cap devra être compensée par une contre-réaction sur les vitesses de chacun des moteurs. Ce

modèle n'est pas optimal : il serait préférable de contrôler la vitesse angulaire qui donne le cap par intégration directe, ainsi que la vitesse linéaire du robot plutôt que de contrôler directement les vitesses de ses deux moteurs. C'est l'objet de l'**asservissement polaire** que vous allez mettre en oeuvre.
Les équations de couplage entre les vitesses des moteurs et les vitesses angulaires et linéaires du robot sont les

suivantes :

$$V_{lineaire} = \frac{V_{Droit} + V_{Gauche}}{2} \quad (mm.s^{-1})$$
$$V_{angulaire} = \frac{V_{Droit} - V_{Gauche}}{D_{Roues}} \quad (rad.s^{-1})$$

Si on inverse ces équations, on obtient :

$$V_{Gauche} = V_{lineaire} - V_{angulaire}\frac{D_{Roues}}{2} \quad (mm.s^{-1})$$
$$V_{Droit} = V_{lineaire} + V_{angulaire}\frac{D_{Roues}}{2} \quad (mm.s^{-1})$$

On peut donc en déduire le schéma synoptique de l'asservissement polaire de la figure 2.

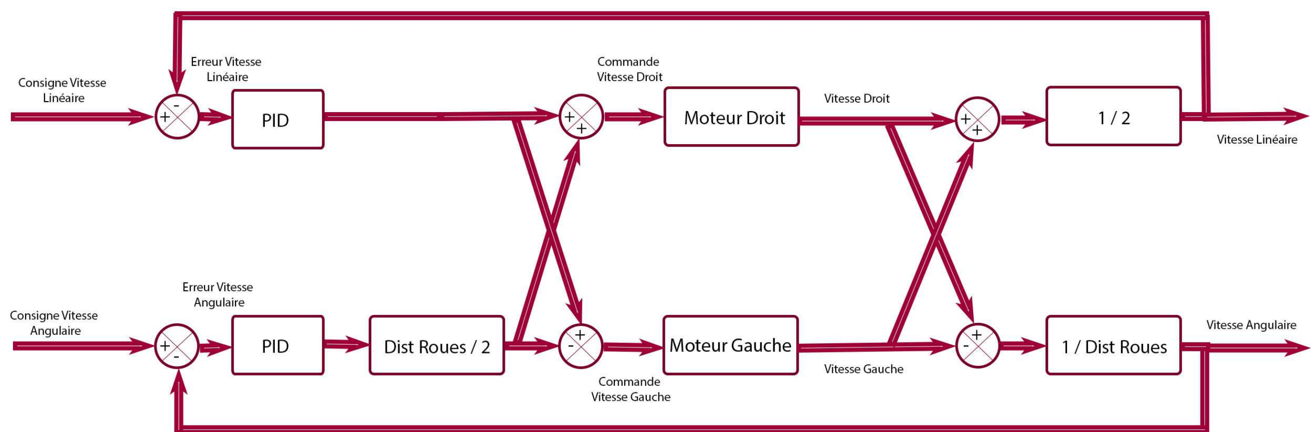

FIGURE 2 – Asservissement polaire en vitesse

⟹ Ecrire une fonction *PWMSetSpeedConsignePolaire* permettant de générer les commandes des moteurs droits et gauche dans cet asservissement polaire en complétant le code à trou ci-dessous. Dans un premier temps, remplacer les deux correcteurs PID par deux correcteurs P, ce qui revient à multiplier l'erreur par une constante de proportionnalité.

```
#define COEFF_VITESSE_LINEAIRE_PERCENT 1/25.
#define COEFF_VITESSE_ANGULAIRE_PERCENT 1/50.
void PWMSetSpeedConsignePolaire(){
    //Correction Angulaire
    double erreurVitesseAngulaire = ...
    double sortieCorrecteurAngulaire = ...
    double correctionVitesseAngulaire = ...
    double correctionVitesseAngulairePourcent =
            correctionVitesseAngulaire * COEFF\_VITESSE\_ANGULAIRE\_PERCENT;

    //Correction éLinaire
    double erreurVitesseLineaire = ...
    double sortieCorrecteurLineaire = ...
    double correctionVitesseLineaire = ...
```

```
    double correctionVitesseLineairePourcent =
                correctionVitesseLineaire * COEFF\_VITESSE\_LINEAIRE\_PERCENT;

    //ééGnration des consignes droite et gauche
    robotState.vitesseDroiteConsigne = correctionVitesseLineairePourcent
                + correctionVitesseAngulairePourcent ;
    robotState.vitesseDroiteConsigne = LimitToInterval(
                robotState.vitesseDroiteConsigne, −100, 100);
    robotState.vitesseGaucheConsigne = correctionVitesseLineairePourcent
                − correctionVitesseAngulairePourcent;
    robotState.vitesseGaucheConsigne = LimitToInterval(
                robotState.vitesseGaucheConsigne, −100, 100);
}
```

⟹ Appeler cette fonction à la place de PWMSetSpeedConsigne() dans l'interruption du *Timer 1*.

Il est à présent temps de régler l'asservissement polaire du robot. Dans un premier temps, nous allons régler l'asservissement en vitesse angulaire avant de régler l'asservissement de vitesse linéaire. Pour cela, durant la phase de réglage de l'asservissement angulaire, il vous faudra mettre la correction de vitesse linéaire à 0 à la main dans la fonction PWMSetSpeedConsignePolaire, donner une consigne de vitesse de rotation (par exemple 3 $rad.s^{-1}$), puis augmenter progressivement le gain du correcteur angulaire jusqu'à atteindre l'oscillation du moteur. Celle-ci se traduit par un bruit de vibration et une forte augmentation du courant absorbé sur l'alimentation stabilisée.

⟹ Réaliser la manipulation présentée précédemment et déterminer le gain proportionnel limite d'oscillation. Valider avec le professeur.

Il reste à présent à remplacer le gain proportionnel par un correcteur PID bien réglé. Les formules d'implantation du PID dans le domaine de Laplace et en temps discret sont rappelées ci-dessous :

$$H(s) = K_p + K_i/s + K_d s = K_p(1 + \frac{1}{T_i s} + T_d s)$$

Ce qui donne avec l'approximation d'Euler :

$$H(z) = \frac{K_p(1 - z^{-1}) + K_i T_e + \frac{K_d}{T_e}(1 - z^{-1})^2}{1 - z{-1}}$$

Soit :

$$S(z)(1 - z^{-1}) = E(z)\left((K_p + K_i T_e + \frac{K_d}{T_e}) + (-K_p - 2\frac{K_d}{T_e})z^{-1} + \frac{K_d}{T_e}z^{-2}\right)$$

$$S_n = S_{n-1} + E_n(K_p + K_i T_e + \frac{K_d}{T_e}) + E_{n-1}(-K_p - 2\frac{K_d}{T_e}) + E_{n-2}(\frac{K_d}{T_e})$$

Le réglage du PID sera effectué par la méthode de Ziegler-Nichols en boucle fermée. C'est une méthode empirique qui consiste à rechercher dans un premier temps le pompage limite, c'est à dire le gain proportionnel limite K_u conduisant à l'oscillation, ce que vous avez fait précédemment. Leur période est T_u et doit être mesurée, mais on prendra 0.05s en première approximation. Une fois K_u et T_u obtenues, il est possible de mettre en oeuvre un correcteur, P, PI ou PID en réglant les gains de la manière suivante :

	P	PI	PID
K_p	$0.5K_u$	$0.45K_u$	$0.6K_u$
T_i	-	$0.83T_u$	$0.5T_u$
T_d	-	-	$0.125T_u$

⟹ Ajouter un fichier *Asservissement.c* (pensez à créer le header correspondant !) à votre projet dans lequel vous mettrez le code à trous suivant en le complétant avec les équations précédentes :

```
#include "Asservissement.h"
#include "timer.h"

double Te = 1 / FREQ_ECH_QEI;

//************************* CORRECTEUR VITESSE ANGULAIRE ***************************
double eAng0, eAng1, eAng2; //valeurs de l'éentre du correcteur gauche à t, t-1, t-2
double sAng0, sAng1, sAng2; //valeurs de la sortie du correcteur gauche à t, t-1, t-2

void SetUpPiAsservissementVitesseAngulaire(double Ku, double Tu)
{
    //éRglage de Ziegler Nichols sans édpassements : un tout petit peu mou
    double Kp = ...
    double Ti = ...
    double Td = 0;
    double Ki = Kp / Ti ;
    double Kd = Kp * Td;

    alphaAng = ...
    betaAng = ...
    deltaAng = ...
}

double CorrecteurVitesseAngulaire(double e)
{
    eAng2 = eAng1;
    eAng1 = eAng0;
    eAng0 = e;
    sAng1 = sAng0;
    sAng0 = sAng1 + eAng0 * alphaAng + eAng1 * betaAng + eAng2*deltaAng;
    return sAng0;
}
```

⟹ Appelez la fonction *SetUpPiAsservissementVitesseAngulaire* à l'initialisation du *main* en expliquant son rôle. Pensez à inclure les header.

⟹ Dans le code de la fonction *PWMSetSpeedConsignePolaire*, remplacer dans le calcul de la correction de vitesse angulaire la proportionnalité à l'erreur par la sortie de la fonction correcteur *CorrecteurVitesseAngulaire*.

⟹ Tester votre code. A ce stade, votre robot doit être asservi en vitesse de rotation. Vérifier que c'est bien le cas, en freinant un des rouleaux du support robot. Si c'est bien le cas, vous avez terminé la partie asservissement en vitesse angulaire.

⟹ Il vous faut à présent réaliser l'asservissement en vitesse linéaire de votre robot. Pour cela, régler à 0 la consigne de vitesse angulaire asservie précédemment. Régler également la consigne de vitesse liénaire à 300 $mm.s^{-1}$. Réintégrer une correction de type proportionnel sur la vitesse linéaire dans *PWM.c*, trouver le gain limite et implanter comme précédemment un asservissement de type PI sur la vitesse angulaire.

⟹ Faites varier la consigne de vitesse linéaire et tester. Votre moteur doit notament pouvoir fonctionner à des très aibles vitesses de rotations si l'asservissement fonctionne bien. Valider l'ensemble avec le professeur.

⟹ Pour terminer cette partie sur l'asservissement de vitesse polaire. Reimplantez la machine à état de fonctionnement du robot en mode autonome pour l'évitement d'obstacles en pilotant votre robot grâce à l'asservissement polaire en vitesse.

2 Gestion des tâches - pilotage avancé

Un robot doit pouvoir effectuer des tâches très variables. Nous avons vu qu'il pouvait être amener à se déplacer de manière autonome en évitant des obstacles, mais il peut aussi être amener à effectuer une succession d'actions prédéterminées formant des tâches. Il est également important que celles-ci puissent être séquentialisées, interrompues ou parallèlisées.

Le but de cette partie est de découvrir les rudiments de la gestion de tâches, qui s'apparente à un *Operating System* temps réel (RTOS) très simplifié, en particulier au niveau des méthodes de synchronisation.

Afin d'explorer ces concepts, nous allons implanter un *Task Manager* qui aura pour rôle de synchroniser les tâches du robot. Ces tâches seront les suivantes :
— Attente pendant 5 secondes.
— Déplacement en mode autonome comme déjà vu.
— Déplacement vers un point donné(asservissement de position)
— Déplacement successif sur les quatres coins d'un carré dans le mode précédent.

2.1 Un exemple de tâche : l'asservissement en position du robot

$\Longrightarrow$ Vous avez à présent terminé la mise en oeuvre de l'odométrie, de la reconstruction de trajectoire et de l'asservissement de votre robot. Vous pouvez à présent imaginer un asservissement en position de votre robot, qui vous permettra d'effectuer des déplacements extrêmement précis.