

Projet conception d'un robot mobile

Professeur : Valentin GIES

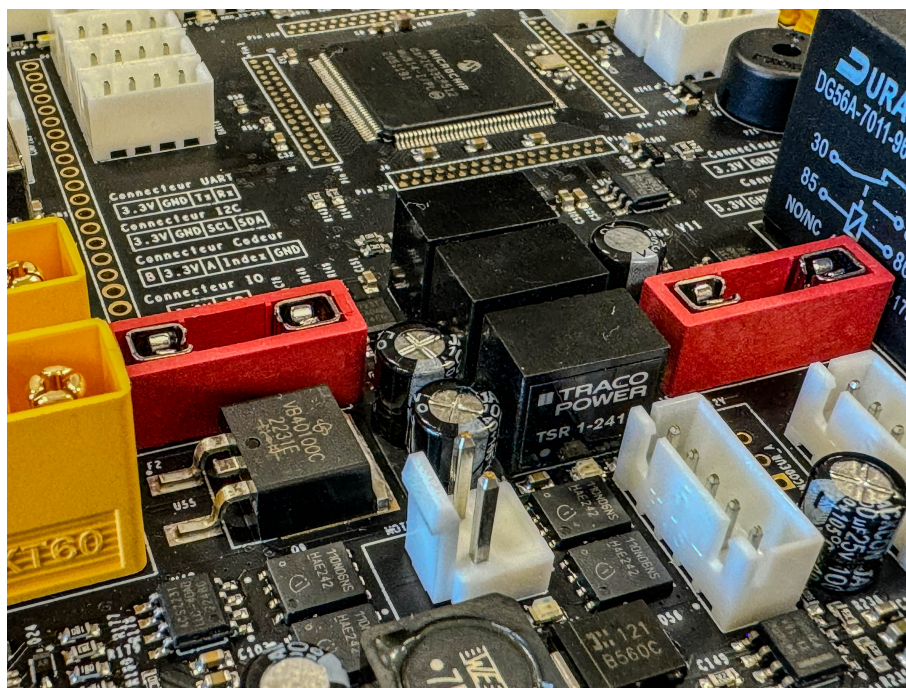


Table des matières

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	Affichage en C# des données de positionnement et vitesse	4
1.4	Affichage en C# des données de vitesse et d'asservissement	5
2	Asservissement en vitesse du robot	6
2.1	Un petit peu de théorie	6
2.2	Implantation de la commande polaire en vitesse	8
2.3	Implantation de l'asservissement polaire	9
2.4	Réglage de l'asservissement polaire	11
3	Asservissement en position avec générateur de trajectoire	12
3.1	Génération du déplacement du <i>ghost</i>	12
3.2	Asservissement du robot réel sur le <i>ghost</i>	14

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 rôle 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 tâche. 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()
{
    QEI1IOCbts.SWPAB = 1; //QEAx and QEBx are swapped
    QEI1GECL = 0xFFFF;
    QEI1GECH = 0xFFFF;
    QEI1CONbits.QEIEN = 1; // Enable QEI Module
}

void InitQEI2(){
    QEI2IOCbts.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 tâche 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 DISTRQUES 0.2812
void QEIUpdateData()
```

```

{
    //On sauvegarde les anciennes valeurs
    QeiDroitPosition_T_1 = QeiDroitPosition;
    QeiGauchePosition_T_1 = QeiGauchePosition;

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

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

    //Conversion en mm (regle pour la taille des roues codeuses)
    QeiDroitPosition = 0.00001620*QEI1RawValue;
    QeiGauchePosition = -0.00001620*QEI2RawValue;

    //Calcul des deltas de position
    delta_d = QeiDroitPosition - QeiDroitPosition_T_1;
    delta_g = QeiGauchePosition - QeiGauchePosition_T_1;

    //Calcul des vitesses
    //attention a remultiplier par la frequence d echantillonnage
    robotState.vitesseDroitFromOdometry = delta_d*FREQ_ECH_QEI;
    robotState.vitesseGaucheFromOdometry = delta_g*FREQ_ECH_QEI;
    robotState.vitesseLineaireFromOdometry = ...
    robotState.vitesseAngulaireFromOdometry = ...

    //Mise a jour du positionnement terrain a 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;
}

```

⇒ 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.

⇒ Implanter dans le fichier "QEI.c" une fonction permettant de réaliser la transmission des données de positionnement et de vitesse avec leur horodatage. Il est important de noter ici que **la transmission va nécessiter l'envoi de données de type float**. Vous allez donc apprendre à les transmettre de manière efficace : un float est en réalité codé sur 32 bits, soit 4 octets. Il est possible de transférer ces 4 octets de manière directe sous forme d'un tableau de 4 octets converti à partir du float. Pour cela, on utilisera la fonction *getBytesFromFloat* disponible dans la bibliothèque *Utilities* en C, que vous téléchargerez depuis la page : <http://www.vgies.com/projet-robot/>. Pensez à inclure les deux fichiers *Utilities.h* et *Utilities.c* dans le projet.

Une fois intégré ces fichiers, le code d'envoi des trames ressemblera au code suivant. Vous noterez qu'une fonction de transmission des *long*, entiers codés sur 32 bits a également été ajoutée.

```
#define POSITION_DATA 0x0061
void SendPositionData()
{
    unsigned char positionPayload[24];
    getBytesFromInt32(positionPayload, 0, timestamp);
    getBytesFromFloat(positionPayload, 4, (float)(robotState.xPosFromOdometry));
    getBytesFromFloat(positionPayload, 8, (float)(robotState.yPosFromOdometry));
    getBytesFromFloat(positionPayload, 12, (float)(robotState.angleRadianFromOdometry));
    getBytesFromFloat(positionPayload, 16, (float)(robotState.vitesseLineaireFromOdometry));
    getBytesFromFloat(positionPayload, 20, (float)(robotState.vitesseAngulaireFromOdometry));
    UartEncodeAndSendMessage(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 apparaître les informations de positionnement sur l'interface du robot. Valider avec le professeur le bon fonctionnement de l'ensemble.

La fonction inverse de reconstitution du float à partir du tableau d'octet reçu en C# est quant à elle disponible dans la librairie *BitConverter* intégrée à Visual Studio.

Pour reconstruire un float à partir d'un tableau de byte en C#, vous utiliserez un code inspiré du suivant, le second paramètre de la fonction *ToSingle* est la position de départ de la donnée à récupérer dans la payload reçue :

```
robot.positionX0do = BitConverter.ToSingle(msgPayload, 4);
robot.positionY0do = BitConverter.ToSingle(msgPayload, 8);
```

⇒ Gérer la réception de la trame *POSITION_DATA* en C# en validant que les *float* soient correctement reconstruits.

1.3 Affichage en C# des données de positionnement et vitesse

Afin de simplifier le débogage des fonctions de déplacement, il est nécessaire de voir en temps réel les données de vitesse et de position issues de vos robots. Pour cela nous allons utiliser des composants C# fournis permettant d'afficher des oscilloscopes et une carte du monde sur laquelle le robot peut se déplacer.

⇒ Téléchargez le composant WPF *WpfOscilloscope* à la page du projet robot : [Télécharger WpfOscilloscope](#).

⇒ Ajoutez ce composant à votre solution en tant que projet existant.

⇒ Dans votre projet WPF qui accueillera l'oscilloscope, référez le projet *WpfOscilloscope* dans les références du projet.

⇒ Dans l'entête du fichier XAML où vous voulez insérer l'oscilloscope, insérez le fragment de code suivant :

```
xmlns:oscillo="clr-namespace:WpfOscilloscopeControl;assembly=WpfOscilloscopeControl"
```

⇒ Dans votre fichier XAML, insérez le code suivant :

```
<oscillo:WpfOscilloscope x:Name="oscilloSpeed"/>
```

Votre oscilloscope devrait apparaître dans la partie graphique de votre fichier XAML. A présent, placez le où vous voulez en utilisant les grilles.

⇒ Pour ajouter une ligne à votre oscilloscope, insérez le code suivant à l'initialisation de votre classe WPF d'affichage :

```
|| oscilloSpeed.AddOrUpdateLine(lineId, 200, "Ligne1");
|| oscilloSpeed.ChangeLineColor(lineId, Color.Blue);
```

Examinez les paramètres de ces fonctions pour comprendre leur usage dans la bibliothèque fournie.

⇒ Pour afficher des données dans l'oscilloscope, vous pouvez utiliser le code suivant appelé depuis un timer d'affichage :

```
|| oscilloSpeed.AddPointToLine(lineId, xValue, yValue);
```

Vous devez à présent disposer d'un oscilloscope fonctionnel. Nous allons à présent faire la même chose pour implanter un composant permettant de visualiser sur une carte la position de notre robot, sa vitesse, et des objets qu'il peut détecter.

⇒ Téléchargez le composant WPF WpfWorldMapDisplay à la page du projet robot : [Télécharger WpfWorldMapDisplay](#).

⇒ Ajoutez ce composant à votre solution en tant que projet existant.

⇒ Dans votre projet WPF qui accueillera la carte, référez le projet WpfWorldMapDisplay dans les références du projet.

⇒ Dans l'entête du fichier XAML où vous voulez insérer la carte, insérez le fragment de code suivant :

```
|| xmlns:ext="clr-namespace:WpfWorldMapDisplay;assembly=WpfWorldMapDisplayIUT"
```

⇒ Dans votre fichier XAML, insérez le code suivant :

```
|| <ext:WorldMapDisplay x:Name="worldMapDisplay"/>
```

Votre carte devrait apparaître dans la partie graphique de votre fichier XAML. A présent, placez la où vous voulez en utilisant les grilles.

⇒ A présent vous pouvez utiliser cette carte comme bon vous semble. Les méthodes les plus utiles sont les suivantes : *UpdateRobotLocation* et *UpdateLidarMap*. A vous de comprendre comment les utiliser et de le faire à bon escient.

1.4 Affichage en C# des données de vitesse et d'asservissement

Afin de régler les asservissements, il est indispensable de voir en temps réel les données de vitesse issues de vos robots, ainsi que les sorties de chacune des parties des correcteurs PID. Pour cela nous allons utiliser des composants C# permettant d'afficher ces valeurs dans un tableau.

⇒ Téléchargez le projet WPF WpfAsservissementDisplay à la page du projet robot : [Télécharger WpfAsservissementDisplay](#).

⇒ Ajoutez ce composant à votre solution en tant que projet existant.

⇒ Dans votre projet WPF qui accueillera les données d'asservissement, référez le projet WpfAsservissementDisplay dans les références du projet.

⇒ Dans l'entête du fichier XAML où vous voulez insérer le tableau des données d'asservissement en vitesse, insérez le fragment de code suivant :

```
|| xmlns:AsservDisplay="clr-namespace:WpfAsservissementDisplay;assembly=WpfAsservissementDisplay"
```

⇒ Dans votre fichier XAML, insérez le code suivant :

```
<AsservDisplay:AsservissementRobot2RouesDisplayControl x:Name="asservSpeedDisplay"/>
```

Votre tableau des données d'asservissement en vitesse devrait apparaître dans la partie graphique de votre fichier XAML. A présent, placez le où vous voulez en utilisant les grilles. Il est à noter que si dans le futur vous souhaitez passer en version 4 roues holonomes, une version du tableau existe également et nécessite d'insérer le code suivant :

```
<AsservDisplay:AsservissementRobot4RouesHoloDisplayControl x:Name="asservSpeedDisplay4RouesHolo"/>
```

⇒ Examinez avec attention le code de la bibliothèque fournie. Vous noterez que deux types d'asservissement pour robot à deux roues peuvent être étudiés : l'asservissement polaire et l'asservissement indépendant (chaque roue est asservie de manière isolée).

⇒ Affichez des données simulées de vitesse et asservissement en polaire et en indépendant dans le tableau, en appelant les fonctions commençant par *Update...* depuis un timer. Validez que tout est correct (les données souhaitées s'affiche au bon endroit) au niveau du fonctionnement de cette bibliothèque graphique.

⇒ Affichez à présent dans le tableau les données de vitesse issues du robot. Pour l'instant seules les données de vitesse de vitesse linéaire et angulaire en polaire sont censées remonter depuis le robot et être affichées.

⇒ Ajoutez une trame en embarqué permettant de remonter également les vitesses de chacun des moteurs du robot, et affichez les dans l'interface à l'aide des fonctions prévues pour cela en polaire.

2 Asservissement en vitesse du robot

2.1 Un petit peu de théorie

Pourquoi et comment asservir ?

Dans cette partie vous allez asservir votre robot en vitesse. Cet asservissement peut être réalisé de manière indépendante : roue par roue avec un correcteur de type PI ou PID sur chaque roue comme présenté à la figure 1. Il peut également être réalisé en polaire : asservissement en vitesse linéaire et en vitesse angulaire comme indiqué à la figure 2.

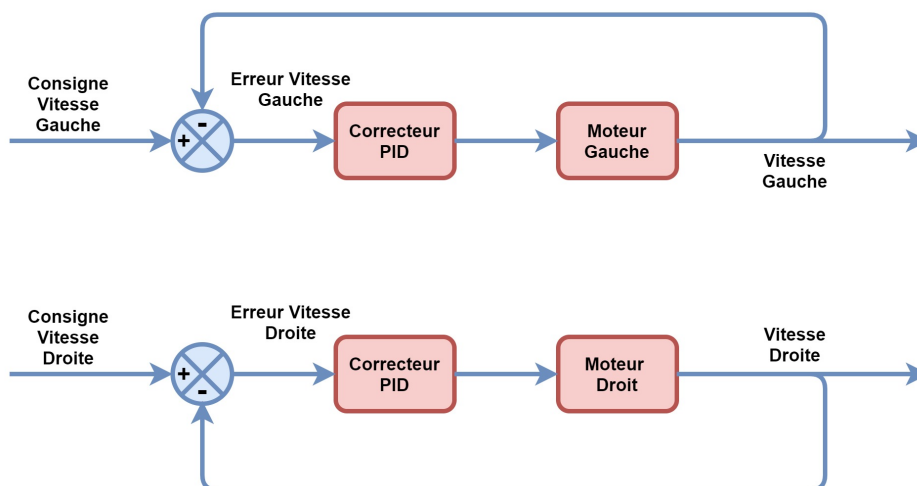


FIGURE 1 – Asservissement PID indépendant

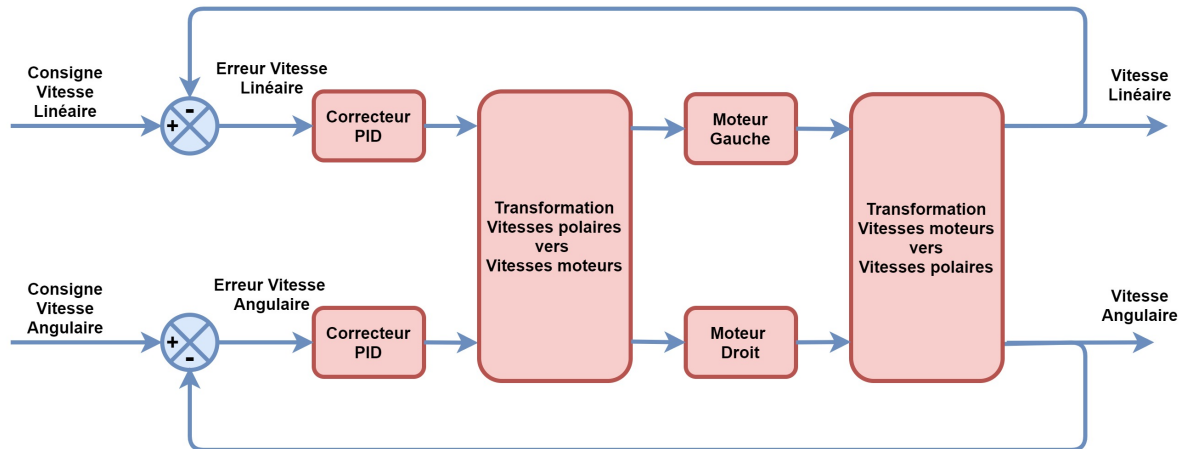


FIGURE 2 – Asservissement polaire en vitesse

Dans les deux cas, l'asservissement permettra d'assurer que les deux roues tournent 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.

Les performances des asservissements en polaires sont toutefois particulièrement intéressantes pour les robots à deux roues. En effet, les inerties en rotation et en translations d'un robot peuvent être très différentes. Supposons par exemple le cas théorique d'un robot constitué d'un barre verticale située au centre du robot et ayant un poids important. Cette barre a une inertie en translation forte mais une inertie en rotation très faible. Une consigne de vitesse appliquée sur l'un des deux moteurs fait bouger le robot en translation et en rotation de manière couplée. Si l'asservissement a un gain faible, le robot sera stable en rotation et en translation, mais son mouvement sera très *mou* en translation. Si à l'inverse le gain est fort, le mouvement sera réactif en translation mais risque d'être instable en rotation. Il serait donc dans ce cas souhaitable d'avoir un gain fort en translation et un gain faible en rotation : c'est ce que permet l'asservissement en polaire, le découplage entre les deux asservissements. Il est toutefois à noter que cet asservissement en polaire pose problème dans le cas d'un robot à 4 roues holonomes, étudié ultérieurement et que dans ce cas, on utilisera un asservissement indépendant sur chacun des 4 moteurs, les consignes données à ceux-ci étant en revanche issues d'une commande de type polaire.

Les transformations polaire-indépendant et indépendant-polaire

Quel que soit le type d'asservissement choisi, la commande au niveau du moteur s'effectuera toujours à l'aide d'une tension moteur, et les données issues des capteurs seront toujours liées à un moteur unique. Il va donc être nécessaire de passer de la commande polaire en vitesse qui sera utilisée quelque soit le type d'asservissement choisi à une commande indépendante moteur par moteur et vice-versa.

La transformation de indépendant à polaire a été étudiée dans la partie odométrie, elle peut se mettre sous la forme matricielle suivante, avec R_{Robot} le rayon du robot :

$$\begin{pmatrix} V_{Linéaire} \\ V_{Angulaire} \end{pmatrix} = \begin{pmatrix} \frac{1}{2} & \frac{1}{2} \\ \frac{1}{2R_{Robot}} & -\frac{1}{2R_{Robot}} \end{pmatrix} \begin{pmatrix} V_{Droit} \\ V_{Gauche} \end{pmatrix}$$

En inversant la matrice, on obtient la relation inverse permettant de calculer les vitesses des 2 moteurs à partir des vitesses polaires :

$$\begin{pmatrix} V_{Droit} \\ V_{Gauche} \end{pmatrix} = \begin{pmatrix} 1 & R_{Robot} \\ 1 & -R_{Robot} \end{pmatrix} \begin{pmatrix} V_{Linéaire} \\ V_{Angulaire} \end{pmatrix}$$

Comment implanter l'asservissement ?

Un asservissement PID peut s'implanter de plusieurs manières. Vous allez l'implanter d'une manière assez robuste, en déterminant les effets proportionnel (P), intégral (I) et dérivé (D) du correcteur de manière séparée à partir de l'erreur de vitesse $\varepsilon(n)$. Chaque terme est calculé en deux étapes, la première est le calcul de l'erreur bornée, qu'elle soit proportionnelle, dérivée ou intégrale. Cette étape de bornage est indispensable, surtout pour la correction intégrale de manière à ce qu'elle ne puisse excéder une valeur maximum ($C_{P/I/D_{Max}}$) de correction (dans un sens ou dans l'autre) qui peut être très rapidement atteinte si une roue se bloque. Si on ne faisait pas cela, il faudrait laisser l'intégrale se *vider* après déblocage, ce qui engendrerait un mouvement incontrôlé du robot. Les calculs des différentes corrections sont effectués comme suit :

— Correction proportionnelle C_P :

$$Err_P(n) = LimitToInterval\left(\varepsilon(n), -\frac{C_{P_{Max}}}{K_P}, \frac{C_{P_{Max}}}{K_P}\right)$$

$$C_P(n) = K_P * Err_P(n)$$

— Correction intégrale C_I :

$$Err_I(n) = LimitToInterval\left(Err_I(n-1) + \frac{\varepsilon(n)}{fEch}, -\frac{C_{I_{Max}}}{K_I}, \frac{C_{I_{Max}}}{K_I}\right)$$

$$C_I(n) = K_I * Err_I(n)$$

— Correction dérivée C_D :

$$Err_D(n) = LimitToInterval((\varepsilon(n) - \varepsilon(n-1))fEch, -\frac{C_{D_{Max}}}{K_D}, \frac{C_{D_{Max}}}{K_D})$$

$$C_D(n) = K_D * Err_D(n)$$

Une fois les trois termes calculés, la commande (polaire ou indépendante) en vitesse est la somme des 3 termes précédents.

$$C_{PID} = C_P + C_I + C_D$$

2.2 Implantation de la commande polaire en vitesse

Nous allons commander les moteurs en commande de vitesse polaire. De plus, les moteurs seront commandé par des vitesses exprimée en $m.s^{-1}$.

⇒ Pour cela, ajoutez à votre code la fonction `PWMSetSpeedConsignePolaire` définie comme ci-dessous. Cette fonction va permettre de générer les consignes à placer sur chaque moteur pour avoir en boucle ouverte un robot ayant une vitesse linéaire et une vitesse angulaire à peu près conformes à la consigne.

```
#define M_TO_PERCENT 10

void PWMSetSpeedConsignePolaire(float vitesseLineaire, float vitesseAngulaire) {

    robotState.vitesseDroiteConsigne = (...);
    robotState.vitesseGaucheConsigne = (...);

    robotState.vitesseDroitePercent = -M_TO_PERCENT * robotState.vitesseDroiteConsigne;
    robotState.vitesseGauchePercent = M_TO_PERCENT * robotState.vitesseGaucheConsigne;

    LimitToInterval(robotState.vitesseDroitePercent , -100, 100);
    LimitToInterval(robotState.vitesseGauchePercent , -100, 100);
}
```

⇒ Complétez la fonction à partir des équations matricielles proposées précédemment, en laissant le coefficient

M_TO_PERCENT présent, sans tester à ce stade.

On va à présent utiliser uniquement cette fonction dans le code pour passer les consignes de vitesse aux moteurs.

⇒ Commentez la fonction `PWMSetSpeedConsigne` afin qu'elle ne puisse plus envoyer de commandes aux moteurs.

⇒ Remplacez les appels à la fonction `PWMSetSpeedConsigne` par des appels à `PWMSetSpeedConsignePolaire` (en les adaptant).

⇒ Pensez également à désactiver l'appel à `OperatingSystemLoop` dans les Timer (afin de ne plus être en mode automatique)

⇒ Appelez la fonction `PWMSetSpeedConsignePolaire` avec une vitesse linéaire de 1m.s-1 et une vitesse angulaire de 0 et testez votre code.

⇒ A présent réglez la valeur de `M_TO_PERCENT` de manière à avoir une vitesse linéaire réelle à peu près conforme (à $\pm 20\%$) à la vitesse linéaire souhaitée.

⇒ Vérifiez que ça fonctionne aussi pour la vitesse angulaire.

2.3 Implantation de l'asservissement polaire

Pour implanter les asservissements dans les robots en embarqué, vous allez devoir implanter les équations précédentes. Afin de rendre le code assez générique, vous allez utiliser une structure `PidCorrector` implantée en C, qui va être utilisée dans l'esprit d'une classe.

⇒ Insérez le code de cette structure `PidCorrector` dans un fichier `asservissement.h` :

```
typedef struct _PidCorrector
{
    double Kp;
    double Ki;
    double Kd;
    double erreurProportionnelleMax;
    double erreurIntegraleMax;
    double erreurDeriveeMax;
    double erreurIntegrale;
    double epsilon_1;
    double erreur;

    //For Debug only
    double corrP;
    double corrI;
    double corrD;
}PidCorrector;
```

⇒ Dans le fichier `robotstate.c`, déclarez deux structures de type `PidCorrector`, l'une dénommée `PidX` pour l'asservissement en vitesse linéaire et l'autre dénommée `PidTheta` pour l'asservissement en vitesse angulaire.

⇒ Dans le fichier `asservissement.c` (à créer si n'existe pas), insérez la fonction `SetupPidAsservissement` permettant de régler les coefficient des `PidCorrector`. Vous noterez que l'on accède aux éléments d'une structure avec un flèche `->` :

```
void SetupPidAsservissement(volatile PidCorrector* PidCorr, double Kp, double Ki, double Kd,
                           double proportionnelleMax, double integraleMax, double deriveeMax)
{
    PidCorr->Kp = Kp;
```

```

    PidCorr->erreurProportionnelleMax = proportionnelleMax; //On limite la correction due au Kp
    PidCorr->Ki = Ki;
    PidCorr->erreurIntegraleMax = integraleMax; //On limite la correction due au Ki
    PidCorr->Kd = Kd;
    PidCorr->erreurDeriveeMax = deriveeMax;
}

```

⇒ Rajoutez à présent en embarqué et en C# dans un message permettant d'appeler la fonction précédente depuis l'interface en C# (avec des coefficients fixés en C#), de manière à paramétrer à distance les coefficients des PID linéaire et angulaire.

⇒ Rajoutez des fonctions de transmission de données de l'embarqué vers le C# qui vous permettront de valider que les valeurs de configuration de l'asservissement ont bien été prise en compte en embarqué. Celles-ci devront être affichées dans le tableau C# dédié à la supervision de l'asservissement. Validez cette étape avec le professeur avant d'aller plus loin.

⇒ Rajoutez également une fonction en C# permettant de transmettre depuis l'interface les consignes de vitesses linéaire et angulaire à l'embarqué.

⇒ Dans le fichier *asservissement.c*, implantez à présent une fonction permettant de calculer les corrections en complétant le template ci-dessous. La fonction prend en arguments le pointeur vers la structure *PidCorrector* et l'erreur courante. Elle sera appelée à chaque fois qu'une erreur de vitesse est calculée.

```

double Correcteur(volatile PidCorrector* PidCorr, double erreur)
{
    PidCorr->erreur = erreur;
    double erreurProportionnelle = LimitToInterval(...);
    PidCorr->corrP = ...;

    PidCorr->erreurIntegrale += ...;
    PidCorr->erreurIntegrale = LimitToInterval(...);
    PidCorr->corrI = ...;

    double erreurDerivee = (erreur - PidCorr->epsilon_1)*FREQ_ECH_QEI;
    double deriveeBornee = LimitToInterval(erreurDerivee, -PidCorr->erreurDeriveeMax/PidCorr->Kd,
        PidCorr->erreurDeriveeMax/PidCorr->Kd);
    PidCorr->epsilon_1 = erreur;
    PidCorr->corrD = deriveeBornee * PidCorr->Kd;

    return PidCorr->corrP+PidCorr->corrI+PidCorr->corrD;
}

```

⇒ Implantez une fonction *UpdateAsservissement* qui sera appelée à chaque mesure de la vitesse par odométrie à la place de *PWMSetSpeedConsignePolaire*. Cette fonction aura le template suivant, qu'il vous faut compléter :

```

void UpdateAsservissement()
{
    robotState.PidX.erreur = ...;
    robotState.PidTheta.erreur = ...;

    robotState.CorrrectionVitesseLineaire =
        Correcteur(&robotState.PidX, robotState.PidX.erreur);
    robotState.CorrrectionVitesseAngulaire = ...;

    PWMSetSpeedConsignePolaire(robotState.CorrrectionVitesseLineaire,

```

```

    robotState.CorrectionVitesseAngulaire);
}

```

⇒ Rajoutez des fonctions de transmission de données de l'embarqué vers le C# qui vous permettront de superviser toutes les variables internes des correcteurs PID. Celle-ci devront être affichées dans le tableau C# dédié à la supervision de l'asservissement. Validez l'affichage avec le professeur, en désactivant **temporairement** l'appel de la fonction *UpdateAsservissement* et en initialisant les valeurs à superviser dans la fonction *SetupPidAsservissement*. **Après validation par le professeur, revenir en arrière sur ces deux modifications.**

2.4 Réglage de l'asservissement polaire

Vous avez à présent bâti la structure de votre asservissement polaire, et il est à présent temps de le régler. Le réglage s'effectue en déterminant dans un premier temps les coefficients de l'asservissement angulaire, avant de déterminer dans un second temps ceux de l'asservissement linéaire.

⇒ Commencez par initialiser les K_p , K_i et K_d à 0 pour tous les correcteurs. Les valeurs maximales des corrections seront placées à un niveau très élevé (par exemple 100). Vérifiez que les moteurs ne tournent pas.

⇒ On va commencer par régler l'asservissement en vitesse angulaire. Pour cela forcez la variable *robotState.PidX.erreur* à 0 dans la fonction *UpdateAsservissement* (mettez en commentaire le code de fonctionnement normal, il va servir ensuite). L'entrée du correcteur de vitesse linéaire est donc forcée à 0, ce qui supprime tout effet de ce correcteur. Mettez à présent une consigne de vitesse angulaire nulle (attention, pas une erreur nulle mais une consigne nulle) en entrée.

⇒ Augmentez progressivement le gain K_P angulaire jusqu'à un $K_{P_{Limite}}$ déclenchant l'oscillation du robot. **Dès que cela se produit, arrêtez rapidement la manipulation au risque de griller les moteurs.** On prendra comme valeur optimale $K_P = K_{P_{Limite}}/2$. Vérifiez qu'avec ce réglage le robot se comporte correctement. Vous devez sentir une forte résistance liée à l'asservissement en posant le robot au sol et en essayant de la faire tourner sur lui-même.

⇒ Une fois K_P réglé, nous allons régler K_I . Augmentez progressivement le gain K_I angulaire jusqu'à un $K_{I_{Limite}}$ déclenchant l'oscillation du robot. **Dès que cela se produit, arrêtez rapidement la manipulation au risque de griller les moteurs.** On prendra comme valeur optimale $K_I = 0.7K_{I_{Limite}}$. Vérifiez qu'avec ce réglage le robot se comporte correctement. Vous ne devriez quasiment plus pouvoir faire tourner le robot sur lui-même sans le faire glisser, mais il doit toujours pouvoir se déplacer en translation.

⇒ Une fois K_P et K_I réglés, reste le réglage de K_D angulaire. Nous le maintiendrons à 0 pour l'instant, le correcteur sera donc de type *PI*.

Une fois le réglage du correcteur angulaire effectué, il est temps de passer au réglage du correcteur linéaire.

⇒ Mettez une consigne de vitesse linéaire à 0 et supprimez le forçage de *robotState.PidX.erreur* à 0 dans la fonction *UpdateAsservissement* en restaurant le code normal. Laisser le correcteur angulaire déjà réglé en fonctionnement, il empêche le robot de tourner sur lui-même vu que la consigne de vitesse angulaire est toujours nulle.

⇒ Augmentez progressivement le gain K_P linéaire jusqu'à un $K_{P_{Limite}}$ déclenchant l'oscillation du robot. **Dès que cela se produit, arrêtez rapidement la manipulation au risque de griller les moteurs.** On prendra comme valeur optimale $K_P = K_{P_{Limite}}/2$. Vérifiez qu'avec ce réglage le robot se comporte correctement. Vous devez sentir une forte résistance liée à l'asservissement en posant le robot au sol et en essayant de le pousser.

⇒ Une fois K_P réglé, nous allons régler K_I linéaire. Augmentez progressivement le gain K_I linéaire jusqu'à un $K_{I_{limite}}$ déclenchant l'oscillation du robot. **Dès que cela se produit, arrêtez rapidement la manipulation au risque de griller les moteurs.** On prendra comme valeur optimale $K_I = 0.7K_{I_{limite}}$. Vérifiez qu'avec ce réglage le robot se comporte correctement. Vous ne devriez quasiment plus pouvoir faire déplacer le robot sans le faire glisser, ni en translation, ni en rotation.

⇒ Laissez K_D linéaire à 0 pour l'instant.

⇒ Testez à présent votre robot avec des véritables consignes de vitesse issues de la manette de jeu ou du clavier. A ce stade, votre robot doit être asservi en vitesse linéaire et angulaire. Votre moteur doit notamment pouvoir fonctionner à des très faibles vitesses de rotations si l'asservissement fonctionne bien. Validez l'ensemble avec le professeur. Si c'est bien le cas, vous avez terminé la partie asservissement en vitesse.

3 Asservissement en position avec générateur de trajectoire

Le module de génération de trajectoires est un module important pour le robot, puisqu'il a pour vocation de permettre le déplacement du robot vers un point défini. Il peut également permettre de passer d'une cible à une autre en cours de déplacement. Ces déplacements doivent s'effectuer de manière asservie en veillant à ce que le déplacement du robot soit le plus précis et le plus fluide possible, avec la possibilité de détecter un blocage dans le mouvement du à un obstacle non détecté par le robot au préalable. Pour cela, on effectuera l'asservissement du robot en deux étapes :

- La première est de générer à tout instant la position théorique du robot. Cette position théorique est appelée le *ghost*. Cette position étant théorique, elle est exempte de parasites dus par exemple à un bruit de mesure dans l'odométrie.
- La seconde est d'asservir le robot réel sur la position du *ghost* à l'aide d'un correcteur à déterminer.

3.1 Génération du déplacement du *ghost*

Dans le cas des robots à deux roues type Eurobot, les contraintes de déplacement sont liées au caractère non-holonyme du robot qui ne permet pas de déplacement latéraux, mais seulement des déplacements longitudinaux et des rotations. De fait, dans ce sujet, nous adopterons une stratégie de déplacement relativement simple en deux étapes :

- Rotation dans la direction du point cible.
- Déplacement longitudinal dans l'axe du robot vers la cible une fois que la rotation est terminée.

Ces deux mouvements seront de type trapézoïdaux en vitesse, de manière à éviter des accélérations brusques pouvant conduire à un glissement du robot. Ils seront chaînés à l'aide d'une machine à état finis ayant 3 états au départ (*Idle*, *Rotation*, *DeplacementLineaire*) définis dans un *enum*.

En état *Idle*, le robot est à l'arrêt, avec des consignes de vitesse linéaires et angulaire générées égales à 0.

En état *Rotation*, l'algorithme à implanter est le suivant :

Algorithm 1: Orientation du ghost vers le waypoint

```

Calcul initiaux
 $\theta_{Restant} = ModuloByAngle(\theta_{ghost}, \theta_{Waypoint}) - \theta_{ghost}$ 
 $\theta_{Arrêt} = \frac{V_{\theta}^2}{2 * Acc_{\theta}}$ 
 $increment_{\theta} = V_{\theta} * T_{sampling}$ 

if  $V_{\theta} < 0$  then
|  $\theta_{Arrêt} = -\theta_{Arrêt}$ 

if  $((\theta_{Arrêt} >= 0 \ \&\& \ \theta_{Restant} >= 0) \ || \ (\theta_{Arrêt} <= 0 \ \&\& \ \theta_{Restant} <= 0))$ 
&&  $|\theta_{Restant}| >= |\theta_{Arrêt}|$  then
| // On accélère en rampe saturée
| if  $\theta_{Restant} > 0$  then
| | // Si la destination est devant, on accélère en positif en saturant la vitesse à  $V_{\theta_{Max}}$ 
| |  $V_{\theta} = Min(V_{\theta} + Acc_{\theta}/F_{QEI}, V_{\theta_{Max}})$ 
| else if  $\theta_{Restant} < 0$  then
| | //Si la destination est derrière, on accélère en négatif en saturant la vitesse à  $-V_{\theta_{Max}}$ 
| | ...
| else
| | // On freine en rampe saturée
| | if  $V_{\theta} > 0$  then
| | | //Si la vitesse positive est positive, on freine en positif en saturant la vitesse à 0
| | | ...
| | else if  $V_{\theta} < 0$  then
| | | //Si la vitesse est négative, on freine en négatif en saturant la vitesse à 0
| | | ...
| | if  $(Abs(thetaRestant) < Abs(incrementAng))$  then
| | |  $increment_{\theta} = \theta_{Restant}$ 

// On intègre le déplacement
 $\theta_{ghost} = \theta_{ghost} + increment_{\theta}$ 

// On gère les erreurs numériques d'arrondis
if  $(V_{\theta} == 0 \ \&\& \ |\theta_{Restant}| < 0.01)$  then
|  $\theta_{ghost} = \theta_{Waypoint}$ 

```

⇒ Implantez cet algorithme de rotation en expliquant bien le rôle de *ModuloByAngle* et la formule de $\theta_{Arrêt}$. Testez-le en validant sur votre interface graphique que la rotation effectuée par le *Ghost* est correcte. Validez avec le professeur cette étape avant de passer à la suivante.

A présent, votre robot se positionne face à la cible, il est donc possible de le faire avancer vers celle-ci. Attention, il est cependant possible que comme dans le cas de la Figure 3, l'axe de déplacement du robot ne soit pas parfaitement l'axe du waypoint (c'est d'ailleurs toujours le cas). Il ne sera donc pas possible de réduire à 0 la distance entre le robot et le waypoint avec un mouvement linéaire. La solution est dans ce cas, de déterminer le projeté du waypoint sur l'axe de déplacement, et de faire les calculs en utilisant comme distance restant à parcourir la distance entre ce projeté et le robot. Dans ce cas, on atteindra le point le plus proche possible du robot.

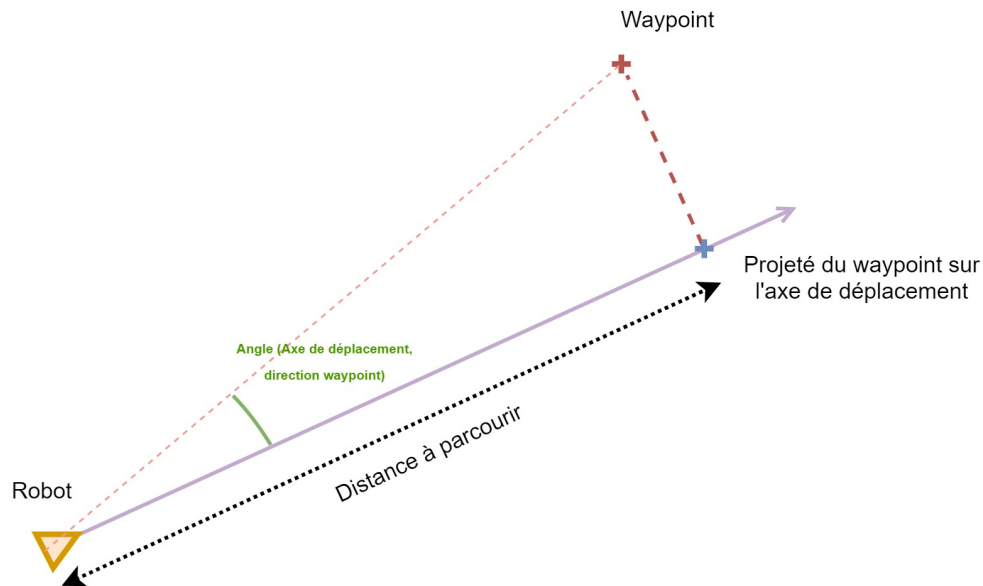


FIGURE 3 – Prise en compte de l'erreur angulaire lors d'un déplacement linéaire (phase 2 du mouvement)

⇒ Implantez dans un premier temps une fonction permettant de calculer cette distance entre un point à parcourir en vous inspirant de la fonction *DistancePointToSegment* présente dans *Utilities* → *Toolbox*. Si vous ne trouvez pas, demandez l'aide au professeur.

⇒ Déterminez ensuite l'angle entre l'axe de déplacement et la direction du waypoint. Si cet angle est compris dans l'intervalle $[-90^\circ; 90^\circ]$, alors le waypoint est placé devant le robot, sinon, il est placé derrière. Cette condition jouera le même rôle que le $\theta_{Restant}$ dans l'algorithme de rotation.

⇒ Implantez à présent un algorithme de déplacement longitudinal ressemblant fortement à l'algorithme de rotation. Validez-le à l'aide de l'interface du robot. Lorsque le mouvement est terminé et que le waypoint (ou plutôt sa projection) est atteint, la machine à état doit repasser en état *Idle*.

3.2 Asservissement du robot réel sur le *ghost*

Vous disposez à présent d'un générateur de trajectoire permettant de faire se déplacer le *Ghost* vers un point déterminé. Vous allez à présent faire en sorte de faire d'asservir le robot sur le *Ghost* commandé à la Figure 4. Vous disposez pour cela de deux correcteurs PD dans le générateur de trajectoire dénommés *PD_Position_Lineaire* et *PD_Position_Angulaire*.

⇒ Justifiez pourquoi il est nécessaire ici d'utiliser des correcteurs PD et non pas PI.

⇒ En réglant pour commencer les correcteurs PD à $K_P = 0$ et $K_D = 1$, implantez l'ensemble de la chaîne d'asservissement, en faisant attention à l'implantation de l'estimateur d'erreur linéaire explicité à la figure 5 (pensez à déterminer le signe de l'erreur en plus de sa valeur). Le code est à insérer dans la fonction *PIDPosition()* du générateur de trajectoire.

⇒ Quand le robot converge vers le *ghost*, potimisez les paramètres de l'asservissement en position, en ajustant K_D dans un premier temps puis K_P ensuite.

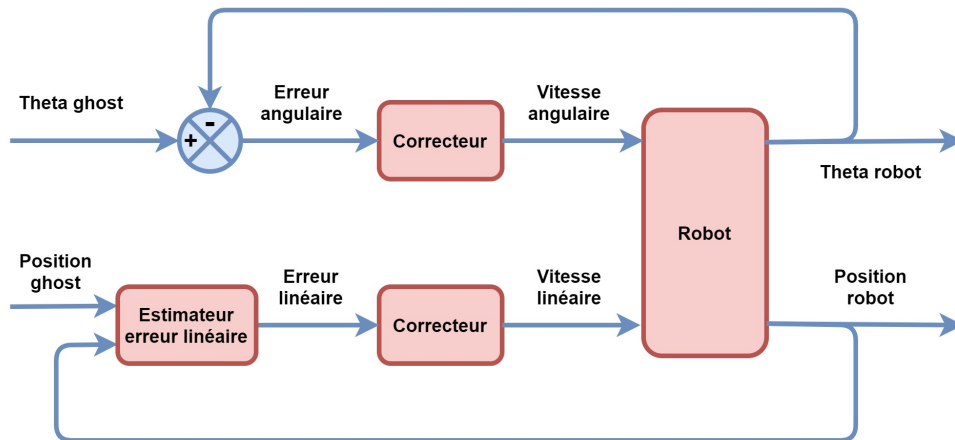


FIGURE 4 – Asservissement du robot sur le *ghost*

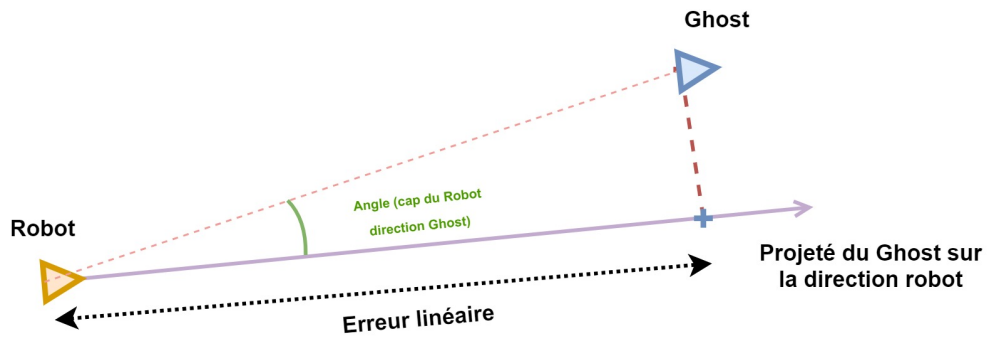


FIGURE 5 – Estimation de l'erreur linéaire

⇒ L'optimisation des correcteurs étant effectuée, vous pouvez à présent faire se déplacer le robot à n'importe quel endroit de la carte à l'aide d'un *CTRL+Click* sur la carte. Validez avec le professeur le bon fonctionnement du générateur de trajectoire.