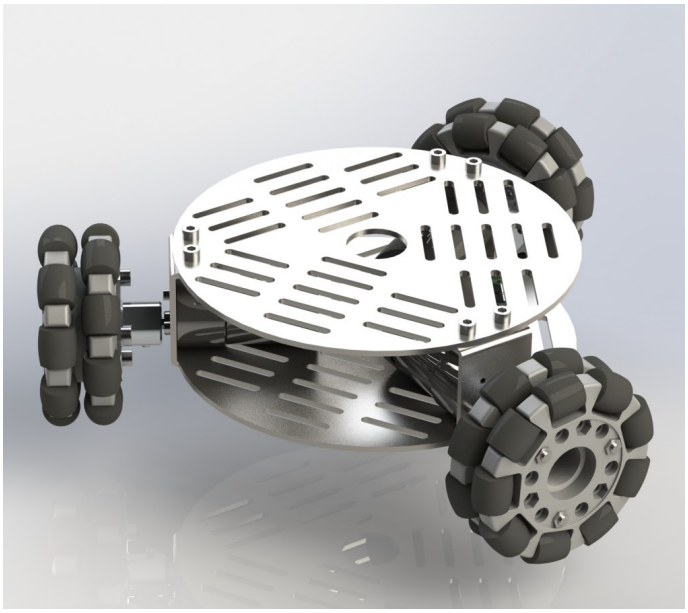
Commande ROS d'un robot holonome

ROB314 - Architectures mécatronique pour la robotique

Bastien HUBERT



ENSTA Paris - mars 2023

Table des matières :

1 Intérêts de la navigation holonome

1.1 Limites de la navigation classique

Que ce soit dans le cadre de la navigation de robots mobiles ou celui de la conduite de voiture, un certain nombre de déplacements nécessitent d'être décomposée en un certain nombre de sous-déplacements plus simple avant de pouvoir être effectués : les manoeuvres. De tels déplacements sont coûteux en temps de calcul (car leur décomposition n'est pas toujours immédiate), en temps de déplacement (car ces déplacements peuvent être nombreux et se font souvent à petites vitesses ou passent par des phases d'arrêt avant d'entamer la suivante), et en précision (car les erreurs inévitables à chaque sous-mouvement se cumulent entre elles).

De plus, d'autres mouvements ne peuvent être réalisés que dans certaines configurations géométriques, comme c'est le cas pour les traversées d'obstacles et de zones étroites, ou pour des opérations de re-calibrage par rapport à un repère fixe.

Enfin, les missions de suivi visuel d'objet en mouvement sont grandement compliquées par la contrainte d'orientation de la base mobile selon son mouvement, ce qui nécessite des algorithmes de vision plus robustes aux pertes d'information, des algorithmes de navigation assez complexes pour pouvoir prendre en compte des contraintes d'orientation, ou des actionneurs supplémentaires.

Pour toutes ces raisons, le modèle de navigation des robots unicycles, tricycles et quadricycle (étudiés en ROB316) ne sont pas satisfaisant dans bon nombre de situations où le robot doit se déplacer efficacement dans des terrains à géométrie complexe ou parsemés d'obstacles.

1.2 Qu'est-ce qu'un robot holonome ?

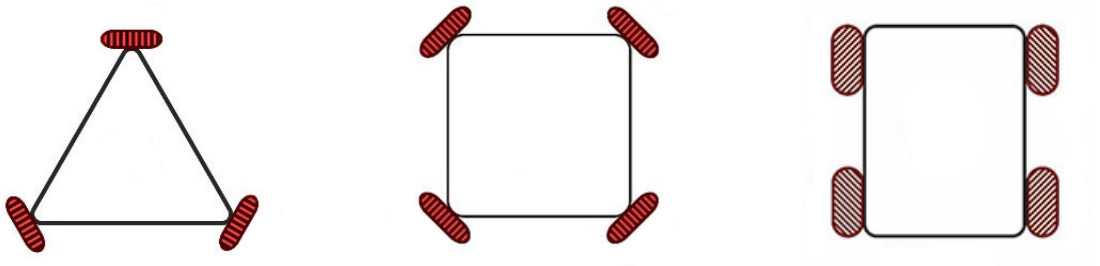
Un robot holonome est un robot mobile se déplaçant à l'aide de roues holonomes, qui sont des roues spécialement conçues pour pouvoir se déplacer dans n'importe quelle direction sur un plan. On en compte 2 types principaux : les roues holonomes à 90° (dites *omniwheels*) et les roues holonomes à 45° (dites *mecanum wheels*) :

(a) 90°

(b) 45°

Figure 1: Les 2 types de roues holonomes

Ces roues peuvent être disposées de différentes façons pour former une base mobile holonome, mais on compte 3 configurations usuelles :



(a) 3 roues à 90° (b) 4 roues à 90° (c) 4 roues à 45°

Figure 2: Les 3 configurations de robots holonomes

Il est bien sûr possible de rajouter autant d'actionneurs et de capteurs que souhaité au-dessus de chacune des ces bases, mais elle constituent l'essentiel de l'aspect holonomique du robot, et nous nous limiterons ici à leur étude.

1.3 Pourquoi utiliser des robots holonomes ?

Le principal intérêt des roues holonomes, et donc des bases mobiles qui les utilisent, est leur capacité à se déplacer sans glisser dans n'importe quelle direction sur un plan. En effet, elles disposent d'un axe de rotation tangentiel motorisable comme les roues classiques, mais également de galets (répartis différemment en fonction du type de roue) qui leur permet de rouler librement radialement à l'axe principal.

La combinaison de ces 2 déplacements potentiels fait qu'il est possible pour les bases holonomes de se déplacer dans n'importe quelle direction plane et de leur imprimer n'importe quelle rotation plane sans engendrer plus de frottements ou de glissements que dans le cadre d'une base mobile classique se déplaçant selon la direction de ses roues, et ce peu importe l'orientation du robot.

La navigation holonome permet donc de découpler entièrement les déplacements selon l'axe avant-arrière du robot, son axe latéral, et les rotations autour de son axe vertical, éliminant ainsi tout besoin de manoeuvre en position comme en orientation.

1.4 Une application concrète : la Coupe de France de Robotique

Prenons un exemple concret de déplacement d'un robot mobile sous contraintes : la Coupe de France de Robotique. Celle-ci consiste à faire effectuer un certain nombre de tâches à plus ou moins haut niveau de précision à un robot dans un espace limité et en temps imparti. Celui-ci doit de plus se déplacer entre les différentes tâches car elles ne se situent pas toutes au même endroit. Les questions de rapidité et précision (ou correction efficace de l'erreur le cas échéant) des déplacements du robot utilisé sont donc cruciales pour remplir au mieux le plus de tâches possibles.

On peut, par exemple, vouloir suivre l'évolution du robot adverse (sans que cela entrave nos mouvements) afin d'adapter notre stratégie de déplacement : être plus ou moins agressif, anticiper les zones d'intérêt du robot adverse pour le contrer ou s'en éloigner pour éviter les collisions, etc.

On peut également avoir besoin d'annuler l'erreur en position du robot en se recalibrant à l'aide des bords de la zone délimitée, ce qui nécessite que le second calibrage ne génère pas des incertitudes annulant l'intérêt du premier calibrage.

Enfin, on peut se rendre compte trop tard d'un écart en position latéral non désiré avec un centre d'intérêt (par exemple dans le cas d'actions à effectuer sur les bords de la zone), empêchant ainsi le bon déroulement d'une tâche.

Dans toutes ces situations, il est clair que l'utilisation d'une base mobile classique n'apporte pas de solution satisfaisante sans engendrer un surcoût important : l'utilisation de robots holonomes est donc une nécessité.

NB : Pour des raison pratiques visant à limiter à la fois la taille du robot, sa consommation énergétique en calculs comme en actions, et le nombre de ports utilisés sur l'ordinateur embarqué, nous ne nous intéresserons ici qu'au cas du robot holonome à 3 roues à 90°.

2 Commande de robots holonomes

Si nous avons vu que la navigation holonome permet d'éliminer efficacement un certain nombre de défauts de la navigation classique, notamment en supprimant pour de bon la nécessité d'effectuer des manoeuvres coûteuses en temps et en précision, son fonctionnement est moins intuitif que dans le cadre d'un robot à roues classiques. Dans cette section, ne détaillerons les équations qui régissent le modèle cinématique d'un robot holonome à 3 roues afin d'en implémenter la commande dans la section 3.

2.1 Le modèle cinématique inverse d'une roue holonome

Commençons par considérer une unique roue holonome en liaison encastrement avec un point O dans un repère $\mathcal{R}_R$. Le point O correspond à la projection orthogonale sur le sol (selon l'axe vertical z_R) du centre géométrique du corps du robot, supposé rigide, et le repère $\mathcal{R}_R$ correspond au repère orthonormé direct associé à ce point. Le centre de la roue, de rayon r, se situe à la verticale du point M_i, lequel est à une distance l du point O. L'axe OM_i forme un angle α_i avec l'axe x_R qui correspond à l'angle entre la roue étudiée et une direction arbitrairement définie comme l'avant du robot. O est par ailleurs en mouvement dans le référentiel terrestre $\mathcal{R}_0$ comme suit :

- translation à vitesse linéaire $\overrightarrow{V_{0\in R/\mathcal{R}_0}} := \dot{x}\,\overrightarrow{x_R} + \dot{y}\,\overrightarrow{y_R}$
- rotation à vitesse angulaire $\overrightarrow{\Omega_{R/\mathcal{R}_0}} := \omega\,\overrightarrow{z_R} = \dot{\theta}\,\overrightarrow{z_R}$

Le mouvement de O engendre un déplacement de M_i dans $\mathcal{R}_0$ à vitesse linéaire $\overrightarrow{V_{M_i\in R/\mathcal{R}_0}} := \dot{u}\,\overrightarrow{u_i} + \dot{v}\,\overrightarrow{v_i}$. Enfin, la roue tourne à une vitesse angulaire $\omega_i\,\overrightarrow{u_i}$.

Le schéma 3 représente la situation ainsi décrite :

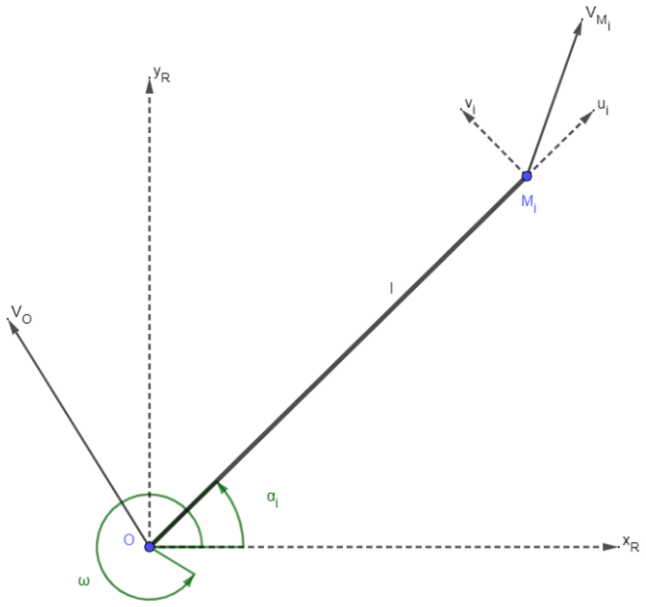


Figure 3: Modèle cinématique d'une roue holonome dans le référentiel $\mathcal{R}_R$ du robot

Comme le corps du robot est supposé rigide, on peut écrire son torseur cinématique $\{\mathcal{V}_{R/\mathcal{R}_0}\}$ supposé constant en tout point du solide :

$$\{\mathcal{V}_{R/\mathcal{R}_0}\} := \left\{ \begin{array}{c} \overrightarrow{\Omega_{R/\mathcal{R}_0}} \\ \overrightarrow{V_{O\in R/\mathcal{R}_0}} \end{array} \right\}_{O,\mathcal{R}_0} = \left\{ \begin{array}{c} \dot{\theta}\,\overrightarrow{z_R} \\ \dot{x}\,\overrightarrow{x_R} + \dot{y}\,\overrightarrow{y_R} \end{array} \right\}_{O,\mathcal{R}_0} \tag{1}$$

D'après la formule de transport de Varignon, on peut réécrire le torseur cinématique $\{\mathcal{V}_{\mathcal{R}/R_0}\}$ au point M_i comme suit :

$$\{\mathcal{V}_{R/\mathcal{R}_0}\} = \left\{ \begin{array}{c} \dot{\theta}\,\overrightarrow{z_R} \\ \dot{x}\,\overrightarrow{x_R} + \dot{y}\,\overrightarrow{y_R} + \overrightarrow{OM_i} \wedge \dot{\theta}\,\overrightarrow{z_R} \end{array} \right\}_{M_i,\mathcal{R}_0} \tag{2}$$

De plus, on peut calculer le produit $\overrightarrow{OM_i} \wedge \dot{\theta}\,\overrightarrow{z_R}$ en utilisant l'orthonormalité directe de $\mathcal{R}_R$:

$$\begin{aligned} \overrightarrow{OM_i} \wedge \dot{\theta}\,\overrightarrow{z_R} &= (\, l\cos(\alpha_i)\,\overrightarrow{x_R} + l\sin(\alpha_i)\,\overrightarrow{y_R}\,) \wedge \dot{\theta}\,\overrightarrow{z_R} \\ &= l\dot{\theta}(\, -\cos(\alpha_i)\,\overrightarrow{y_R} + \sin(\alpha_i)\,\overrightarrow{x_R}\,) \end{aligned} \tag{3}$$

D'où (2) devient :

$$\begin{aligned} \{\mathcal{V}_{R/\mathcal{R}_0}\} &= \left\{ \begin{array}{c} \dot{\theta}\,\overrightarrow{z_R} \\ \dot{x}\,\overrightarrow{x_R} + \dot{y}\,\overrightarrow{y_R} + l\dot{\theta}(\, -\cos(\alpha_i)\,\overrightarrow{y_R} + \sin(\alpha_i)\,\overrightarrow{x_R}\,) \end{array} \right\}_{M_i,\mathcal{R}_0} \\ &= \left\{ \begin{array}{c} \dot{\theta}\,\overrightarrow{z_R} \\ (\,\dot{x} + l\dot{\theta}\sin(\alpha_i)\,)\,\overrightarrow{x_R} + (\,\dot{x} - l\dot{\theta}\cos(\alpha_i)\,)\,\overrightarrow{y_R} \end{array} \right\}_{M_i,\mathcal{R}_0} \end{aligned} \tag{4}$$

En outre, par hypothèse de roulement sans glissement de la roue sur le sol, on a :

$$\overrightarrow{V_{M_i\in R/\mathcal{R}_0}} = \dot{u}\,\overrightarrow{u_i} + \dot{v}\,\overrightarrow{v_i} = \dot{u}\,\overrightarrow{u_i} + \omega_i\, r\,\overrightarrow{v_i} \tag{5}$$

Le calcul de $\overrightarrow{V_{M_i\in R/\mathcal{R}_0}} \cdot \overrightarrow{v_i}$ dans (4) et (5) permet alors d'obtenir l'égalité suivante :

$$\begin{aligned} \omega_i\, r &= (\,(\,\dot{x} + l\dot{\theta}\sin(\alpha_i)\,)\,\overrightarrow{x_R} + (\,\dot{x} - l\dot{\theta}\cos(\alpha_i)\,)\,\overrightarrow{y_R}\,) \cdot \overrightarrow{v_i} \\ &= -\sin(\alpha_i)(\,\dot{x} + l\dot{\theta}\sin(\alpha_i)\,) + \cos(\alpha_i)(\,\dot{x} - l\dot{\theta}\cos(\alpha_i)\,) \\ &= -\dot{x}\sin(\alpha_i) + \dot{y}\cos(\alpha_i) - l\dot{\theta} \end{aligned} \tag{6}$$

2.2 Extension du modèle à un robot holonome à 3 roues

Le robot étudié est constitué d'un corps rigide central, auquel sont connectées 3 roues holonomes de telle sorte que $M_0M_1M_2$ forme un triangle équilatéral. On définit aussi le vecteur $l^{-1}\,\overrightarrow{OM_0}$ comme étant le vecteur unitaire pointant vers l'avant du robot. Il vient donc que :

$$\begin{cases} \alpha_0 &= 0 \\ \alpha_1 &= \frac{2\pi}{3} \\ \alpha_2 &= \frac{4\pi}{3} \end{cases} \tag{7}$$

On peut réécrire l'équation (6) pour $i \in [\![1,\ 3]\!]$ sous la forme matricielle suivante :

$$\begin{pmatrix} \omega_1 \\ \omega_2 \\ \omega_3 \end{pmatrix} = \frac{1}{r} \begin{pmatrix} -\sin(\alpha_1) & \cos(\alpha_1) & -l \\ -\sin(\alpha_2) & \cos(\alpha_2) & -l \\ -\sin(\alpha_3) & \cos(\alpha_3) & -l \end{pmatrix} \begin{pmatrix} \dot{x} \\ \dot{y} \\ \dot{\theta} \end{pmatrix} \tag{8}$$

En substituant les α_i par leur valeur, on obtient alors l'équation de la cinématique inverse du robot holonome à 3 roues :

$$\begin{pmatrix} \omega_1 \\ \omega_2 \\ \omega_3 \end{pmatrix} = \underbrace{\frac{-1}{2r} \begin{pmatrix} 0 & -2 & 2l \\ \sqrt{3} & 1 & 2l \\ -\sqrt{3} & 1 & 2l \end{pmatrix}}_{P :=} \begin{pmatrix} \dot{x} \\ \dot{y} \\ \dot{\theta} \end{pmatrix} \tag{9}$$

NB : $\dot{x}$ et $\dot{y}$ correspondent aux composantes de $\overrightarrow{V_{O \in \mathcal{R}/R_0}}$ dans le système de coordonnées de $\mathcal{R}_R$. Cela signifie que donner une commande $\dot{x} > 0$ revient en réalité à faire reculer le robot à une vitesse $\dot{x}$. De plus, z_0 et z_R étant confondus, une commande $\dot{\theta} > 0$ fait bien tourner le robot autour de son axe en sens trigonométrique à une vitesse angulaire $\dot{\theta}$.

2.3 Modèle cinématique direct du robot

Nous pouvons à présent commander le robot en lui donnant une commande en vitesses linéaire et angulaire, mais nous ne disposons pour l'instant d'aucune information de retour sur le déplacement effectif du robot.

Le déterminant de la matrice P calculée en (9) vaut $\frac{-3\sqrt{3}\,l}{2r^3}$. Celle-ci est donc inversible, et on peut inverser l'équation (9) pour obtenir l'équation de la cinématique directe du robot holonome à 3 roues :

$$\begin{pmatrix} \dot{x} \\ \dot{y} \\ \dot{\theta} \end{pmatrix} = \frac{-r}{3} \begin{pmatrix} 0 & \sqrt{3} & -\sqrt{3} \\ -2 & 1 & 1 \\ l^{-1} & l^{-1} & l^{-1} \end{pmatrix} \begin{pmatrix} \omega_1 \\ \omega_2 \\ \omega_3 \end{pmatrix} \tag{10}$$

NB : Les remarques faites sur les signes de $\dot{x}$, $\dot{y}$ et $\dot{\theta}$ s'appliquent également ici.

3 Implémentation ROS

Nous disposons à présent des équations (9) et (10), qui vont nous permettre d'implémenter la commande en vitesse du robot holonome à 3 roues avec ROS, d'abord en boucle ouverte, puis en boucle fermée.

3.1 Commande d'un moteur à courant continu

Afin de commander facilement le déplacement du robot, il est nécessaire de pouvoir contrôler individuellement la vitesse de rotation de chaque moteur. Dans notre cas, nous utilisons des moteurs à courant continu, dont la vitesse de rotation est proportionnelle à la tension entre ses deux bornes. L'interfaçage entre la Raspberry Pi du robot et chaque moteur DC se fait au moyen du shield *Adafruit DC & Stepper Motor HAT for Raspberry Pi - Mini Kit* (`https://www.adafruit.com/product/2348`).

Ce shield permet à la Raspberry Pi sur lequel il est monté de communiquer avec au maximum 4 moteurs DC (identifiés par un nombre entre 0 et 3) au moyen de ses GPIOs. La gestion de la tension délivrée aux moteurs est faite par le shield en par pulse width modulation (PWM).

Le contrôle d'un moteur DC se fait donc en 3 étapes :

- instanciation d'un objet de la classe `DC_motor` avec son identifiant
- actualisation de la vitesse de rotation du moteur par la méthode `DC_motor::set_speed`
- définition de la direction du sens de rotation du moteur avec la méthode `DC_motor::turn`

Le code ROS C++ correspondant à la classe `DC_motor` est le suivant :

```
enum Direction {FORWARD, BACKWARD};

class DC_motor {
  private:
    int id_, PWM_, IN2_, IN1_;

  public:
    DC_motor( int id ) {
        id_ = id;
        // each DC motor is controlled by a specific set of pins as specified by the HAT
        if ( id_ == 0 ) {PWM_ = 8; IN2_ = 9; IN1_ = 10;}
        else if ( id_ == 1 ) {PWM_ = 13; IN2_ = 12; IN1_ = 11;}
        else if ( id_ == 2 ) {PWM_ = 2; IN2_ = 3; IN1_ = 4;}
        else if ( id_ == 3 ) {PWM_ = 7; IN2_ = 6; IN1_ = 5;}
        else {PWM_ = 0; IN2_ = 0; IN1_ = 0;}
    }

    void turn( Direction dir ) {
        if ( dir == FORWARD ) {setPWM( IN1_, 4095, 0 ); setPWM( IN2_, 0, 4095 );}
        else if ( dir == BACKWARD ) {setPWM( IN1_, 0, 4095 ); setPWM( IN2_, 4095, 0 );}
```

```
        else {setPWM( IN1_, 0, 4095 ); setPWM( IN2_, 0, 4095 );}
    }

    void set_speed( int speed ) {
        speed = (speed < 0 ) ? 0 : ( speed > 4095 ) ? 4095 : speed;
        setPWM( PWM_, 0, speed );
    }

    void print_infos( float speed, Direction dir ) {
        ROS_INFO_STREAM( "motor " << id_ << " " <<
            PWM_ << " " << IN2_ << " " << IN1_ << " " << speed << " " << dir );
    }
};
```

3.2 Manipulations simultanées de 3 moteurs DC

Si le robot est techniquement commandable grâce à `DC_motor`, nous pouvons rajouter une couche logicielle pour monter en abstraction et en simplifier la commande. Ici, nous avons choisi de définir une classe `Motor_hat`, chargée de créer 4 instances de `DC_motor` correspondant aux 4 emplacements possibles (seuls les 3 premiers sont utilisés), d'initialiser la communication avec le shield, et de commencer à écouter le topic ROS `/cmd_vel_asserv`.

La réception d'un message sur ce topic déclenche l'exécution de la méthode `Motor_hat::callback`, qui convertit la commande en vitesse reçue en vitesses de rotation de chaque moteur selon l'équation (9) puis appelle la méthode `Motor_hat::turn_motor` pour chaque moteur. Cette méthode normalise la direction et la vitesse de rotation ainsi obtenues, puis les transmets au moteur via les méthodes de `DC_motor` associées.

Le code ROS C++ correspondant à la classe `Motor_Hat` est le suivant :

```
#define L 0.120
#define R 0.024
#define MAX_SPEED 4.8332

enum Motor {FRONT, LEFT, RIGHT};

class Motor_hat {
  private:
    ros::Subscriber sub_;
    std::vector<DC_motor> motors_;

    void turn_motor( Motor id, float w ) {
        DC_motor* motor;
        // each motor must be mapped with their specific location on the robot
        // to receive the corresponding velocity command
        motor = ( id == FRONT ) ? &motors_[1] : ( id == LEFT ) ? &motors_[0] : &motors_[2];
```

```
        Direction dir = ( w >= 0 ) ? FORWARD : BACKWARD;
        float speed = ( w < 0 ) ? -w : w;
        speed = ( speed > MAX_SPEED ) ? MAX_SPEED : speed;
        speed = 4096 * speed / (float)MAX_SPEED;

        motor->set_speed( speed );
        motor->turn( dir );
        motor->print_info( speed, dir );
    }

    void callback( const geometry_msgs::Twist &msg ) {
        ROS_INFO_STREAM( "cmd asserv " <<
            msg.linear.x << " " << msg.linear.y << " " << msg.linear.z << " " <<
            msg.angular.x << " " << msg.angular.y << " " << msg.angular.z );

        float x, y, w;
        x = msg.linear.x; y = msg.linear.y; w = msg.angular.z;

        float w1, w2, w3, cste = sqrt(3) * 0.5;
        w1 = ( y - L * w ) / R;
        w2 = ( -cste * x - 0.5 * y - L * w ) / R;
        w3 = ( cste * x - 0.5 * y - L * w ) / R;
        ROS_INFO_STREAM( "w " << w1 << " " << w2 << " " << w3 );

        turn_motor( FRONT, w1 );
        turn_motor( LEFT, w2 );
        turn_motor( RIGHT, w3 );
    }

  public:
    Motor_hat() {
        for ( unsigned int id = 0; id < 3; ++id ) {motors_.push_back( DC_motor( id ) );}
        initPWM( 0x60 ); setPWMFreq( 1600 );
        ros::NodeHandle nh;
        sub_ = nh.subscribe( "cmd_vel_asserv", 10, &Motor_hat::callback, this );
    }
};
```

3.3 Le driver de commande en vitesse en boucle ouverte

Il ne reste plus qu'à créer le noeud ROS du driver pour pouvoir commander le robot en boucle ouverte. Le code de ce noeud est très simple puisqu'il se contente d'initialiser le noeud et de créer une instance de `Motor_hat`.

Le code ROS C++ correspondant au driver en boucle ouverte est le suivant :

```
#include <ros/ros.h>

#include "motor_hat.hpp"

int main( int argc, char** argv ) {
    ros::init( argc, argv, "holonomic_3wheels_driver" );
    Motor_hat hat;
    ros::spin();
    return 0;
}
```

Il est désormais possible de commander le robot en vitesse depuis un terminal, par exemple avec la commande :

```
rostopic pub /cmd_vel_asserv geometry_msgs/Twist "linear:
    x: 0.1
    y: 0.1
    z: 0.0
angular:
    x: 0.0
    y: 0.0
    z: 0.02"
```

NB : Les vitesses linéaires sont exprimées en m/s et les vitesses angulaires en rad/s.

3.4 Ajout du retour d'odométrie par roues encodeuses

Les potentielles incertitudes de déplacement du robot en raison d'erreurs de modèle peuvent engendrer des différences entre la vitesse demandée et celle à laquelle se déplace réellement le robot. Pour corriger ceci, il est nécessaire d'introduire une boucle de retour d'information sur la rotation des moteurs grâce à des roues encodeuses.

Si le code correspondant à ce retour d'information est raisonnablement simple à imaginer en tant que symétrique de `Motor_hat` et `DC_motor`, celui-ci n'a toutefois pas été implémenté.

Celui-ci se décomposerait en une classe `Encoder` qui se chargerait de récupérer une information de nombre de crans (donc de variation angulaire) dont a tourné la roue à chaque appel d'une de ses méthodes, et d'une classe `Encoder_hat` qui utiliserait l'équation (10) pour en déduire une variation de position de la forme :

$$\begin{pmatrix} \Delta x \\ \Delta y \\ \Delta\theta \end{pmatrix} = \frac{-r}{3} \begin{pmatrix} 0 & \sqrt{3} & -\sqrt{3} \\ -2 & 1 & 1 \\ l^{-1} & l^{-1} & l^{-1} \end{pmatrix} \begin{pmatrix} \Delta\theta_1 \\ \Delta\theta_2 \\ \Delta\theta_3 \end{pmatrix} \tag{11}$$

Cette variation serait alors publié à intervalle régulier Δt par un attribut de `Encoder_hat` dans le topic `/odom`.

3.5 Ajout d'un correcteur PID

En supposant que nous disposons du code décrit précédemment, nous pouvons réaliser une boucle de contrôle de type PID en combinant la commande de vitesse et l'information de retour des encodeurs. Nous avons donc défini la classe `PID` de telle sorte qu'elle dispose d'un attribut écoutant le topic de commande en vitesse `/cmd_vel`, un autre écoutant le topic de retour d'information `/odom`, et d'une méthode calculant puis transmettant la commande asservie par PID au robot sur le topic `/cmd_vel_asserv`. De plus, nous sommes passé par une classe utilitaire `Vec3` afin de vectorialiser les équations du PID pour gagner en lisibilité.

Il est à noter que celui-ci repose sur l'approximation :

$$\begin{pmatrix} \dot{x} \\ \dot{y} \\ \dot{\theta} \end{pmatrix} \approx \frac{1}{\Delta t} \begin{pmatrix} \Delta x \\ \Delta y \\ \Delta \theta \end{pmatrix} \tag{12}$$

Cette approximation n'est acceptable que pour des valeurs suffisamment petites de Δt, c'est-à-dire à une fréquence d'acquisition des roues encodeuses suffisamment élevée.

Le code ROS C++ correspondant à la classe `PID` est le suivant :

```
#include <geometry_msgs/Twist.h>

class Vec3 {
  public:
    float x, y, z;

    Vec3() {x = 0.0; y = 0.0; z = 0.0;}
    Vec3( float x1, float y1, float z1 ) {x = x1; y = y1; z = z1;}
    Vec3( const Vec3 &vec ) {x = vec.x; y = vec.y; z = vec.z;}

    Vec3 operator+=( const Vec3 &vec ) {x += vec.x; y += vec.y; z += vec.z; return *this;}
    Vec3 operator-=( const Vec3 &vec ) {x -= vec.x; y -= vec.y; z -= vec.z; return *this;}
    Vec3 operator+( const Vec3 &vec ) {Vec3 v = *this; return v += vec;}
    Vec3 operator-( const Vec3 &vec ) {Vec3 v = *this; return v -= vec;}
    Vec3 operator*( float c ) {return Vec3( this->x * c, this->y * c, this->z * c );}
    Vec3 operator/( float c ) {return Vec3( this->x / c, this->y / c, this->z / c );}

    geometry_msgs::Twist Vec2Twist() {
        geometry_msgs::Twist twist;
        twist.linear.x = this->x; twist.linear.y = this->y; twist.angular.z = this->z;
        return twist;
    }
};

Vec3 Twist2Vec( const geometry_msgs::Twist &twist ) {
```

```
    return Vec3( twist.linear.x, twist.linear.y, twist.angular.z );
}

class PID {
  private:
    float Kp_, Ki_, Kd_, dt_;
    Vec3 integral_, prev_err_;
    ros::Subscriber cmd_sub_, odom_sub_;
    ros::Publisher pub_;
    geometry_msgs::Twist cmd, odom;

    void cmd_callback( const geometry_msgs::Twist &msg ) {
        ROS_INFO_STREAM( "cmd " <<
            msg.linear.x << " " << msg.linear.y << " " << msg.linear.z << " " <<
            msg.angular.x << " " << msg.angular.y << " " << msg.angular.z );
        cmd = msg;
    }

    void odom_callback( const geometry_msgs::Twist &msg ) {
        ROS_INFO_STREAM( "odom " <<
            msg.linear.x << " " << msg.linear.y << " " << msg.linear.z << " " <<
            msg.angular.x << " " << msg.angular.y << " " << msg.angular.z );
        odom = msg;
    }

  public:
    PID( float Kp, float Ki, float Kd, float dt ) {
        Kp_ = Kp; Ki_ = Ki; Kd_ = Kd; dt_ = dt;
        integral_ = Vec3(); prev_err_ = Vec3();

        ros::NodeHandle nh;
        cmd_sub_ = nh.subscribe( "cmd_vel", 10, &PID::cmd_callback, this );
        odom_sub_ = nh.subscribe( "odom", 10, &PID::odom_callback, this );
        pub_ = nh.advertise<geometry_msgs::Twist>( "cmd_vel_asserv", 10 );
    }

    void compute() {
        Vec3 err = Twist2Vec( cmd ) - Twist2Vec( odom ) / dt_;
        Vec3 derivative;

        integral_ += err * dt_;
        derivative = ( err - prev_err_ ) / dt_;
```

```
        Vec3 out = err * Kp_ + integral_ * Ki_ + derivative * Kd_;
        prev_err_ = err;

        geometry_msgs::Twist cmd_corr = ( Twist2Vec( cmd ) + out ).Vec2Twist();
        pub_.publish( cmd_corr );
    }
};
```

3.6 Le driver en boucle fermée

Nous pouvons reprendre le code du driver en boucle ouverte et le modifier afin d'obtenir celui du driver en boucle fermée. En effet, il s'agit principalement d'instancier des objets de type `Encoder_hat` et `PID`, et de s'assurer que le calcul de la commande asservie s'effectue bien à chaque pas de temps.

Le code ROS C++ correspondant au driver en boucle fermée est le suivant :

```
#include <ros/ros.h>

#include "motor_hat.hpp"
#include "encoder_hat.hpp"
#include "PID.hpp"

int main( int argc, char** argv ) {
    float dt = 0.1;

    ros::init( argc, argv, "holonomic_3wheels_driver" );
    Motor_hat mhat;
    Encoder_hat ehat;
    PID pid( 0.8, 0.1, 0.1, dt );
    ros::Rate r( 1.0 / dt );

    while( ros::ok() ) {
        mhat.compute(); // computing and publishing odometry
        ros::spinOnce(); // listening to topics and executing callbacks
        pid.compute(); // computing PID speed correction (which will be processed next time)
        r.sleep();
    }
    return 0;
}
```

La commande asservie en vitesse du robot est à présent possible depuis un terminal, par exemple avec la commande :

```
rostopic pub /cmd_vel geometry_msgs/Twist "linear:
```

```
    x: 0.1
    y: 0.1
    z: 0.0
angular:
    x: 0.0
    y: 0.0
    z: 0.02"
```

Le driver peut également être inclus dans un projet plus large ayant automatisé la transmission de la commande en vitesse dans `/cmd_vel`.

NB : Les vitesses linéaires sont exprimées en m/s et les vitesses angulaires en rad/s.

4 Annexes

Les fonctions concernant la PWM sont définies dans les fichiers `PWM.h` et `PWM.c`, dont les contenus sont les suivants :

4.1 PWM.h

```
#pragma once

#include <wiringPiI2C.h>
#include <time.h>
#include <math.h>

// Copied from https://github.com/4ndr3w/PiBot . Many thanks!

enum PWMConstant {
    __MODE1 = 0x00,
    __MODE2 = 0x01,
    __SUBADR1 = 0x02,
    __SUBADR2 = 0x03,
    __SUBADR3 = 0x04,
    __PRESCALE = 0xFE,
    __LED0_ON_L = 0x06,
    __LED0_ON_H = 0x07,
    __LED0_OFF_L = 0x08,
    __LED0_OFF_H = 0x09,
    __ALL_LED_ON_L = 0xFA,
    __ALL_LED_ON_H = 0xFB,
    __ALL_LED_OFF_L = 0xFC,
    __ALL_LED_OFF_H = 0xFD,

    __RESTART = 0x80,
    __SLEEP = 0x10,

    __ALLCALL = 0x01,
    __INVRT = 0x10,
    __OUTDRV = 0x04,
};

void initPWM(int address = 0x40);
void setPWMFreq(int freq);
void setPWM(int channel, int on, int off);
void setAllPWM(int on, int off);
```

4.2 PWM.c

```
#include "PWM.h"

int pwmHatFD = -1;

// Copied from https://github.com/4ndr3w/PiBot . Many thanks!

void initPWM(int address)
{
    pwmHatFD = wiringPiI2CSetup(address);


    // zero all PWM ports
    setAllPWM(0,0);

    wiringPiI2CWriteReg8(pwmHatFD, __MODE2, __OUTDRV);
    wiringPiI2CWriteReg8(pwmHatFD, __MODE1, __ALLCALL);

    int mode1 = wiringPiI2CReadReg8(pwmHatFD, __MODE1);
    mode1 = mode1 & ~__SLEEP;
    wiringPiI2CWriteReg8(pwmHatFD, __MODE1, mode1);

    setPWMFreq(60);
}

void setPWMFreq(int freq)
{
    float prescaleval = 25000000;
    prescaleval /= 4096.0;
    prescaleval /= (float)freq;
    prescaleval -= 1.0;
    int prescale = floor(prescaleval + 0.5);

    int oldmode = wiringPiI2CReadReg8(pwmHatFD, __MODE1);
    int newmode = (oldmode & 0x7F) | 0x10;
    wiringPiI2CWriteReg8(pwmHatFD, __MODE1, newmode);
    wiringPiI2CWriteReg8(pwmHatFD, __PRESCALE, floor(prescale));
    wiringPiI2CWriteReg8(pwmHatFD, __MODE1, oldmode);

    wiringPiI2CWriteReg8(pwmHatFD, __MODE1, oldmode | 0x80);
}

void setPWM(int channel, int on, int off)
```

```
{
    wiringPiI2CWriteReg8(pwmHatFD, __LED0_ON_L+4*channel, on & 0xFF);
    wiringPiI2CWriteReg8(pwmHatFD, __LED0_ON_H+4*channel, on >> 8);
    wiringPiI2CWriteReg8(pwmHatFD, __LED0_OFF_L+4*channel, off & 0xFF);
    wiringPiI2CWriteReg8(pwmHatFD, __LED0_OFF_H+4*channel, off >> 8);
}

void setAllPWM(int on, int off)
{
    wiringPiI2CWriteReg8(pwmHatFD, __ALL_LED_ON_L, on & 0xFF);
    wiringPiI2CWriteReg8(pwmHatFD, __ALL_LED_ON_H, on >> 8);
    wiringPiI2CWriteReg8(pwmHatFD, __ALL_LED_OFF_L, off & 0xFF);
    wiringPiI2CWriteReg8(pwmHatFD, __ALL_LED_OFF_H, off >> 8);
}
```