## 1.3 Partie C - Localisation I : encodeurs et IMU

Pour cette dernière partie du laboratoire #1, nous allons maintenant utiliser les fonctionnalités développées précédemment dans le but de faire un premier pas vers des algorithmes de localisation du robot. Il s'agira plus particulièrement de vous amener à appliquer les connaissances théoriques acquises durant le cours afin de développer le modèle cinématique du robot utilisé au laboratoire, qui possède une morphologie de robot à action différentielle. L'objectif principal sera de vous amener à pouvoir localiser le robot $(x, y, \theta)$ automatiquement dans un repère inertiel.

### 1.3.1 Objectifs académiques

L'objectif de ce laboratoire est d'abord de revisiter les équations cinématiques d'un robot à action différentielle. Une fois ces équations bien comprises et les paramètres fonctionnels du robot identifiés, il s'agira d'implanter les équations dans un nœud ROS qui nous permettra par la suite de localiser le robot dans un repère inertiel. Plus particulièrement, les objectifs spécifiques reliés à cette séance de laboratoire sont :

- Développer les équations cinématiques du robot sous forme discrète [17] et les implanter dans un *node* ROS en Python ;
- Approfondir les notions de méthodes de programmation orientées-objet sous Python et comprendre comment imbriquer des *subscribers* dans une classe ;
- Programmer des méthodes dans une classe Python. Certaines de ces méthodes agiront comme *callbacks* lorsque des messages seront reçus ;
- Se familiariser avec l'utilisation de messages plus complexes que les « std_msgs » (e.g. : String ou Int32) vu précédemment, tels que les *geometry_msgs* et plus particulièrement le message de type *TransformStamped* ;
- Apprendre à se servir d'un *broadcaster* et à s'en servir pour visualiser des repères dans l'utilitaire rviz ;
- Évaluer le modèle cinématique développé par l'entremise d'expériences simples de navigation en boucle ouverte, afin de constater l'efficacité mais aussi les limites d'une localisation basée entièrement sur les encodeurs.

### 1.3.2 Jalon 1 : Rappel et développement des équations cinématiques pour une implantation sur le robot

Avant de commencer le travail de programmation, il faut d'abord rappeler les équations cinématiques d'un robot à action différentielle. Un tel robot, accompagné des repères pertinents à notre analyse, est représenté à la fig. 1.18.

Premièrement, les distances parcourues par les roues sont données par les équations 1.1 :

$$D_G = (R + l)\theta \quad D_D = (R)\theta, \tag{1.1}$$

17. Ici, la notion de « forme discrète » concerne surtout les intégrales propres aux équations cinématiques. Pour intégrer sur un intervale de temps fini, il faudra recourir à une méthode numérique simple servant à approximer les intégrales : la méthode d'Euler (*forward Euler method*).

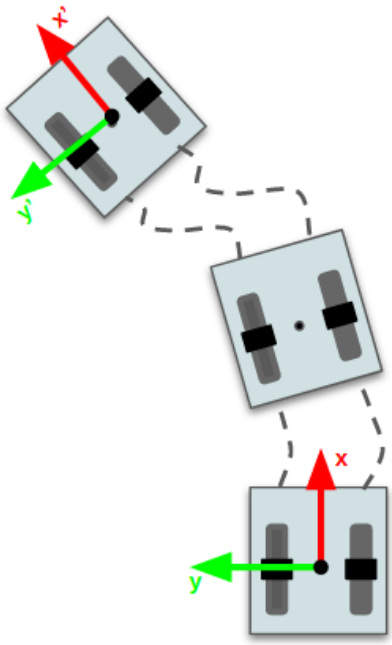


**Figure 1.18:** *Un robot à action différentielle se déplaçant dans un plan. Le repère inertiel (axes x et y) est confondu avec le repère du robot (axes x' et y') au départ. La pose du robot mobile après un déplacement est exprimée en fonction du repère inertiel.*

où $D_G$ et $D_D$ sont les distances parcourues par les roues gauche et droite respectivement. $R$ est le rayon instantané de courbure en mètre, tel que mesuré par rapport à la roue intérieure à la rotation, tandis que $l$ est la distance entre les deux roues en mètre. Grâce aux encodeurs à effet hall disposés sur les moteurs du robot, nous pourrons directement avoir accès à l'information de distance parcourue, en connaissant le nombre d'impulsions émises par tour de roue et le diamètre des roues. L'orientation du repère du robot (en radian) par rapport au repère inertiel est donné par l'éq. 1.2 :

$$\theta = \frac{D_G - D_D}{l}. \tag{1.2}$$

Ainsi, la vitesse de rotation en radian/sec est simplement cette orientation divisée par le temps, d'où l'équation1.3 :

$$\omega = \frac{D_G - D_D}{l \cdot t}, \tag{1.3}$$

où $\omega$ est la vitesse en radian par seconde. Or, puisque la distance parcourue en mètre divisée par le temps donne la vitesse linéaire, on peut déterminer la vitesse linéaire de chaque roue à partir de leur distance parcourue et réécrire l'éq. 1.3 :

$$\omega = \frac{v_G - v_D}{l}, \tag{1.4}$$

où $v_G$ et $v_D$ sont respectivement les vitesses des roues gauche et droite en m/sec. Enfin, la vitesse linéaire du repère associé au robot mobile est simplement la moyenne de la vitesse des deux roues :

$$v = \frac{v_G + v_D}{2}. \tag{1.5}$$

Une fois que l'on connaît les expressions de $v$ et de $\omega$, il s'agit simplement de résoudre les intégrales suivantes pour quantifier le déplacement en translation et en rotation :

$$\Delta x = \int_0^t v(t)\cos(\theta(t))dt \tag{1.6}$$

$$\Delta y = \int_0^t v(t)\sin(\theta(t))dt \tag{1.7}$$

$$\Delta \theta = \int_0^t \omega(t)dt \tag{1.8}$$

### 1.3.3 Jalon 2 : Implantation en format discret sur le robot

On vous demande maintenant d'écrire un nouveau *node* Python, à l'intérieur du *package* « odometrie » (que vous devrez également créer), et que vous nommerez « localisation.py ». On vous demande plus particulièrement de doter ce *node* des caractéristiques suivantes :

- Il devra utiliser le concept de classe en Python afin de tirer profit de cette fonctionnalité des langages de programmation orientés-objet ;
- S'abonner aux *topics* des encodeurs afin de recevoir leur données ;
- Résoudre les équations cinématiques du robot ;
- Utiliser la fonction de « *broadcasting* » afin de diffuser les données de pose issues de la cinématique.

#### Informations supplémentaires : discrétisation des équations cinématiques

Les équations 1.6 à 1.8 comportent des intégrales qui seront évaluées sur un horizon de temps fini et à répétition. Ainsi, il est nécessaire de discrétiser ces intégrales afin de pouvoir les implanter dans le *node* écrit en Python. Pour ce faire, nous utiliserons l'une des approximations numériques les plus simples en ce concerne l'approximation d'intégrales : la méthode d'Euler vers l'avant (*forward Euler Method*) [18]. En discrétisant les équations 1.6 à 1.8 grâce à la méthode d'Euler vers l'avant, on se retrouvera avec les équations suivantes :

$$x_{k+1} = x_k + v_k \cdot T_s \cdot \cos\theta_k, \tag{1.9}$$

$$y_{k+1} = y_k + v_k \cdot T_s \cdot \sin\theta_k, \tag{1.10}$$

$$\theta_{k+1} = \theta_k + \omega_k \cdot T_s, \tag{1.11}$$

où $k$ fait référence à un index identifiant la période de temps ($k = 0, 1, 2, ...$) et $T_s$ est la période de temps, en seconde, à laquelle nous mettrons à jour la pose du robot. Les

18. Pour obtenir plus de détails concernant cette méthode numérique d'intégration, vous pouvez consulter le lien suivant.

équations 1.9 à 1.11 seront des équations à programmer directement dans votre *node*. Cependant, pour résoudre ces équations, il faudra aussi programmer les expressions propres à la vitesse linéaire et angulaire et ce, à partir d'informations connues. Ainsi, les expressions de vitesse linéaire et angulaire, en format discrétisé, sont données par :

$$v_k = \frac{\Delta s}{T_s}, \tag{1.12}$$

$$\omega_k = \frac{\Delta \theta}{T_s}, \tag{1.13}$$

où $T_s = t_{k+1} - t_k$. $\Delta s$ et $\Delta \theta$ sont respectivement la distance parcourue (en mètre) et l'angle décrit (en radian) par le robot durant une période de temps donnée. Ces deux éléments peuvent quant à eux s'exprimer par :

$$\Delta s = \frac{(\Delta D_G + \Delta D_D)}{2} \quad \text{et} \tag{1.14}$$

$$\Delta \theta = \frac{(\Delta D_D - \Delta D_G)}{l}, \tag{1.15}$$

où $\Delta D_G$ et $\Delta D_D$ sont respectivement les distances parcourues par la roue gauche et la roue droite durant une période de temps $T_s$. Finalement, en ce qui a trait à ces distances parcourues, celles-ci peuvent s'exprimer en fonction du nombre d'impulsion des encodeurs :

$$\Delta D_G = 2\pi r\left(\frac{\Delta \text{enc\_a}}{\text{enc\_tour}}\right) \quad \Delta D_D = 2\pi r\left(\frac{\Delta \text{enc\_b}}{\text{enc\_tour}}\right), \tag{1.16}$$

où $\Delta$enc_a=(enc_a$_{k+1}$ − enc_a$_k$) et $\Delta$enc_b=(enc_b$_{k+1}$ − enc_b$_k$) représentent respectivement le nombre d'impulsion(s) d'encodeur reçus au niveau de la roue gauche et de la roue droite durant une période de temps $T_s$. enc_tour fait quant à elle référence au nombre d'impulsions totales reçues pour une seule rotation complète de la roue. Toutes ces équations devront être implantées dans le *node* « localisation.py ». Notez cependant qu'il est possible de faire plusieurs substitutions afin de rendre le code résultant plus compact, au besoin.

**Table 1.1:** *Paramètres cinématiques des robots mobiles du cours GPA778.*

| Élément | Paramètre | Valeur |
|---|---|---|
| Roues | rayon ($r$) | 3.575 cm |
| Roues | distance $l$ | 16.55 cm |
| Encodeurs | enc_tour | 158.2 impulsions/tour |
| Encodeur A | roue associée | gauche |
| Encodeur B | roue associée | droite |
| Encodeur A | polarité | décrémente lorsque robot se déplace vers l'avant |
| Encodeur B | polarité | incrémente lorsque robot se déplace vers l'avant |

**Informations supplémentaires : paramètres cinématiques du robot**

Le tableau 1.1 présente les paramètres cinématiques des robots mobiles du cours. Ces paramètres seront nécessaires à l'écriture de votre code.

**Informations supplémentaires : pseudocode**

Comme dans le cas du langage C++, le langage Python supporte aussi la programmation orientée-objet et la définition de classes. Les classes, dans le contexte de la programmation orientée-objet, permettent de lier les variables aux fonctions. Dans le cadre de ce laboratoire, si vous n'êtes pas confortable avec le concept de classe, ou encore si vous aimeriez en apprendre plus sur les classes lorsqu'il s'agit du langage Python, on vous invite à consulter les liens ci-dessous :

- http://www.xavierdupre.fr/app/teachpyx/helpsphinx/c_classes/classes.html.
  - Il s'agit d'une excellente référence (en français), de la part du professeur Xavier Dupré de l'École nationale de la statistique et de l'administration économique Paris (ENSAE Paris).
- https://docs.python.org/2.7/tutorial/classes.html.
  - Il s'agit de la documentation officielle de Python portant sur les classes (en anglais).
- https://www.w3schools.com/python/python_classes.asp.
  - Cette page provient de W3Schools, un site web destiné entre autres à l'apprentissage de langages de programmation.

Pour accomplir votre tâche, on met à votre disposition le pseudocode qui se trouve à la page suivante, pseudocode qu'il faudra maintenant compléter. À la suite de ce pseudocode se trouve quelques explications supplémentaires quant aux éléments de programmation qu'il contient.

```
#!/usr/bin/env python3

import numpy as np
import time
import rclpy
from rclpy.node import Node
from std_msgs.msg import Int32
from geometry_msgs.msg import TransformStamped

# Importations pour les matrices de transformations homogenes
from scipy.spatial.transform import Rotation as R
from tf2_ros import TransformBroadcaster


class Locator(Node):
    __nb_enc_tours = 158.2
    __wheel_radius_dg = 0.035
    __wheel_radius_dd = 0.03578
    __wheel_distance = 0.17825

    def __init__(self):
        super().__init__('localisation')
        # Frequence : 10.0 Hz
        self.loop_rate = 10.0  # Nous allons utiliser un minuteur (timer) ROS2 a 10Hz

        # Donnees de temps / pose
        self.prev_time = None # A initialiser lors du premier appel
        self.robot_pose = TransformStamped()

        # Diffuseur TF
        self.br = TransformBroadcaster(self)

        # Pose initiale
        self.robot_pose.header.frame_id = 'odom'
        self.robot_pose.child_frame_id = 'r_robot'
        self.robot_pose.transform.translation.x = 0.0
        self.robot_pose.transform.translation.y = 0.0
        self.robot_pose.transform.translation.z = 0.0
        self.robot_theta = 0.0

        r = R.from_euler('xyz', [0, 0, self.robot_theta], degrees=False)
        q = r.as_quat()

        self.robot_pose.transform.rotation.x = q[0]
        self.robot_pose.transform.rotation.y = q[1]
        self.robot_pose.transform.rotation.z = q[2]
        self.robot_pose.transform.rotation.w = q[3]

        # Donnees provenant des encodeurs
        self.enca_count = 0
        self.encb_count = 0
        self.enca_count_ref = 0
        self.encb_count_ref = 0
```

```
        # Creation des subscriber
        self.enc_sub_A = self.create_subscription(
            Int32,
            'encodeur_A',
            self.callbackenca,
            10
        )
        self.enc_sub_B = self.create_subscription(
            Int32,
            'encodeur_B',
            self.callbackencb,
            10
        )

        # Creation d'un minuteur a ~10 Hz
        self.timer_ = self.create_timer(1.0 / self.loop_rate, self.timer_callback)

        self.get_logger().info("Le noeud Locator a demarre.")

    def callbackenca(self, msg: Int32):
        # Inversion de polarite
        self.enca_count = -msg.data

    def callbackencb(self, msg: Int32):
        self.encb_count = msg.data

    def solve_kinematics(self):
        """
        Nous allons calculer delta_dg, delta_dd, delta_s, etc. et mettre a jour la transformation.
        """
        current_time = self.get_clock().now().seconds_nanoseconds()[0] # uniquement la partie en secondes
        if self.prev_time is None:
            # Premiere iteration, on initialise simplement prev_time
            self.prev_time = current_time
            return

        T = current_time - self.prev_time # dt en secondes
        if T <= 0:
            return

        -resoudre la cinematique ici , ex: delta_dg = 2 * np.pi * self ._wheel_radius * delta_enc_a / self._nb_enc_tours
-mettre a jour la pose du robot: time stamp, translation et quaternion, ex:

        # Deplacement lineaire moyen
        delta_s = ...
        # Variation d'angle pour un differentiel
        delta_l = ...

        # Vitesses lineaire et angulaire
        v_lineaire = delta_s / T
        v_angulaire = delta_l / T

        # Pose actuelle
        x = self.robot_pose.transform.translation.x
        y = self.robot_pose.transform.translation.y
        theta = self.robot_theta
```

```
        # Nouvelle pose apres les dernieres T secondes
        new_x = ...
        new_y = ...
        new_theta = ...

        # Mise a jour des references
        self.enca_count_ref = ...
        self.encb_count_ref = ...
        self.prev_time = ...

        # Mise a jour de la transformation du robot
        self.robot_pose.header.stamp = self.get_clock().now().to_msg()
        self.robot_pose.transform.translation.x = new_x
        self.robot_pose.transform.translation.y = new_y
        self.robot_theta = new_theta

        r = R.from_euler('xyz', [0.0, 0.0, self.robot_theta], degrees=False)
        q = r.as_quat()

        self.robot_pose.transform.rotation.x = q[0]
        self.robot_pose.transform.rotation.y = q[1]
        self.robot_pose.transform.rotation.z = q[2]
        self.robot_pose.transform.rotation.w = q[3]

        self.br.sendTransform(self.robot_pose)

        # Afficher la pose dans le terminal
        self.get_logger().info(
            f"x: {self.robot_pose.transform.translation.x:.3f}, "
            f"y: {self.robot_pose.transform.translation.y:.3f}, "
            f"theta (deg): {np.rad2deg(self.robot_theta):.3f}"
        )

    def timer_callback(self):
        # Appele a ~10Hz
        self.solve_kinematics()


def main(args=None):
    rclpy.init(args=args)
    node = Locator()
    try:
        rclpy.spin(node)
    except KeyboardInterrupt:
        pass
    finally:
        node.destroy_node()
        rclpy.shutdown()

if __name__ == '__main__':
    main()
```

En analysant le code fournit, on constate la définition d'une classe « Locator » qui sera chargée de plusieurs rôles dont la réception des données provenant des encodeurs, le calcul de la cinématique du robot et la diffusion des résultats par *broadcasting* et par impression dans le terminal.

Les quatres premières variables de la classe (on les appelle « attributs » de la classe) concernent les paramètres cinématiques du robot et sont précédées d'un tiret bas. Ceci

est une convention [19] indiquant que ces variables particulières devraient être réservées pour un usage interne à la classe seulement.

Ensuite, la première méthode de la classe « ___init___(self) » est le constructeur de la classe. Cette méthode particulière sera évoquée automatiquement à chaque fois qu'une nouvelle instance de la classe sera créée. Dans ce constructeur, on initialise différentes variables importantes relativement au calcul de la cinématique du robot. Vous pourrez constater que ces variables sont toutes précédées du préfixe « self. ». Ce préfixe permet de faire référence à la classe elle même, ainsi, l'utilisation du préfixe « self. » permet d'accéder à tous les attributs et méthodes de la classe et ce, à partir d'elle-même. Ainsi, les variables créées dans le constructeur ayant le préfixe « self. » sont des variables de la classe, et non pas des variables locales au constructeur seulement. Dans ce constructeur, on définit ainsi une fréquence de boucle ROS de 10 Hz, ce qui sera une fréquence de rafraîchissement suffisante pour notre application. On crée aussi une transformation entre le repère « map » et le repère « r_robot » que l'on initialise à $\{0, 0, 0\}$ en position et à $\{0, 0, 0, 1\}$ au niveau du quaternion définissant son orientation. Le repère « odom » est un repère automatiquement généré par ROS et son utilitaire *rviz*, tandis que le repère « r_robot » est un nom arbitraire que nous donnons au repère du robot, dont l'origine est située à mi-chemin entre les roues avant du robot.

Vous pourrez également constater que le constructeur créé deux *subscribers*, un pour chaque encodeur. Ainsi, lorsque l'on créera une nouvelle instance de la classe *Locator*, cette instance s'abonnera automatiquement aux *topics* des encodeurs et, lorsque des données seront reçues, ce seront ses propres méthodes de type *callback* qui seront évoqués. Cette façon de faire est une manière élégante de gérer la réception de données provenant de *topics* et de les rendre disponibles aux autres méthodes de la classe, sans devoir les passer en paramètres ou en créant des variables globales. En effet, cela est dû au fait que les variables « self.enca_count » et « self.encb_count » recevant les données publiées par les *topics* sont des attributs de la classe *Locator*. Enfin, la méthode « solve_kinematics() » sera elle aussi automatiquement appelée lors de l'expiration de la minuterie configurée avec une cadence de 10 Hz.

### 1.3.4 Jalon 3 : Validation

Pour tester que votre *node* « localisation » fonctionne, modifiez d'abord le script *main.py* écrit au dernier laboratoire de telle sorte qu'il puisse émettre une consigne de vitesse constante mais différente sur les deux moteurs (e.g. : 720 pour le moteur gauche et 760 pour le moteur droit). Assurez-vous également de faire en sorte que les moteurs tournent seulement sur un horizon de temps fini (e.g. : 4-5 secondes), ou encore de pouvoir manuellement arrêter les moteurs lorsque vous cesserez le programme *main* en appuyant sur les touches ctrl+c. **Testez votre programme *main* autant de fois que nécessaire, afin de vous assurez que le robot pourra être arrêté de façon**

19. Il s'agit simplement d'une convention, c'est-à-dire une indication donnée aux programmeurs qui pourraient consulter ce code à l'effet que ces variables devraient généralement être réservées à un usage interne à la classe seulement. Par contre, si vous enfreignez cette convention, aucune erreur ne sera émise par l'interpréteur Python.

**contrôlée pour ainsi éviter tout risque d'endommagement**.

Une fois les tests nécessaires effectués, placez le robot sur le repère inertiel du laboratoire en prenant soin de bien faire confondre l'origine des deux repères (inertiel et robot) et de lui donner la bonne orientation. Lancez ensuite tous les *nodes* nécessaires et dans cet ordre :

1. Le *node* « encodeurs » du *package* capteurs_odometrie.
2. Le *node* « localisation » du *package* odometrie.
3. Enfin, le *node* « main » du *package* navigation. On lance ce node en dernier, pour s'assurer que les algoritmes de localisation soient en fonction avant qu'un déplacement du robot n'ait lieu. Plus tard dans la session, nous automatiserons cette séquence de démarrage pour tous ces éléments, en plus d'ajouter d'autres nodes à notre environnement.

Vérifier les informations de pose obtenues, et comparer ces résultats à une validation manuelle à l'aide d'un ruban à mesurer.

### 1.3.5 Questions supplémentaires

1. Alors que vos *nodes* « encodeurs » et « localisation » sont en fonction, connectez-vous au robot par vnc et ouvrez un terminal dans cet environnement. Dans ledit terminal, lancez l'utilitaire *rviz* en tapant la commande « ros2 run rviz2 rviz2 » dans le terminal. Une fois que l'application *rviz* apparaît, cliquez sur « add » et, dans le menu déroulant qui apparaîtra, sélectionnez « TF ». Dans la fenêtre de gauche, développez la structure de l'élément TF que vous venez d'ajouter et dans le sous-menu « Frames », sélectionnez le repère « r_robot » et « odom ». Utilisez enfin votre script *main* pour déplacer quelque peu le robot et arrêtez-le. Prenez une capture d'écran et ajoutez-là à votre rapport. Vous devriez obtenir un résultat semblable à la fig. 1.19.

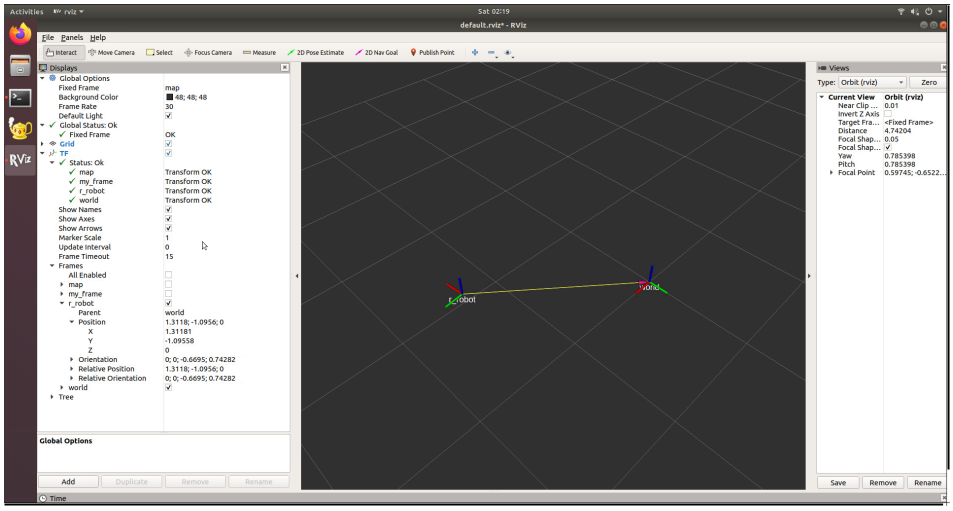


**Figure 1.19:** *Pose du repère « r_robot », exprimée par rapport au repère « world » dans rviz. L'exemple illustré utilise le repère « world » or, au laboratoire, ce sera par rapport au repère « odom » qu'il faudra exprimer la pose.*

2. Selon vous, est-ce que l'algorithme de localisation que vous venez de développer permet de localiser précisément le robot par rapport au repère inertiel défini au laboratoire ? Discutez de la précision de votre algorithme, en identifiant, notamment, les différentes sources d'erreur pouvant affecter votre estimation de localisation.

### 1.3.6 Démonstration au (à la) chargé·e de laboratoire

Pour ce laboratoire, il faudra simplement valider le fonctionnement de votre algorithme de localisation auprès du (de la) chargé·e de laboratoire.