SYSMER 2A

Lucas CARPENTIER, Thibault COURTOIS

22 décembre 2023

Développement Robotique sous ROS : contrôle du robot TurtleBot 2

Professeur encadrant :
Vincent HUGEL

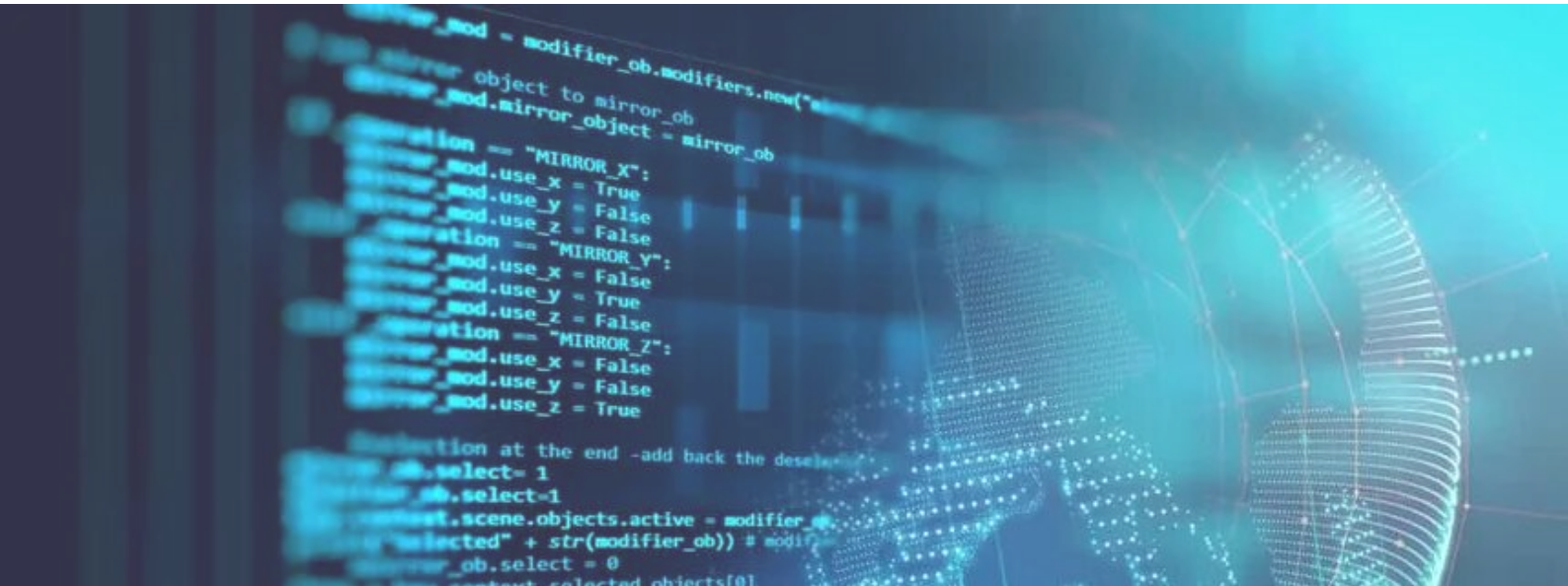

Table des matières

Introduction

Robot Operating System, plus communément appelé ROS, est un cadre de développement open source facilitant la création de logiciels pour les robots. Sa structure permet la communication entre différents nœuds logiciels, simplifiant ainsi le développement de systèmes robotiques complexes. Avec des outils de simulation, de débugage et une large communauté, ROS est largement utilisé en recherche, éducation et industrie pour accélérer le développement et la validation d'applications robotiques. Durant ces travaux pratiques, nous avons pu simuler un robot TurtleBot en utilisant le simulateur Gazebo, le contrôler à l'aide d'un joystick.

Dans ROS, la communication entre les nœuds se fait via des "topics". Ceux-ci sont des canaux de communication où les nœuds peuvent publier des messages ou s'abonner pour recevoir des messages spécifiques. Un nœud peut publier des données sur un "topic" donné, et d'autres nœuds intéressés par ces données peuvent s'y abonner pour les recevoir. Cela permet une communication asynchrone entre les différents composants du système robotique, facilitant la modularité et la flexibilité du développement. En résumé, les 'topics' constituent un élément clé de la communication entre les nœuds au sein de l'écosystème ROS, favorisant une collaboration efficace.

Au cours de nos travaux pratiques, nous avons mis en œuvre ROS en simulant le TurtleBot 2 dans l'environnement virtuel de Gazebo. Le TurtleBot 2, développé par Clearpath Robotics, est une plateforme robotique mobile polyvalente. Il est équipé de capteurs, dont des caméras et des capteurs de distance, permettant une perception précise de son environnement. Contrôlé à l'aide d'un joystick grâce à la librairie Rospy en Python, notre objectif était de tester et d'optimiser les algorithmes de contrôle du TurtleBot 2. Cette simulation préliminaire a posé les bases d'une transition vers des essais réels. Le TurtleBot 2, en tant que plateforme intégrée à ROS, offre une solution accessible et modulaire pour explorer les applications robotiques dans des contextes variés.

Ce rapport documente ainsi cette première approche du middleware ROS via l'utilisation du TurtleBot 2.

1 Implémentation du contrôle du robot

1.1 Environnement de simulation

Pour mettre en œuvre le contrôle du robot, nous avons initialement testé notre code localement à l'aide du simulateur Gazebo. Ce simulateur crée une carte avec des obstacles, permettant le contrôle du robot. L'objectif est de garantir le bon fonctionnement des commandes avant de les tester sur le robot réel. Le déplacement du robot a été effectué à l'aide d'un joystick de manette de jeu vidéo, dont les commandes sont traitées par le module joy4control. Nous avons utilisé plusieurs terminaux pour cela :

- Un terminal pour démarrer le "master", un processus central facilitant la communication entre les nœuds logiciels dans le système robotique.
- Un terminal pour exécuter le programme de gestion du joystick.
- Un terminal pour enregistrer les données nécessaires.

Pour le contrôle du robot en conditions réelles, nous avons vérifié la configuration correcte de l'environnement du shell. En utilisant la commande **echo $ROS_MASTER_URI**, nous avons confirmé que l'adresse IP du TurtleBot était correcte (10.0.1.101), garantissant ainsi la connectivité au réseau approprié. De plus, nous avons lancé un nouvel interpréteur de commandes dans chaque terminal avec **bash**, permettant l'interprétation des commandes de l'utilisateur, particulièrement dans des environnements non graphiques. Le reste du contrôle est semblable à la simulation en local telle que l'utilisation de la manette et de joy4control

1.2 Machine à états

Pour implémenter le contrôle du TurtleBot, nous avons utilisé une machine à état finis. Cela offre l'avantage d'être facilement compréhensible, facilement représentable mais également de simplifier le script du code. Nous utilisons alors l'orienté objet de Python pour structurer le code et cela évite de devoir recourir à des structures moins "élégantes" telle que les conditions if/elif/else ou les match. L'automate est alors défini par l'ensemble des états possibles et l'ensemble des transitions qu'il peut effectuer d'un état vers un autre. Voici un schéma de notre machine à états finis :

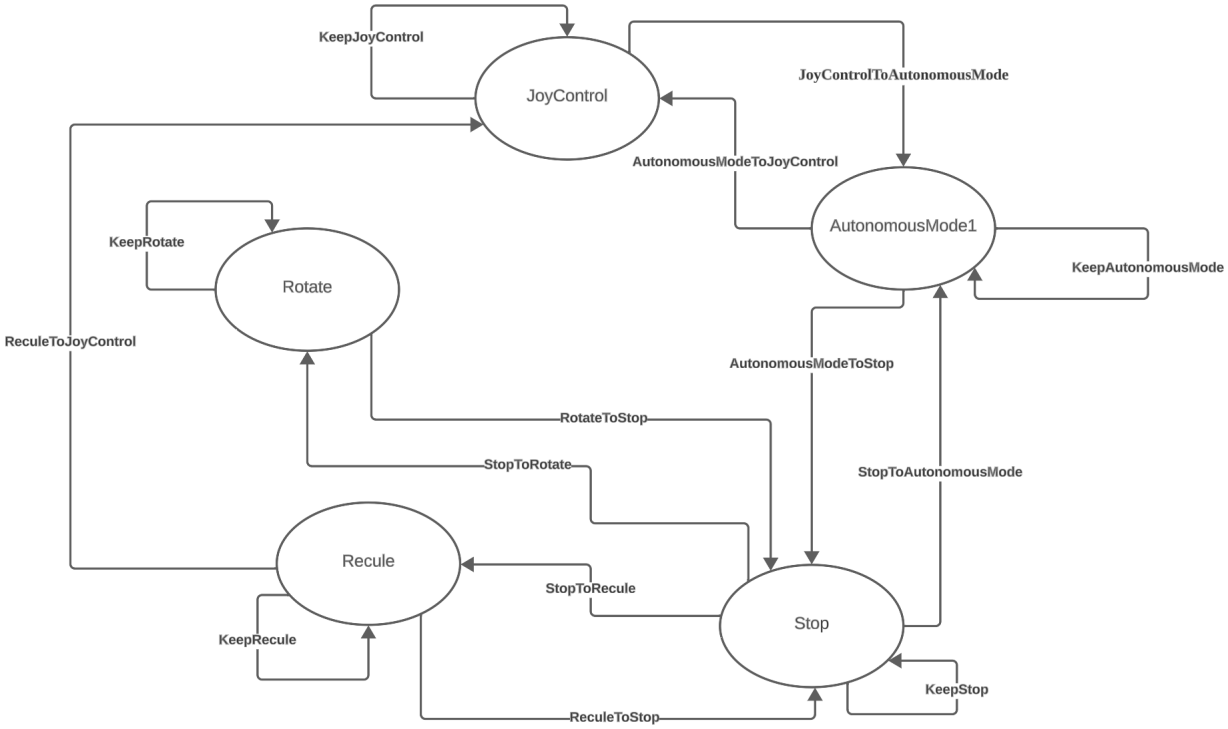

Figure 1 : Schéma machine à états

On retrouve alors ces états :

— JoyControl : État initial après l'allumage. L'utilisateur contrôle le robot avec la manette. En mode manuel, aucune transition ne se fait sans l'intervention de l'utilisateur.

— AutonomousMode : Permet au robot de se déplacer en ligne droite sans intervention de l'utilisateur. Celui-ci doit l'activer. En cas de collision, le robot essayera de se libérer.

— Stop : Arrête le robot lorsqu'il rencontre un obstacle.

— Recule : Le robot recule pour se libérer d'un obstacle. Un compteur limite la durée du recul pour éviter de faire une marche arrière infinie. Le compteur doit prendre en compte qu'un obstacle peut se situer derrière lui.

— Rotate : Permet d'éviter la répétition de la situation. En fonction du bumper activé, le robot dévie sa trajectoire afin de se libérer.

2 Programmation en Orienté Objet

2.1 Le constructeur

La structure de l'orienté objet de Python nous permet de déclarer une classe Robot à l'intérieur de laquelle nous pourrons écrire différentes fonctions qui serviront à contrôler l'automate. En programmation orientée objet, le constructeur d'une classe est une méthode spéciale appelée *init* (en Python). Il est utilisé pour initialiser les attributs d'une instance de la classe lors de sa création. Le constructeur est appelé automatiquement dès qu'une nouvelle instance de la classe est créée. Son rôle principal est de configurer l'état initial de l'objet en initialisant ses attributs. Voici le script du constructeur correspondant à notre programme :

```
class RobotBehavior(object):
########################################################################################
# constructor, called at creation of instance
########################################################################################
def __init__(self, handle_pub, T):
    self.twist = Twist()
    self.twist_real = Twist()
    self.vreal = 0.0 # longitudial velocity
    self.wreal = 0.0 # angular velocity
    self.vmax = 1.5
    self.wmax = 4.0
    self.previous_signal = 0
    self.button_pressed = False
    self.joy_activated = False
    self.pub = handle_pub
    self.T = T
    self.obstacledetecte=False
    self.cpt1=0
    self.cpt2=0
    self.cptr=0
    self.cpt3=0
    self.cpt4=0
    self.cptObs=0
    self.obstacle_middle=False
    self.obstacle_right=False
    self.obstacle_left=False
```

```
# instance of fsm with source state, destination state, condition,callback:
    self.fs = fsm([ ("Start","JoyControl", True ),
    ("JoyControl","AutonomousMode1", self.check_JoyControl_To_AutonomousMode1,
    self.DoAutonomousMode1),
    ("JoyControl","JoyControl", self.KeepJoyControl, self.DoJoyControl),
    ("AutonomousMode1","JoyControl", self.check_AutonomousMode1_To_JoyControl,
    self.DoJoyControl),
    ("AutonomousMode1","AutonomousMode1", self.KeepAutonomousMode1,
    self.DoAutonomousMode1),
    ("AutonomousMode1","Stop1", self.check_AutonomousMode1_To_Stop1,
    self.DoStop1),
    ("Stop1","Stop1", self.KeepStop1, self.DoStop1),
    ("Stop1","Recule", self.check_Stop1_To_Recule, self.DoRecule),
    ("Recule","Recule", self.KeepRecule, self.DoRecule),
    ("Recule","Stop2",self.check_Recule_To_Stop2, self.DoStop2),
    ("Stop2","Stop2", self.KeepStop2, self.DoStop2),
    ("Stop2","Rotate", self.check_Stop2_To_Rotate, self.DoRotate),
    ("Rotate","Rotate", self.KeepRotate, self.DoRotate),
    ("Rotate","Stop3",self.check_Rotate_To_Stop3, self.DoStop3),
    ("Stop3","Stop3", self.KeepStop3, self.DoStop3),
    ("Stop3","AutonomousMode1", self.check_Stop3_To_AutonomousMode1,
    self.DoAutonomousMode1),
    ("AutonomousMode1","AvoidObst",self.check_AutonomousMode1_To_AvoidObst,
    self.DoAvoidObst),
    ("AvoidObst","AvoidObst",self.KeepAvoidObst,self.DoAvoidObst),
    ("AvoidObst","AutonomousMode1",self.check_AvoidObst_To_AutonomousMode1,
    self.DoAutonomousMode1),
    ("AvoidObst","JoyControl", self.check_AvoidObst_To_JoyControl,
    self.DoJoyControl),("AvoidObst","Stop2",self.check_AvoidObst_To_Stop2,
    self.DoStop2)] )
```

Script du Constructeur de la classe Robot

2.2 Méthodes servant au déplacement

En programmation orientée objet, les méthodes jouent un rôle essentiel en encapsulant le comportement spécifique à un objet. Ces fonctions définies à l'intérieur des classes, regroupent les opérations associées à un objet particulier. Cela permet donc de favoriser la modularité et la réutilisation du code. Les méthodes permettent la définition de comportements propres à chaque objet. Dans notre cas, les fonctions de la classe Robot permettent de contrôler sa trajectoire en agissant sur sa position. Voici le script des méthodes utilisées :

```
def DoJoyControl(self, fss, value):
    self.button_pressed = False
    self.cpt_recul = 0
    self.cpt_tourne = 0
```

```
        self.smooth_velocity()
        self.pub.publish(self.twist_real)
        self.collision = False  # Condition pour procedure de desengagement
        print('joy control')
        pass

    def DoAutonomousMode1(self, fss, value):
        self.button_pressed = False
        self.cpt_recul = 0
        self.cpt_tourne = 0
        # Go forward
        go_fwd = Twist()
        go_fwd.linear.x = 0.2
        go_fwd.linear.y = 0
        go_fwd.linear.z = 0
        go_fwd.angular.z = 0
        self.pub.publish(go_fwd)
        print('autonomous')
        pass

    def DoArret(self, fss, value):
        self.collision = False  # Remise a zero de la condition collision
        self.recul = True  # Activation de l'ordre recul
        self.cpt_recul = 0
        self.cpt_tourne = 0
        go_fwd = Twist()
        go_fwd.linear.x = 0
        go_fwd.linear.y = 0
        go_fwd.linear.z = 0
        go_fwd.angular.z = 0
        self.pub.publish(go_fwd)
        print('autonomous stop')
        pass

    def DoRecul(self, fss, value):
        self.recul = False  # Remise a zero de la condition recul
        self.cpt_recul = self.cpt_recul + 1
        # Recul
        go_back = Twist()
        go_back.linear.x = -0.2
        self.pub.publish(go_back)
        print('autonomous back')
        pass

    def DoTourne(self, fss, value):
        self.cpt_tourne = self.cpt_tourne + 1  #
        go_turn = Twist()
        go_turn.linear.x = 0  # Arret de tout mouvement
```

```
        go_turn.linear.y = 0
        go_turn.linear.z = 0
        if self.bumper == 0:
            go_turn.angular.z = -1
        else:
            go_turn.angular.z = 1
        self.pub.publish(go_turn)
        print('autonomous tourne')
        pass

    def DoAvoidObst(self,fss,value):
        if (((self.obstacle_right) and (not self.obstacle_middle) and
        (not self.obstacle_left)) or ((self.obstacle_right) and
        (self.obstacle_middle) and (not self.obstacle_left))):
        goEviteD=Twist()
        R=1
        Vavoid=self.vmax/8.0
        Wavoid=Vavoid/R
        goEviteD.linear.x=Vavoid
        goEviteD.angular.z=Wavoid
        self.pub.publish(goEviteD)
        if (((not self.obstacle_right) and (not self.obstacle_middle)
        and (self.obstacle_left)) or ((not self.obstacle_right) and
        (self.obstacle_middle) and (self.obstacle_left)) or
        ((not self.obstacle_left) and (self.obstacle_right) and
        (self.obstacle_middle))):
        goEviteG=Twist()
        R=1
        Vavoid=self.vmax/8.0
        Wavoid=Vavoid/R
        goEviteG.linear.x=Vavoid
        goEviteG.angular.z=-Wavoid
        self.pub.publish(goEviteG)
```

Script des fonctions de transition de la classe Robot

Les attributs self.twist.linear et self.twist.angular sont issus de la bibliothèque geometrymsgs, et ils sont utilisés pour représenter la vitesse du robot dans l'espace, décomposée en ses composantes linéaires et angulaires. En modifiant ces variables, on peut donc ajuster la trajectoire du robot.

2.3 Méthodes pour les transitions

Afin de pouvoir déterminer à quels moments nous devons utiliser une fonction ou méthode, nous avons donc réalisé l'ensemble des fonctions de transitions qui permettent ou non d'utiliser les méthodes. Pour se simplifier la tâche, ces fonctions ne renvoient que des valeurs booléennes. Cela permet alors une meilleure compréhension du code et une source d'erreur plus faible. Voici les fonctions de transitions :

```
def check_JoyControl_To_AutonomousMode1(self, fss):
    return self.button_pressed
def check_AutonomousMode1_To_Arret(self, fss):
    return False
def check_Arret_To_Recul(self):
    return self.collision
def check_Recul_To_Tourne(self, fss):
    return self.recul
def check_Tourne_To_AutonomousMode1(self, fss):
    return self.cpt_tourne > self.seuiltourne
def check_Tourne_To_JoyControl(self, fss):
    return self.joy_activated
def check_Recul_To_JoyControl(self, fss):
    return self.joy_activated
def check_AutonomousMode1_To_JoyControl(self, fss):
    return self.joy_activated
def KeepJoyControl(self, fss):
    return not self.check_JoyControl_To_AutonomousMode1(fss)
def KeepAutonomousMode1(self, fss):
    return self.check_AutonomousMode1_To_Arret(fss)
def KeepArret(self):
    return not self.check_Arret_To_Recul()
def KeepRecul(self):
    return not self.check_Recul_To_Tourne() and
    not self.check_Recul_To_JoyControl()
def KeepTourne(self, fss):
    return not self.check_Tourne_To_AutonomousMode1(fss) and
    not self.check_Tourne_To_JoyControl(fss)
def check_AutonomousMode1_To_AvoidObst(self,fss):
    return (self.obstacle_middle or self.obstacle_right or self.obstacle_left)
def KeepAvoidObst(self,fss):
    return ((not self.check_AvoidObst_To_AutonomousMode1(fss)) and (not self.ch
    (not self.check_AvoidObst_To_Stop2(fss)))
def check_AvoidObst_To_AutonomousMode1(self,fss):
    self.cpt4=self.cpt4+1
    if (self.cpt4>50):
        return True
    else:
        return False
```

Script des fonctions de transition de la classe Robot

3 Implémentation du capteur

Avec le code précèdent, nous pouvions contrôler le robot soit en mode manuel ou bien en mode automatique. Le contrôle manuel permet de choisir la trajectoire avec précision tandis que le mode automatique présentait quelques défauts. En effet, l'évitement des obstacles n'était pas fait et donc

cela rendait l'utilisation du robot limitée. Nous avons donc décidé d'implémenter un capteur afin de détecter les obstacles et ainsi les éviter. Voici la stratégie utilisée :

```
#############################################################
# Lidar
#############################################################
    def processScan(self,data):

        dist_detect = 1 # 1 m, to be adjusted
        scan_range = data.angle_max - data.angle_min
        nb_values = len(data.ranges) # nombre de valeurs
        for count,value in enumerate(data.ranges):
                obs = ( (not math.isnan(value)) and value < dist_detect)
                current_angle = data.angle_min + count* data.angle_increment
                if (obs and count==nb_values/2):
                        self.obstacle_middle=True
                if (obs and count==nb_values/4):
                        self.obstacle_right=True
                if (obs and count==3*nb_values/4):
                        self.obstacle_left=True
```

Stratégie pour l'évitement

On définit la distance de détection à un 1 mètre. On récupère ensuite les données qui se situent dans la range du capteur et qui sont donc à une distance inférieure à 1 mètre. Ensuite, nous déterminons avec plusieurs conditions ou l'obstacle se trouve. Une fois détecté on change la valeur de la variable booléenne à *True*. Cela va donc permettre d'initier l'utilisation de la méthode servant à l'évitement.

```
def DoAvoidObst(self,fss,value):
    if (((self.obstacle_right) and (not self.obstacle_middle) and
    (not self.obstacle_left)) or ((self.obstacle_right) and
    (self.obstacle_middle) and (not self.obstacle_left))):
        goEviteD=Twist()
        R=1
        Vavoid=self.vmax/8.0
        Wavoid=Vavoid/R
        goEviteD.linear.x=Vavoid
        goEviteD.angular.z=Wavoid
        self.pub.publish(goEviteD)
    if (((not self.obstacle_right) and (not self.obstacle_middle) and
    (self.obstacle_left)) or       ((not self.obstacle_right) and
    (self.obstacle_middle) and (self.obstacle_left)) or
    ((not self.obstacle_left) and (self.obstacle_right) and
    (self.obstacle_middle))):
        goEviteG=Twist()
        R=1
        Vavoid=self.vmax/8.0
        Wavoid=Vavoid/R
```

```
goEviteG.linear.x=Vavoid
goEviteG.angular.z=-Wavoid
self.pub.publish(goEviteG)
```

Stratégie pour l'évitement

Cette stratégie n'est pas la plus optimale car elle peut manquer de précision dans certaines situations et de fluidité. Pour une meilleure précision nous aurions pu utiliser une meilleure stratégie mais nous avons manqué de temps.

Lien vers la vidéo

Conclusion

En conclusion, ces travaux pratiques ont permis une immersion dans le Robot Operating System (ROS), un framework open source facilitant le développement de logiciels pour les robots. Grâce à sa structure favorisant la communication entre différents nœuds logiciels, ROS simplifie la création de systèmes robotiques complexes. Son utilisation étendue en recherche, éducation et industrie témoigne de son impact significatif dans le domaine.

Au cours de ces travaux, la simulation d'un robot TurtleBot à l'aide du simulateur Gazebo a été réalisée, avec un contrôle effectué via un joystick. La communication entre les nœuds dans ROS, basée sur l'utilisation de "topics", a été explorée, offrant une communication asynchrone entre les composants du système robotique.

L'implémentation du contrôle du TurtleBot à l'aide d'une machine à états finis a été détaillée, permettant des transitions fluides entre les états tels que le contrôle manuel, le mode autonome, l'arrêt, le recul, la rotation, et l'évitement d'obstacles. La programmation orientée objet a été exploitée pour structurer le code de manière élégante, avec des méthodes dédiées au déplacement du robot.

L'ajout d'un capteur Lidar a amélioré les capacités du robot en permettant la détection d'obstacles, initiée par une stratégie de détection basée sur la distance. En dépit de certaines limites de la stratégie d'évitement d'obstacles adoptée, cette intégration constitue une première étape vers des fonctionnalités robotiques plus avancées.

En résumé, ces travaux ont fourni une introduction pratique à ROS, mettant en lumière ses fonctionnalités essentielles et son application dans le contrôle et la simulation de robots.