# Compte Rendu Robotique Mobile

MEA3 Robotique Mobile

Juan SANCHEZ

# Sommaire

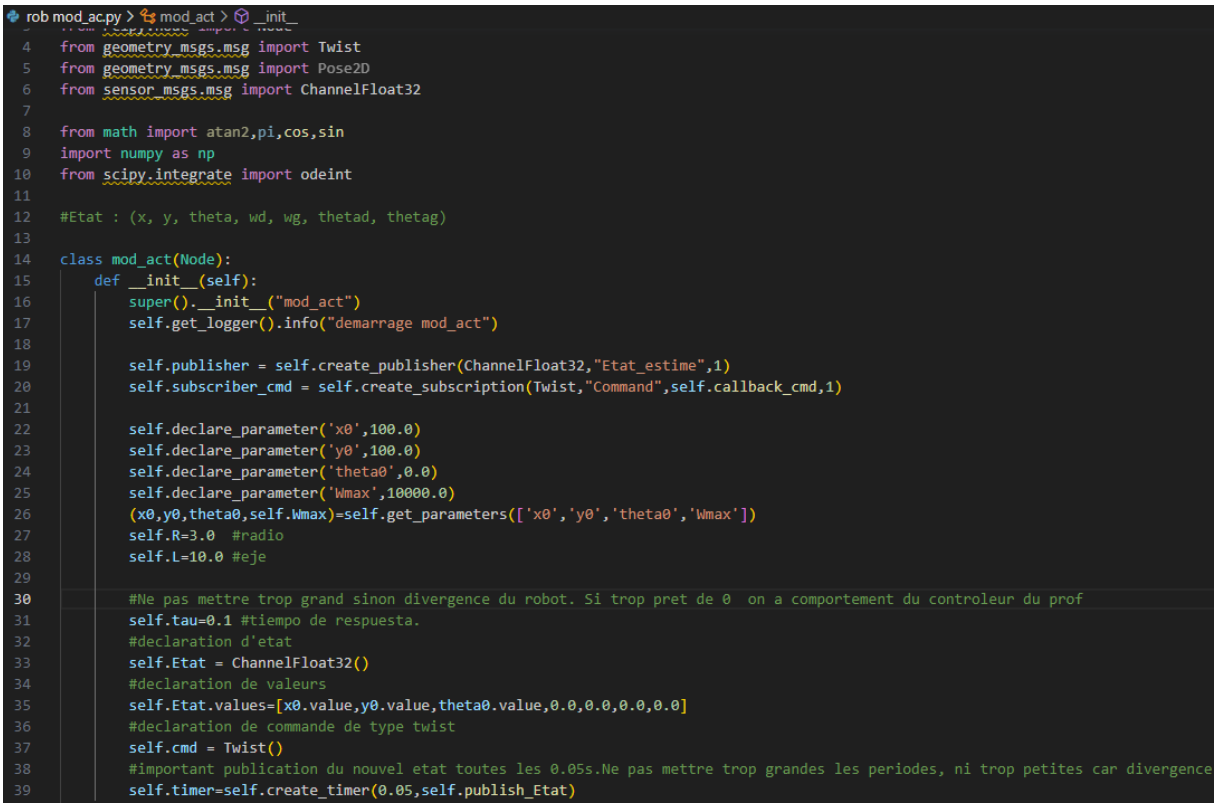
**Modèle d'Actionnement du Robot :**

Le modèle d'actionnement du robot est un nœud ROS qui simule le comportement dynamique du robot en fonction des commandes reçues.

Le code peut être décomposé en 5 grandes parties comme expliqué ici :

1. **Initialisation et Paramètres** :
    - Le nœud est initialisé avec plusieurs paramètres comme la position initiale (x0, y0, theta0) et la vitesse maximale (Wmax).
    - Les constantes physiques du robot comme le rayon des roues (R) et la distance entre les roues (L) sont définies.
2. **Publications et Abonnements** :
    - Publie l'état estimé du robot sur le sujet Etat_estime en utilisant le message ChannelFloat32.
    - Reçoit les commandes de type Twist sur le sujet Command.
3. **Estimation de l'État :**
    - Utilise un intégrateur numérique (odeint de scipy) pour calculer la nouvelle position et orientation du robot en fonction des commandes reçues.
    - Le modèle cinématique prend en compte les vitesses angulaires des roues et l'orientation du robot pour estimer son déplacement.
4. **Calcul des Vecteurs de Commande** :
    - La fonction robot calcule les nouvelles valeurs d'état en fonction des commandes linéaires et angulaires.
    - Intègre les équations de mouvement pour mettre à jour la position (x, y) et l'orientation (theta) du robot.
5. **Publication de l'État :**
    - Publie régulièrement l'état estimé du robot (position et orientation) à un intervalle fixe (toutes les 0,05 secondes).

## Explication Code :



```
rob mod_ac.py > mod_act > __init__
4   from geometry_msgs.msg import Twist
5   from geometry_msgs.msg import Pose2D
6   from sensor_msgs.msg import ChannelFloat32
7
8   from math import atan2,pi,cos,sin
9   import numpy as np
10  from scipy.integrate import odeint
11
12  #Etat : (x, y, theta, wd, wg, thetad, thetag)
13
14  class mod_act(Node):
15      def __init__(self):
16          super().__init__("mod_act")
17          self.get_logger().info("demarrage mod_act")
18
19          self.publisher = self.create_publisher(ChannelFloat32,"Etat_estime",1)
20          self.subscriber_cmd = self.create_subscription(Twist,"Command",self.callback_cmd,1)
21
22          self.declare_parameter('x0',100.0)
23          self.declare_parameter('y0',100.0)
24          self.declare_parameter('theta0',0.0)
25          self.declare_parameter('Wmax',10000.0)
26          (x0,y0,theta0,self.Wmax)=self.get_parameters(['x0','y0','theta0','Wmax'])
27          self.R=3.0  #radio
28          self.L=10.0 #eje
29
30          #Ne pas mettre trop grand sinon divergence du robot. Si trop pret de 0  on a comportement du controleur du prof
31          self.tau=0.1 #tiempo de respuesta.
32          #declaration d'etat
33          self.Etat = ChannelFloat32()
34          #declaration de valeurs
35          self.Etat.values=[x0.value,y0.value,theta0.value,0.0,0.0,0.0,0.0,0.0]
36          #declaration de commande de type twist
37          self.cmd = Twist()
38          #important publication du nouvel etat toutes les 0.05s.Ne pas mettre trop grandes les periodes, ni trop petites car divergence
39          self.timer=self.create_timer(0.05,self.publish_Etat)
```

## Méthode init :

Ce script implémente un modèle de simulation pour un robot mobile utilisant le Framework ROS 2. Il intègre une estimation de l'état du robot à partir de commandes reçues et publie régulièrement cet état.

Tout d'abord, comme pour tout script python qui travaille dans un environnement ROS2, on définit les Publisher et le suscriber afin de recevoir et envoyer les informations. Dans ce cas de figure, nous publions l'état estimé et on prend les nouvelles commandes venant du suscriber de commande.py.

Ensuite nous déclarons et fixons les constantes que nous allons utiliser dans notre programme comme vous pouvez le voir de la ligne 22 a la ligne 28.

Puis de la ligne 31 a la ligne 35 nous déclarons les variables que nous allons utiliser plus tard dans la méthode publish _Etat. Comme c'est le cas de Etat définie comme un ChannelFloat32(un type de l'environnement ROS), à l'intérieur de cet objet 'Etat' nous déclarons et initialisons le membre 'values'. Nous faisons de même pour cmd, mais cette fois-ci elle est de type twist (). Et pour finir, à l'aide d'un timer nous rafraichissons la méthode publish _Etat toutes les 0.05s.

### Publish_Etat :

```python
    #Debut du publisheur
    def publish_Etat(self):
        #declaration des variables
        msg = ChannelFloat32()
        msg.values = [0.0 , 0.0 , 0.0]

        #partie integration numerique, reprise du code python donné en exemple
        dt=0.05
        ts=[0,dt]
        #nouvel etat a integrer(issu du calcul du modele du robot)
        nouvel_etat = odeint(self.robot,self.Etat.values,ts,args=(self.cmd.linear.x,self.cmd.angular.z))
```

Dans publish état la première chose à faire c'est de créer le message que la fonction publish devra retourner. On initialise son membre 'values' a 0 (comme fait préalablement avec la création de l'autre variable de type ChannelFloat32 plus haut).

Ensuite nous intégrons numériquement (comme expliqué dans le cours) le résultat issu du calcul du vecteur d'état calculé plus bas dans le code. Une fois le calcul réalisé a l'aide du odeint (qui appelle la méthode robot (définie plus bas), l'état actuel plus le temps d'intégration), nous venons ranger l'estimation de l'état à un instant donné dans msg.values, et nous publions le nouvel état (x,y,theta).

```python
rob mod_ac.py > ...
        #declaration des variables
        msg = ChannelFloat32()
        msg.values = [0.0 , 0.0 , 0.0]

        #partie integration numerique, reprise du code python donné en exemple
        dt=0.05
        ts=[0,dt]
        #nouvel etat a integrer(issu du calcul du modele du robot)
        nouvel_etat = odeint(self.robot,self.Etat.values,ts,args=(self.cmd.linear.x,self.cmd.angular.z))
        for i in range (7):
            self.Etat.values[i]=nouvel_etat[1,i]
        for i in range (3):
            msg.values[i]=self.Etat.values[i]


        self.publisher.publish(msg)
```

### Méthode Robot :

La méthode robot ne fait que prendre en paramètre un état donné par notre contrôleur, deux vecteurs d'état u et v, et un calcule la vitesse linéaire et de rotation de chaque 'axe du robot, pour pouvoir ainsi avoir une estimation de sa position à un instant donné à l'aide de l'intégration numérique.

D'abord, nous reprenons les formules de Wd et Wg donnés en cours. Elles ne prennent que le rayon de la roue, la distance qui les sépare et les deux vecteurs u et w pour pouvoir effectuer le calcul des vitesses angulaires des roues.

Ensuite nous pouvons calculer les vitesses angulaires de chaque roue, ainsi que la vitesse de déplacement du robot en y, en x et en thêta. Juste avant de retourner un tableau de la bibliothèque

numpy avec le nouveau vecteur d'état, nous reprenons la dérivée de thêta par rapport au temps de chaque roue de l'itération précédente à fin d'effectuer les nouveaux calculs.

```python
    #model du robot pour estuimation realiste de l'etat
    def robot(self,Etat,t,u,w):
        #definitions du cours de omegaconst droite et gauche
        WD = u*(1/self.R) + w*(self.L/self.R)
        WG = u*(1/self.R) - w*(self.L/self.R)
        #equations temporelles des moteurs
        dwd_dt=(1/self.tau)*(WD-self.Etat.values[3])
        dwg_dt=(1/self.tau)*(WG-self.Etat.values[4])

        #estimation de l'etat suivant pour le vecteur de commande, ainsi que le vecteur d'etat
        xpoint=(self.R/2)*(self.Etat.values[3]+self.Etat.values[4])*cos(self.Etat.values[2])
        ypoint=(self.R/2)*(self.Etat.values[3]+self.Etat.values[4])*sin(self.Etat.values[2])
        dtheta_dt=(self.R/(2*self.L))*(self.Etat.values[3]-self.Etat.values[4])

        #avec ces 5 equations on a le nouveau vecteur d'etat pour simuler le systeme

        #on remet le nouveau etat suivant dans etat
        dthetad_dt= self.Etat.values[3]
        dthetag_dt= self.Etat.values[4]

        return np.array([xpoint, ypoint, dtheta_dt, dwd_dt, dwg_dt, dthetad_dt, dthetag_dt])
```

**Méthode callback_cmd :**

Cette méthode met à jour les commandes de mouvement (`linear.x` et `angular.z`) à partir des messages reçus.

```python
    #dans notre cas du robot a deux roues nous actualisons uniquement la vitesse angulaire z et le x
    def callback_cmd(self,msg):
        self.cmd.linear.x = msg.linear.x
        self.cmd.angular.z = msg.angular.z
```

**Méthode fonction main :**

La fonction principale initialise le contexte ROS 2, crée une instance de la classe `mod_act` et lance la boucle d'événements ROS jusqu'à ce que le nœud soit arrêté. Le code vérifie si le script est exécuté directement et appelle la fonction `main` pour démarrer le nœud ROS 2.

**Résumé :**

Le script simule un robot mobile en intégrant ses équations de mouvement et en publiant régulièrement son état estimé. Les commandes de mouvement sont reçues via des messages `Twist`, et l'état est publié en tant que message `ChannelFloat32`. L'intégration numérique est utilisée pour mettre à jour l'état du robot en fonction des commandes reçues et des caractéristiques physiques du robot.

scripts > controleur.py

Welcome

```
from sensor_msgs.msg import ChannelFloat32
#message de l'un des capteurs de type tableau avec name puis tableau. Pour acceder a uune valeur c'est tabl
from math import atan2,pi,cos,sin,acos,sqrt

class Controleur(Node):
    def __init__(self):
        super().__init__("Controleur")
        self.get_logger().info("demarrage Controleur")


        self.subscriber_Etat = self.create_subscription(ChannelFloat32,"Etat_estime",self.callback_Etat,1)
        self.subscriber_Laser= self.create_subscription(ChannelFloat32,"Laser",self.callback_Laser,1)
        self.subscriber_Mission = self.create_subscription(Pose2D,"Cible",self.callback_Mission,1)
        self.etat=Pose2D(x=0.0,y=0.0,theta=0.0)
        self.cible=Pose2D(x=300.0,y=50.0,theta=-1.57)
        self.d1=80.0
        self.d2=80.0
        self.d3=80.0




        self.publisher = self.create_publisher(Twist,"Command",1)
        self.timer=self.create_timer(0.05,self.publish_cmd)

    def publish_cmd(self):

        #commandes
        #theta en radians attention
        #le truc en commantaire me ramene a l'angle final
        pi=3.14159
        #a = self.etat.theta
       # cible=self.cible.theta
        cnt=0

        x=(self.cible.x-self.etat.x)
        y=(self.cible.y-self.etat.y)
        xy=(sqrt((x*x)+(y*y)))


        #cette partie du code permet de directionner la voiture avec le bon angle
        teta=atan2(y,x) #le dphy
        cible=self.cible.theta #angle ciblé
        thetaetat= self.etat.theta #etat actuel de theta
        if(xy>100.0):
                vitesse=130.0
        if(abs(teta-thetaetat)>0.1 and (xy>1.0) and b==0):
            a=(teta-thetaetat)/1.2
        else:
            if(xy>100.0):
                vitesse=200.0
            elif(xy>1.0):
                vitesse=abs(x)+abs(y)

        if(xy<=0.9):
            if(abs(cible-thetaetat)>0.01):
                a=cible-thetaetat
        if(xy>60.0):
            if(self.d2<80.0):
                vitesse=-100.0
                a=0.8
            elif(self.d1<80.0):
                vitesse= 100.0
                a=0.8
            elif(self.d3<80.0):
```

# Explication du Code :

Dans mon code, la première chose à faire c'est c'était la définition des suscribers, pour pouvoir recevoir les informations venant des différents nodes ros2 (ici L'état estimé du robot :Position actuelle et orientation, laser : Les distances mesurées par les capteurs laser., et mission : La position et l'orientation cible), ainsi que la création des variables qu'on utilisera par la suite, comme c'est le cas de d1, d2, d3 pour les lasers, état pour ensuite faire des calculs et la cible pour savoir la ou on doit aller. Une fois les variables déclarées et publiées (ou envoyés sous forme d'un twist) on dit au programme de rafraichir la commande du robot toutes les 0.05s a l'aide d'un timer.

Pour la définition de la commande, dans un premier temps je déclare x et y comme la différence entre la cible et la position actuelle du robot, de façon à savoir plus tard combien nous reste pour arriver à la cible en coordonnées cartésiennes. Puis je déclare une autre variable xy qui représente la diagonale entre la cible et le robot, très utile pour définir des zones de taille différentes en fonction des besoins.

Ensuite, pour les angles je défini d'abord teta comme atan(y/x) (Il s'agit de l'angle theta du robot), puis cible (l'angle de la cible, c'est un très mauvais choix de nom, c'est une chose à améliorer : le choix des noms des variables).

### Commande du robot :

Dans l'idée que j'avais au début, le robot devait d'abord tourner puis aller vers la cible en ligne droite, chose qui faisait que le robot soit très lent et qu'en cas d'obstacle il ait du mal à les contourner. C'est pour cette même raison la que j'ai décidé de m'y prendre d'une autre manière.

Dans un deuxième temps, la deuxième idée, qui est celle que j'ai implémentée pour la compétition, Le robot se déplace en même temps qu'il tourne ce qui le rend très efficace et évite qu'il ne bloque lors qu'il rencontre un obstacle sur son chemin.

Dans le code on a d'abord un if xy>100 nous mettons la vitesse a 130, de façon a ce que si le robot est encore très loin de la cible (en dehors du carré de 100) alors on avance en vitesse « max ».

Le if d'après nous donne la commande de l'angle dans le cas où on est loin et il n'y a pas d'obstacle. L'idée c'est que la voiture doit tourner jusqu'à ce qu'elle soit dans un carré de 2*2 et la valeur absolue antre son angle et celle de la cible soit inferieur a 0.1. L'angle de la commande qui est donc la différence entre la cible et l'angle de la voiture diminue lors qu'on arrive à l'angle cible jusqu'à l'atteindre (et donc la différence sera nulle). (Le b==0 ne fait rien dans le if, j'ai oublié de l'enlever lors d'un test)

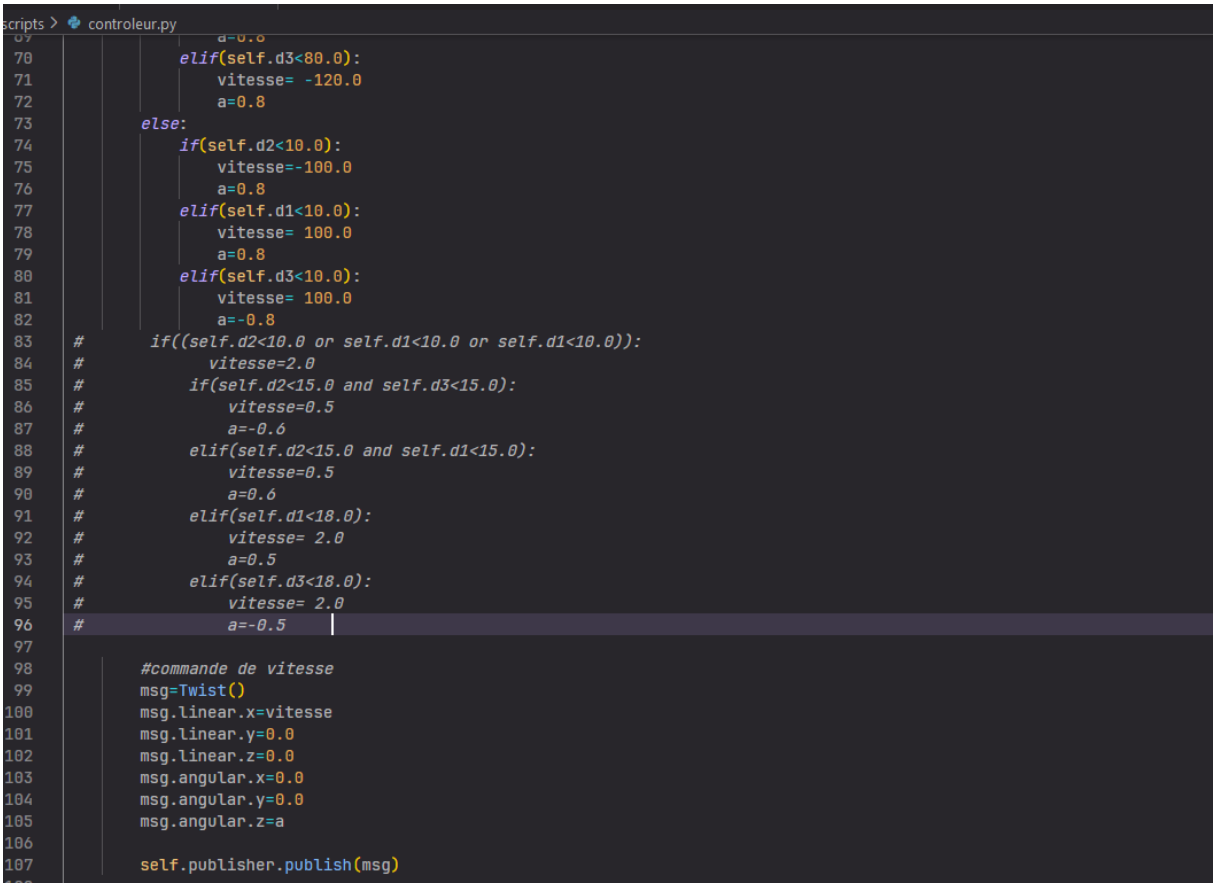
Le else est exécuté lorsque l'on est relativement proche de la cible (entre 100.0>xy>1.0) la vitesse devient proportionnelle elle aussi a la distance qui reste jusqu'à la cible (en valeur absolue bien sûr).

Un dernier if nous permet d'arrêter la voiture lorsqu'elle est dans le carré 2*2 ciblé, et faire tourner la voiture à l'angle souhaité (on reprend la même différence utilisé préalablement), et on met la vitesse à 0 (manque dans le code

**Intégration des lasers :**

Ensuite, vient la partie qui nous permet d'intégrer les lasers à la commande pour que la voiture puisse détecter les obstacles sur la route et puisse donc les éviter. Pour cette partie, l'idée est de faire tourner toujours à droite, et si on est trop prêt de l'objet on recule suffisamment pour permettre à la voiture de tourner à droite. La grande limitation de cette approche est si la cible est derrière un obstacle, car

on va reculer mais la voiture ne va pas contourner l'obstacle, elle va à nouveau avancer et ainsi de suite. Pour corriger ce comportement erroné on peut privilégier le contournement d'obstacle avant de vouloir aller à la cible, chose que malheureusement je n'ai pas pu traiter dans ce programme et que j'ai laissé en commentaires). Une autre limitation est qu'on ne peut pas savoir si en reculant nous allons percuter un obstacle, car on ne possède pas de capteur laser à l'arrière de la voiture.




Pour la partie code, il y a trois cas à traiter, si la voiture, est en dehors d'un carré de 60*60. Et trois autres si la voiture est dans ce carré. D'abord, si un objet est détecté au loin par l'un des lasers (en dehors du carré), alors on fait quelque chose pour chacun des cas. Le premier, c'est un objet et pile en face de nous, dans ce cas on fait reculer la voiture et on lui dit de tourner à droite, pour que dans le tour d'après le laser d1 détecte l'obstacle. Ensuite d1, s'il détecte un objet alors il doit juste tourner à droite et continuer à avancer en limitant la vitesse. Et le troisième cas, si c'est d3 alors on tourne, mais cette fois-ci a gauche et en limitant la vitesse.

Si la voiture est dans le carré proche de la cible, on fait exactement la même chose mais avec beaucoup plus de douceur en termes de vitesse. Il est important aussi de remarquer que le contournement des objets se fait uniquement s'ils sont vraiment très prêts de la voiture (d'où le réglage des lasers a 10). Ce l'évite que la voiture diverge si la cible est devant la voiture, mais avec un mur derrière cette cible.

Pour finir avec les lasers, c'est dans cette partie-là où je pourrais apporter bien plus d'améliorations. Une chose qui m'a aidé à mieux comprendre et à mieux programmer le robot fut la modélisation de machines à état en SED. En effet, elles nous permettent de voir plus claire ce que nous voulons réaliser et pourquoi dans mon programme il y a des cas bloquants

**Publication de la commande :**

Puis la commande de la vitesse angulaire en z et la vitesse linéaire en x est publié sous forme de Twist

**Déclaration des méthodes :**

Ensuite, pour finir il ne reste plus que la déclaration des méthodes ainsi que le main du ros : `callback_Etat`, `callback_Mission`, et `callback_Laser`

Comme expliqué plus haut, ces méthodes mettent à jour respectivement :

- L'état estimé du robot (position actuelle et orientation).

- La position et l'orientation cible.

- Les distances mesurées par les capteurs laser.

**Déclaration du main :**

Puis comme dans tout code contenant du ros2 une fonction main doit être déclarée.

D'après ce que j'ai compris cette fonction initialise le contexte ROS 2, crée une instance de la classe `Contrôleur` et lance la boucle d'événements ROS jusqu'à ce que le nœud soit arrêté. Puis le code vérifie si le script est exécuté directement et appelle la fonction `main` pour démarrer le nœud ROS 2.

**Résumé :**

Donc en résumé, controleur.py ajuste en temps réel les commandes de mouvement calculées dans publish_cmd basés sur la position actuelle, la cible, et les lectures des capteurs laser. Les commandes sont publiées en tant que messages `Twist`, et l'état du robot est mis à jour régulièrement à partir des données reçues des abonnements ROS 2.

```
109
110     #definition de l'état estimé envoyé par le mod d'actionnement, position de la cible, ainsi que les capteurs laser
111
112     def callback_Etat(self,msg):
113         self.etat.x=msg.values[0]
114         self.etat.y=msg.values[1]
115         self.etat.theta=msg.values[2]
116
117     def callback_Mission(self,msg):
118         self.cible.x=msg.x
119         self.cible.y=msg.y
120         self.cible.theta=msg.theta
121
122     def callback_Laser(self,msg):
123         self.d1=msg.values[0]
124         self.d2=msg.values[1]
125         self.d3=msg.values[2]
126
127
128
129 def main(args=None):
130     rclpy.init(args=args)
131     node=Controleur()
132     rclpy.spin(node)
133     rclpy.shutdown()
```