MODAL INF472R - ROBOTS ET DRONES

Rapport de Modal

9 février 2022

Joe KHAWAND

ÉCOLE POLYTECHNIQUE

INSTITUT POLYTECHNIQUE DE PARIS

F1TENTH

f1tenth.org

Sommaire

1 Introdution

Les premières semaines furent consacrées à l'installation de Linux et de ROS-noetic, ainsi qu'à la prise en main de ROS à travers les différents tutoriels présents sur le site `http://wiki.ros.org/ROS/Tutorials`.

2 Navigation avec TurtleBot

Nos premiers pas sur ROS furent avec les TurtleBot. Ces robots de prototypage viennent avec de grandes librairies de packages pré-codés, comme le SLAM la navigation et un simulateur de TurtleBot.

Après avoir testé les différentes fonctionnalités de base comme le teleop et l'affichage des points vus par le lidar sur rviz, nous avons codé un arrêt d'urgence basique pour le TurtleBot.

home > joe > catkin_ws > src > avoid > src > ttbot.py > ...

```python
#!/usr/bin/env python3
import rospy
from sensor_msgs import msg
from sensor_msgs.msg import LaserScan
from geometry_msgs.msg import Twist

rospy.init_node('ligne', anonymous=True)
pub1=rospy.Publisher('cmd_vel', Twist,queue_size=10 )

vel= Twist()
vel.linear.x=0.22

def dist(msg):
    if msg.ranges[0]> 0.5 :
        pub1.publish(vel)
    else:
        vel.linear.x=0
        pub1.publish(vel)

subs = rospy.Subscriber('/scan', LaserScan, dist)

rospy.spin()
```

FIGURE 2.1: Arrêt d'urgence simple

Celui-ci voit si l'obstacle est à moins de 0.5m directement devant lui et s'arrête.

Dans un second temps, nous avons testé SLAM en simulation et sur le TurtleBot.

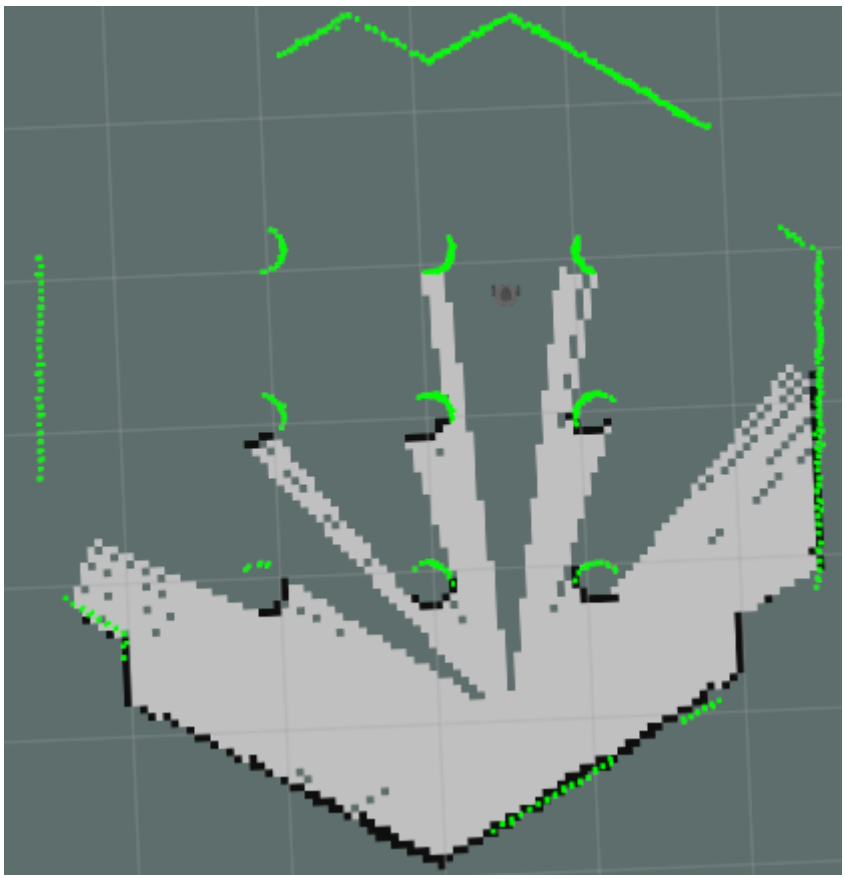

Figure 2.2: SLAM en simulation

3 Vision

3.1 Prise en main de OpenCV

Le TD présent sur moodle m'a permis de me familiariser avec l'implémentation d'OpenCv dans un contexte ROS. La subtilité dans cette implémentation est le passage des messages de ROS à des données OpenCv. Ceci s'effectue grâce à un convertisseur CvBridge.

vis.py > ...

```python
#!/usr/bin/env python3
from __future__ import print_function
from email.mime import image
import sys
import rospy
import cv2
from std_msgs.msg import String
from sensor_msgs.msg import Image
from cv_bridge import CvBridge, CvBridgeError

class image_converter:

    def __init__(self):
        print("init")
        self.bridge = CvBridge()
        self.image_sub = rospy.Subscriber("/cv_camera/image_raw",Image,self.callback)

    def callback(self,data):
        try:
            cv_image = self.bridge.imgmsg_to_cv2(data, "bgr8")
        except CvBridgeError as e:
            print(e)

        # Image processing below
        (rows,cols,channels) = cv_image.shape
        if cols > 60 and rows > 60 :
            cv2.circle(cv_image, (50,50), 10, 255)
        cv2.imshow("Image window", cv_image)
        cv2.waitKey(3)

def main(args):
    ic = image_converter()
    rospy.init_node('image_converter', anonymous=True)
    try:
        rospy.spin()
    except KeyboardInterrupt:
        print("Shutting down")
    cv2.destroyAllWindows()

if __name__ == '__main__':
    main(sys.argv)
```

FIGURE 3.1: Code pour ouvrir la webcam avec OpenCV.

3.2 Implémentation de Camshift

CamShift est un algorithme de détection d'objet avec zone de tracking adaptative. Comme celui-ci est déjà présent dans les librairies d'OpenCV, la difficulté ici est l'implémentation de l'algorithme avec ROS.

Le fonctionnement global de la boucle peut être résumé en les étapes suivantes :

1. On récupère dans un premier temps les données de la camera (de type *Image*) et on les transforme grâce CvBridge en données utilisables par OpenCv.
2. On règle après le rectangle en utilisant la première image qui nous parvient de la camera.
3. On applique ensuite l'algorithme de Camshift disponible dans les librairies d'OpenCV.
4. Finalement on dessine le rectangle sur l'image et on l'affiche.

camshift.py > cam_shift > callback

```
14  setup=True
15  class cam_shift:
16
17      def __init__(self):
18          print("init")
19          self.bridge = CvBridge()
20          self.image_sub = rospy.Subscriber("/cv_camera/image_raw",Image,self.callback)
21
22      def callback(self,data):
23          global setup
24          global roi
25          global hsv_roi
26          global mask
27          global roi_hist
28          global term_crit
29          global track_window
30          # Recuperation du message image et transformation en image opencv en utilisant "Bridge"
31          try:
32              cap = self.bridge.imgmsg_to_cv2(data, "bgr8")
33          except CvBridgeError as e:
34              print(e)
35
36          if (setup): #Setup a faire une fois au debut
37
38              # Position initiale du rectangle
39              r,h,c,w = 250,90,400,125
40              track_window = (c,r,w,h)
41
42              # set up du rectangle pour le tracking
43              roi = cap[r:r+h, c:c+w]
44              hsv_roi =  cv2.cvtColor(roi, cv2.COLOR_BGR2HSV)
45              mask = cv2.inRange(hsv_roi, np.array((0., 60.,32.)), np.array((180.,255.,255.)))
46              roi_hist = cv2.calcHist([hsv_roi],[0],mask,[180],[0,180])
47              cv2.normalize(roi_hist,roi_hist,0,255,cv2.NORM_MINMAX)
48
49              # SCritere d'arret, 10 iterations ou bouger de 1 pt
50              term_crit = ( cv2.TERM_CRITERIA_EPS | cv2.TERM_CRITERIA_COUNT, 10, 1 )
51
52              setup=False
53          # Image processing
54          ret = True
55          hsv = cv2.cvtColor(cap, cv2.COLOR_BGR2HSV)
56          dst = cv2.calcBackProject([hsv],[0],roi_hist,[0,180],1)
57
58          # Appliquer l'algo de Camshift
59          ret,track_window = cv2.CamShift(dst, track_window, term_crit)
60
61          # Dessiner le rectangle
62          pts = cv2.boxPoints(ret)
63          pts = np.int0(pts)
64          img2 = cv2.polylines(cap,[pts],True, 255,2)
65          cv2.imshow('img2',img2)
66          k = cv2.waitKey(60) & 0xff #affichage de l'image en 60 fps
67          if k == 27:
68              return
69          else:
70              cv2.imwrite(chr(k)+".jpg",img2)
```

Figure 3.2: Code de Camshift

4 F1-tenth Project

Le projet "F1-tenth" fut la culmination de notre travail avec ROS. Le but de ce projet était de coder différents scripts permettant à une voiture de type "F1-tenth" de faire une course en autonomie totale.

4.1 Emergency Breaking

Le premier script que nous avons codé est un script d'arrêt d'urgence basique. Ce script récupère les données du lidar dans un cône prédéfini et calcule le temps de collision ; si ce temps est inférieur à un seuil donné, la voiture s'arrête.

Après plusieurs tests avec l'algorithme de wallfollowing, j'ai remarqué que la voiture s'arrêtait dans les virages car elle voyait le mur. J'ai donc décidé de ne pas récupérer les données en cône et de juste prendre la valeur devant la voiture.

En simulation, cela marche très bien, mais j'ai remis le cône sur la voiture pour plus de sécurité. (code en commentaire)

```python
class Safety(object):
    """
    La classe qui s'occupe du freinage d'urgence
    """

    def __init__(self):
        """
        /brake topic: Topic pour envoyer les instructions au robot.(message de type AckermannDriveStamped)
        /brake_bool topic: Topic pour activer le freinage(message de type Bool).
        /scan topic: donnees du lidar
        /odom topic: permet de recuperer la vitesse a partir des donnees odometriques

        """
        #brake= "/vesc/low_level/ackermann_cmd_mux/input/safety"
        brake2='/brake'
        #odom="/vesc/odom"
        odom2="/odom"

        self.speed = 0
        self.break_pub= rospy.Publisher(brake2,AckermannDriveStamped, queue_size=50)
        self.breakbool_pub = rospy.Publisher("/brake_bool",Bool, queue_size=50)
        odom_sub = rospy.Subscriber(odom2, Odometry, self.odom_callback)
        scan_sub = rospy.Subscriber("/scan", LaserScan, self.scan_callback)
        # Creation des ROS subscribers et publishers.

    def odom_callback(self, odom_msg):
        #  Mise a jour de la vitesse
        self.speed = odom_msg.twist.twist.linear.x

    def scan_callback(self, scan_msg):
        # Calcul du Temps de collision TTC
        # comme le robot se colle au mur j'ai decide de voir juste l'angle devant pour eviter des arret soudains
        #for i in range(0, len(scan_msg.ranges)):
            i=-abs(scan_msg.angle_min)
            k=0
            while i<0:
                i=i+scan_msg.angle_increment
                k+=1
            r = max(0, self.speed)
            #*cos(scan_msg.angle_min+(scan_msg.angle_increment*-i))

            if r !=0:
                if(scan_msg.ranges[k]/r)< 0.4:
                    ack_msg= AckermannDriveStamped()
                    ack_msg.header.stamp= rospy.Time.now()
                    ack_msg.drive.steering_angle = 0
                    ack_msg.drive.speed=0

                    self.breakbool_pub.publish(True)
                    self.break_pub.publish(ack_msg)
```

FIGURE 4.1: Code de l'arrêt d'urgence.

4.2 Wall following

L'algorithme de Wall following permet à la voiture de suivre un mur en compensant l'erreur de mesure grâce à un contrôle PID. La boucle principale de cet algorithme peut être résumé en les étapes suivantes :

1. Dans un premier temps on récupère les données du lidar du topic */scan* qu'on envoie dans la fonction *lidar_callback*.
2. Dans cette fonction on récupère les distances à droite et à un angle de 70 degrés de la voiture et on les utilise pour calculer l'erreur présentée dans le cours.
3. Cette erreur est ensuite envoyée dans la fonction *pid_control* où elle sera utilisée pour calculer l'angle de virage de la voiture avec formule présentée dans le cours.
4. Cet angle est finalement publié dans le topic */nav* avec un message de type *AckermannDriveStamped*.

```python
#PID CONTROL PARAMS
kp = 3.2#TODO
kd = 0#TODO
ki = 0#TODO
servo_offset = 0.0
prev_error = 0.0
error = 0.0
integral = 0.0

#WALL FOLLOW PARAMS
ANGLE_RANGE = 270 # Hokuyo 10LX has 270 degrees scan
DESIRED_DISTANCE_RIGHT = 0.6 # meters
DESIRED_DISTANCE_LEFT = 0.55
VELOCITY = 3 # meters per second
CAR_LENGTH = 0.50 # Traxxas Rally is 20 inches or 0.5 meters

class WallFollow:
    """ Implement Wall Following on the car
    """
    def __init__(self):
        #Topics & Subs, Pubs
        lidarscan_topic = '/scan'
        #drive_topic = '/vesc/low_level/ackermann_cmd_mux/input/navigation'
        drive_topic2= '/nav'

        self.lidar_sub = rospy.Subscriber(lidarscan_topic,LaserScan,self.lidar_callback) #TODO: Subscribe to LIDAR
        self.drive_pub = rospy.Publisher(drive_topic2,AckermannDriveStamped,queue_size=50) #TODO: Publish to drive

    def getRange(self, data, angle):
        # data: single message from topic /scan
        # angle: between -45 to 225 degrees, where 0 degrees is directly to the right
        # Outputs length in meters to object with angle in lidar scan field of view
        #make sure to take care of nans etc.
        #TODO: implement
        a= data.angle_min
        inc=data.angle_increment
        i= int((angle-np.pi/2-a)/inc)
        if( not math.isnan(data.ranges[i])) and data.ranges[i] <= data.range_max:
            return data.ranges[i]
        else:
            return data.range_max
```

```python
    def pid_control(self, error, velocity):
        global integral
        global prev_error
        global kp
        global ki
        global kd

        integral += error
        angle = kp*error+ki*(integral)+kd*(error-prev_error)

        #TODO: Use kp, ki & kd to implement a PID controller for
        if abs(angle*180/np.pi)<10 : velocity=3
        elif abs(angle*180/np.pi)< 20 : velocity=2
        else : velocity=1
        drive_msg = AckermannDriveStamped()
        drive_msg.header.stamp = rospy.Time.now()
        drive_msg.header.frame_id = "laser"
        drive_msg.drive.steering_angle = angle
        drive_msg.drive.speed = velocity
        self.drive_pub.publish(drive_msg)
        prev_error=error

    def followLeft(self, data, leftDist):
        #Follow left wall as per the algorithm
        #TODO:implement
        return 0.0

    def lidar_callback(self, data):
        a=self.getRange(data,(np.pi)/3)
        b=self.getRange(data,0)

        ang=np.pi/3
        alpha=np.arctan2((a*np.cos(ang)-b),(a*np.sin(ang)))

        error = DESIRED_DISTANCE_RIGHT - (b*np.cos(alpha)+CAR_LENGTH*np.sin(alpha))
        #send error to pid_control
        self.pid_control(error, VELOCITY)

def main(args):
    rospy.init_node("WallFollow_node", anonymous=True)
    wf = WallFollow()
    rospy.sleep(0.1)
    rospy.spin()

if __name__=='__main__':
    main(sys.argv)
```

Figure 4.2: Code de Wallfollowing.

Dans cet algorithme le choix des constantes *kp*, *ki*, et *kd* est primordial, car celles-ci permettent de corriger les différentes déviations de la voiture. Après plusieurs tests, j'ai remarqué que pour le cas de la simulation, une constante $kp = 3.2$ était suffisante pour une bonne navigation. En effet, comme la simulation est un milieu parfait la voiture dévie peu et donc n'a pas besoin de compensations de type *ki* et *kd*.

4.3 Implémentation sur la voiture

L'implémentation de ces 2 scripts sur la voiture fut rude. Dans un premier temps, j'ai essayé de contrôler la voiture en ssh via l'ordinateur, mais cela n'a pas très bien fonctionné à cause de la connection internet. J'ai ainsi dans un second temps, copié les algorithmes sur la voiture en veillant

à bien changer la version de python utilisée et j'ai réussi à faire fonctionner les scripts sur la voiture. Cepedant, j'ai fait face à un problème majeur. Les moteurs de la *voiture*2 grésillaient à vitesse faible et patinaient, ce qui introduisait une erreur imprévue dans le mouvement de la voiture. Ayant prouvé que les scripts fonctionnaient un minimum sur la voiture (arrêt d'urgence parfaitement fonctionnel et wallfollow fonctionnel avec grésillement sur les virages) j'ai décidé de me focaliser par la suite sur le travail en simulation.

4.4 Pure Pursuit

Le dernier projet du modal était PurePursuit où le but était de faire suivre une succession de points à la voiture. Celui-ci fonctionne de la sorte :

1. On enregistre les points du trajet avec waypointlogger.py
2. Ces points sont ensuite disposés dans une array de la forme suivante : $[x_1, y_1, x_2, y_2, ...]$
3. On récupère la position du robot grâce au topic */gt_pose* et on envoie cette position dans la fonction *pose_callback*.
4. Dans *pose_callback* on trouve l'angle du robot en transformant sa position en quaternions et on calcule ensuite y
5. Finalement, on utilise y pour calculer l'angle de virage qu'on publie sous forme d'*AckermannDriveStamped*.

```python
home = expanduser('~')
file = open(home+'/waypoint.csv')
csvreader = csv.reader(file)
points = []
inter=0
kp=1

#creation d'un tableau [x,y,x,y,x,y .....]
for row in csvreader:
        for i in row:
            points.append(float(i))

file.close()
print(len(points))
velocity=1
p=1000
L=1
xp= float(points[0])
yp= float(points[1])

def distance(a,b,c,d):

    return m.sqrt((c-a)**2+(d-b)**2)

class PurePursuit(object):

    #The class that handles pure pursuit.

    def __init__(self):
        # TODO: create ROS subscribers and publishers.
        pos_sub = rospy.Subscriber('/gt_pose', PoseStamped, self.pose_callback)
        self.drive_pub = rospy.Publisher('/nav',AckermannDriveStamped,queue_size=50)



    def pose_callback(self, pose_msg):
        global p
        global points
        global L
        global xp
        global yp
        global kp

        xr= float(pose_msg.pose.position.x)
        yr= float(pose_msg.pose.position.y)

        # TODO: find the current waypoint to track using methods mentioned in lecture
        l= distance(xr,yr,points[p+2],points[p+3])

        if l<= L: #nouveau point dans le cercle
            p=p+100
            xp=points[p]
            yp=points[p+1]
        else: #2 options interpoler ou continuer a aller vers le point(rien faire)
            if distance(xr,yr,xp,yp)<L/4:#interpole
                xp=xr+(points[p+200]-xr)*(L/1)
                yp=yp+(points[p+201]-yr)*(L/1)

        # TODO: calculate curvature/steering angle

        quat = pose_msg.pose.orientation
        siny_cosp = 2 * (quat.w * quat.z + quat.x * quat.y)
        cosy_cosp = 1 - 2 * (quat.y * quat.y + quat.z * quat.z)
        yaw = m.atan2(siny_cosp, cosy_cosp)
        tetapoint = m.atan2(yp - yr, xp - xr)

        alpha = tetapoint - yaw

        while alpha > np.pi:
            alpha -= 2 * np.pi
        while alpha <= - np.pi:
            alpha += 2 * np.pi

        if alpha > np.pi / 2:
            curv = 0.4188
        elif alpha < - np.pi / 2:
            curv = - 0.4188
        else:
            y = 1 * m.sin(alpha)
            curv = 2 * y / (1*1)
            while curv > np.pi:
                curv -= 2 * np.pi

            while curv <= - np.pi:
                curv += 2 * np.pi

            curv = min(curv, 0.4188)
            curv = max(curv, -0.4188)

        # Publish drive message
        drive_msg=AckermannDriveStamped()
        drive_msg.header.stamp = rospy.Time.now()
        drive_msg.header.frame_id = "pursuit"
        drive_msg.drive.steering_angle = curv
        drive_msg.drive.speed = velocity
        self.drive_pub.publish(drive_msg)

def main():
    rospy.init_node('pure_pursuit_node')
    pp = PurePursuit()
    rospy.spin()

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

Figure 4.3: Code de Pure Pursuit

La difficulté majeure de ce problème était la compréhension des quaternions et leur utilisation dans *rospy*, mais une fois cela assimilé le reste fut implémenté avec facilité.

5 Lien Github

Voici un lien vers tous les codes décris dans ce rapport : `https://github.com/Khokho199/ModalRobotique`