**Proiect Sincretic-Autobot**

**Niculescu Darius, Mladin Alexandru, Plesa Valentin-Razvan**

## 1. Introducere

În domeniul roboticii mobile, una dintre cele mai frecvente aplicații este dezvoltarea de algoritmi pentru urmărirea unui traseu (bandă de mers) și evitarea obstacolelor. Platformele robotice precum TurtleBot3 oferă un mediu excelent de testare și prototipare, fiind o soluție flexibilă și accesibilă pentru începători și profesioniști deopotrivă.

**Conducerea la distanță (remote) și autonomia** sunt două aspecte importante în zilele noastre, întrucât se dorește reducerea erorilor umane și creșterea siguranței. Astfel de concepte pot fi extinse la vehicule autonome în trafic, roboți de livrare, roboți industriali etc.

Proiectul de față îmbină noțiunile fundamentale de **prelucrare a imaginilor** (detectarea unei benzi galbene), **utilizarea senzorilor de tip LiDAR** (detecția și ocolirea obstacolelor) și elemente de **control** (menținerea robotului pe traiectorie). În plus, se prezintă și o încercare nefinalizată de recunoaștere a semnelor de circulație (în special semn de STOP), pentru a arăta potențialul de extindere a proiectului.

## 2. Prezentarea temei

**Tema proiectului** constă în realizarea unei aplicații pentru robotul mobil TurtleBot3 (simulat în Gazebo, programat în Python cu ROS2 Humble) care să:

1. Urmărească **banda de mers** (o linie galbenă) bazându-se pe imagini capturate cu camera robotului.
2. Detecteze și **evite obstacolele** detectate cu ajutorul senzorului LiDAR.
3. Să includă o **încercare de detectare a semnelor de circulație** (un semn de STOP), care momentan este comentată în cod și nu funcționează complet (încercare eșuată).

Robotul se deplasează autonom, ținând centrul benzii, iar când detectează obstacole, aplică un comportament de ocolire (trecând pe partea opusă a benzii). După ce obstacolul este lăsat în urmă, robotul revine pe banda inițială și își continuă traseul.

## 3. Tehnologii utilizate

Pentru dezvoltarea proiectului s-au folosit următoarele tehnologii:

1. **Robot hardware & simulare**:
   1. *TurtleBot3* (în simulare Gazebo).
2. **Sistem de operare și framework robotic**:
   1. *ROS2 Humble Hawksbill* – versiunea de ROS2 utilizată pentru noduri, mesaje, topici etc.
3. **Limbaj de programare**:
   1. *Python 3* (bibliotecile de bază: rclpy, numpy, opencv-python, cv\_bridge).
4. **Simulare & mediu de vizualizare**:
   1. *Gazebo Simulator* – pentru rularea mediului virtual cu un model 3D de TurtleBot3 și obstacole plasate în scenă.
   2. *RViz2* (opțional) – pentru vizualizarea datelor LiDAR, a traiectoriilor, etc.
5. **Biblioteci de procesare imagine**:
   1. *OpenCV* – folosită pentru detecția liniei galbene și (opțional) a semnului STOP.
6. **Altele**:
   1. *numpy* – operații pe matrice și array-uri, prelucrări de date de la senzor.
   2. *time* – gestiunea temporizărilor și tranzițiilor în stările FSM (finite state machine).
   3. *cv\_bridge* – pentru conversia mesajelor ROS de tip sensor\_msgs/Image către format OpenCV.

## 4. Ghidul programatorului

#!/usr/bin/env python3

import rclpy

from rclpy.node import Node

from sensor\_msgs.msg import Image, LaserScan

from geometry\_msgs.msg import Twist

from cv\_bridge import CvBridge

import cv2

import numpy as np

import time

class LaneFollower(Node):

    def \_init\_(self):

        super().\_init\_('lane\_follower\_node')

        # Subscriber camera (pentru stop sign + linie galbenă)

        self.image\_sub = self.create\_subscription(

            Image,

            '/camera/image\_raw',

            self.image\_callback,

            10

        )

        # Subscriber LiDAR (pentru obstacol)

        self.scan\_sub = self.create\_subscription(

            LaserScan,

            '/scan',

            self.lidar\_callback,

            10

        )

        self.cmd\_vel\_pub = self.create\_publisher(Twist, '/cmd\_vel', 10)

        self.bridge = CvBridge()

        # ---------------------------

        # Parametri lane-follow

        # ---------------------------

        self.linear\_speed = 0.4

        self.angular\_gain = 0.015

        self.filtered\_error = 0.0

        self.alpha = 0.7

        # LiDAR

        self.latest\_scan\_ranges = None

        self.num\_lidar\_rays = 0

        # Prag pentru considerare obstacol în față

        self.lidar\_threshold = 1.6

        # FSM stări

        self.LANE\_FOLLOW = 0

        self.CROSS\_MIDDLE\_LINE = 1

        self.PASS\_OBSTACLE = 2

        self.CROSS\_BACK = 3

        self.fsm\_state = self.LANE\_FOLLOW

        self.state\_start\_time = time.time()

        # Timp viraj stânga (depășire amplă)

        self.turn\_left\_time = 1.6

        # DETECTAREA OBSTACOLULUI “CONSISTENT” ÎN LANE\_FOLLOW

        # ex. 5 cadre consecutive

        self.obstacle\_in\_front\_count = 0

        self.obstacle\_in\_front\_count\_needed = 10

        # În PASS\_OBSTACLE: confirmare obstacol în spate

        self.obstacle\_behind\_count\_needed = 10

        self.obstacle\_behind\_count = 0

        self.get\_logger().info("Lane Follower node started (LiDAR obstacole, + consecutive frames).")

    # ---------------------------------------------------------------------

    # LiDAR

    # ---------------------------------------------------------------------

    def lidar\_callback(self, scan\_msg):

        self.latest\_scan\_ranges = scan\_msg.ranges

        self.num\_lidar\_rays = len(scan\_msg.ranges)

    # ---------------------------------------------------------------------

    # image\_callback

    # ---------------------------------------------------------------------

    def image\_callback(self, msg):

        cv\_image = self.bridge.imgmsg\_to\_cv2(msg, desired\_encoding='bgr8')

        # 1) STOP SIGN (opțional)

        '''

        if self.detect\_stop\_sign(cv\_image):

            self.get\_logger().info("STOP sign detected! Robot stops.")

            self.stop\_robot()

            self.show\_debug(cv\_image, None)

            return

          '''

        # 2) FSM

        if self.fsm\_state == self.LANE\_FOLLOW:

            # Verificare obstacol cu “buffer” de cadre

            if self.lidar\_obstacle\_in\_front():

                self.obstacle\_in\_front\_count += 1

            else:

                self.obstacle\_in\_front\_count = 0

            # Dacă am atins numărul minim de cadre consecutive

            if self.obstacle\_in\_front\_count >= self.obstacle\_in\_front\_count\_needed:

                self.get\_logger().info("Obstacle in front (x frames) => CROSS\_MIDDLE\_LINE.")

                self.fsm\_state = self.CROSS\_MIDDLE\_LINE

                self.state\_start\_time = time.time()

            else:

                # Lane-follow normal

                self.lane\_follow(cv\_image)

        elif self.fsm\_state == self.CROSS\_MIDDLE\_LINE:

            # Viraj stânga un timp

            if (time.time() - self.state\_start\_time) < self.turn\_left\_time:

                twist = Twist()

                twist.linear.x = 0.2

                twist.angular.z = 0.3

                self.cmd\_vel\_pub.publish(twist)

            else:

                self.get\_logger().info("Done turning left => PASS\_OBSTACLE.")

                self.fsm\_state = self.PASS\_OBSTACLE

                self.obstacle\_behind\_count = 0

        elif self.fsm\_state == self.PASS\_OBSTACLE:

            # Mergi inainte până obstacolul e în spate-dreapta x cadre

            if self.lidar\_obstacle\_in\_right\_rear():

                self.obstacle\_behind\_count += 1

                self.get\_logger().info(f"Obstacle behind-dreapta frames= {self.obstacle\_behind\_count}")

            else:

                self.obstacle\_behind\_count = 0

            if self.obstacle\_behind\_count >= self.obstacle\_behind\_count\_needed:

                self.get\_logger().info("Obstacle in spate => CROSS\_BACK.")

                self.fsm\_state = self.CROSS\_BACK

                self.state\_start\_time = time.time()

            else:

                twist = Twist()

                twist.linear.x = 0.3

                twist.angular.z = 0.0

                self.cmd\_vel\_pub.publish(twist)

        elif self.fsm\_state == self.CROSS\_BACK:

            line\_x = self.detect\_yellow\_line\_center(cv\_image)

            width = cv\_image.shape[1]

            center\_x = width // 2

            OFFSET = 50  # Ajustează tu

            if line\_x >= 0 and line\_x < (center\_x - OFFSET):

                self.get\_logger().info("Line on left => back to LANE\_FOLLOW.")

                self.fsm\_state = self.LANE\_FOLLOW

            else:

                 twist = Twist()

                 twist.linear.x = 0.25

                 twist.angular.z = -0.35

                 self.cmd\_vel\_pub.publish(twist)

        # Debug

        blur = cv2.blur(cv\_image, (3,3))

        edge = cv2.Canny(blur, 160, 180)

        self.show\_debug(cv\_image, edge)

    # ---------------------------------------------------------------------

    # LANE FOLLOW

    # ---------------------------------------------------------------------

    def lane\_follow(self, cv\_image):

        blur = cv2.blur(cv\_image, (3,3))

        edge = cv2.Canny(blur, 160, 180)

        cx = self.freespace(edge, cv\_image)

        if cx >= 0:

            width = cv\_image.shape[1]

            center\_x = width // 2

            error = cx - center\_x

            self.filtered\_error = (self.alpha \* self.filtered\_error) + ((1 - self.alpha) \* error)

            angular\_z = -self.filtered\_error \* self.angular\_gain

            twist = Twist()

            twist.linear.x = self.linear\_speed

            twist.angular.z = angular\_z

            self.cmd\_vel\_pub.publish(twist)

        else:

            self.stop\_robot()

    def freespace(self, canny\_frame, img):

        height, width = canny\_frame.shape

        DreaptaLim = width // 2

        StangaLim = width // 2

        for i in range(width // 2, width-1):

            if canny\_frame[height - 10, i]:

                DreaptaLim = i

                break

        for i in range(width // 2):

            if canny\_frame[height - 10, width // 2 - i]:

                StangaLim = width // 2 - i

                break

        if StangaLim == width // 2:

            StangaLim = 1

        if DreaptaLim == width // 2:

            DreaptaLim = width - 1

        contour = []

        contour.append((StangaLim, height - 10))

        for j in range(StangaLim, DreaptaLim + 1, 10):

            for i in range(height - 10, 9, -1):

                if canny\_frame[i, j]:

                    contour.append((j, i))

                    break

                if i == 10:

                    contour.append((j, i))

        contour.append((DreaptaLim, height - 10))

        contours = [np.array(contour)]

        M = cv2.moments(contours[0])

        if M["m00"] != 0:

            cx = int(M["m10"] / M["m00"])

            cy = int(M["m01"] / M["m00"])

            cv2.arrowedLine(img, (width//2, height-10), (cx, cy), (60,90,255), 4)

            return cx

        else:

            return -1

    # ---------------------------------------------------------------------

    # LIDAR: obstacol in fata?

    # ---------------------------------------------------------------------

    def lidar\_obstacle\_in\_front(self):

        if self.latest\_scan\_ranges is None:

            return False

        n = self.num\_lidar\_rays

        if n == 0:

            return False

        # ±15° = 0..15 + n-15..n-1

        end\_front = min(15, n)

        for i in range(end\_front):

            if self.latest\_scan\_ranges[i] < self.lidar\_threshold:

                return True

        start\_back = max(n-15, 0)

        for i in range(start\_back, n):

            if self.latest\_scan\_ranges[i] < self.lidar\_threshold:

                return True

        return False

    # ---------------------------------------------------------------------

    # LIDAR: obstacol in spate-dreapta

    # ---------------------------------------------------------------------

    def lidar\_obstacle\_in\_right\_rear(self):

        """

        Return True daca TOT sectorul [300..359] + [0..30]

        e > self.lidar\_threshold => nimic aproape => obstacol e behind-right

        """

        if self.latest\_scan\_ranges is None:

            return False

        n = len(self.latest\_scan\_ranges)

        if n == 0:

            return False

        ranges = self.latest\_scan\_ranges

        good = True

        # 300..359

        startA = 300

        if startA >= n:

            startA = n-1

        for i in range(startA, n):

            if ranges[i] < self.lidar\_threshold:

                good = False

                break

        # 0..30

        if good:

            endB = min(30, n)

            for i in range(endB):

                if ranges[i] < self.lidar\_threshold:

                    good = False

                    break

        return good

    # ---------------------------------------------------------------------

    # DETECT STOP SIGN

    # ---------------------------------------------------------------------

    '''

    def detect\_stop\_sign(self, cv\_image):

        """

        Detectează semnul STOP utilizând culoarea roșie, forma octogonală și verificări suplimentare.

        """

        hsv = cv2.cvtColor(cv\_image, cv2.COLOR\_BGR2HSV)

        # Interval pentru roșu (ajustat pentru toleranță la marginea albă)

        lower\_red1 = np.array([0, 50, 50])

        upper\_red1 = np.array([10, 255, 255])

        lower\_red2 = np.array([170, 50, 50])

        upper\_red2 = np.array([180, 255, 255])

        mask1 = cv2.inRange(hsv, lower\_red1, upper\_red1)

        mask2 = cv2.inRange(hsv, lower\_red2, upper\_red2)

        red\_mask = cv2.bitwise\_or(mask1, mask2)

        # Morfologie pentru curățare

        kernel = np.ones((5, 5), np.uint8)

        red\_mask = cv2.morphologyEx(red\_mask, cv2.MORPH\_OPEN, kernel)

        red\_mask = cv2.morphologyEx(red\_mask, cv2.MORPH\_CLOSE, kernel)

        # Detectare contururi

        contours, \_ = cv2.findContours(red\_mask, cv2.RETR\_EXTERNAL, cv2.CHAIN\_APPROX\_SIMPLE)

        for cnt in contours:

            area = cv2.contourArea(cnt)

            if area < 100:  # Prag minim pentru dimensiunea semnului

                continue

            # Aproximare poligon

            approx = cv2.approxPolyDP(cnt, 0.03 \* cv2.arcLength(cnt, True), True)

            sides = len(approx)

            # Verificăm dacă are formă octogonală

            if 6 <= sides <= 10:

                x, y, w, h = cv2.boundingRect(approx)

                aspect\_ratio = w / float(h)

                # Validare pentru aspect ratio pătrat

                if 0.8 < aspect\_ratio < 1.2:

                    # Desenăm contur pe imagine

                    cv2.drawContours(cv\_image, [approx], -1, (0, 255, 255), 3)

                    # Opțional: verificare textură

                    roi = cv\_image[y:y + h, x:x + w]

                    if self.validate\_stop\_sign\_text(roi):

                        return True

        return False

    def validate\_stop\_sign\_text(self, roi):

        """

        Validare suplimentară pentru semnul STOP prin textură (simplu sau OCR).

        """

        gray = cv2.cvtColor(roi, cv2.COLOR\_BGR2GRAY)

        \_, binary = cv2.threshold(gray, 100, 255, cv2.THRESH\_BINARY)

        # Detectăm margini

        edges = cv2.Canny(binary, 50, 150)

        # Verificăm proporția pixelilor de margine

        non\_zero\_pixels = cv2.countNonZero(edges)

        total\_pixels = roi.shape[0] \* roi.shape[1]

        edge\_density = non\_zero\_pixels / total\_pixels

        # Dacă densitatea marginii e între un prag, e semn STOP

        return 0.02 < edge\_density < 0.2

       '''

    # ---------------------------------------------------------------------

    # DETECT LINIE GALBENA

    # ---------------------------------------------------------------------

    def detect\_yellow\_line\_center(self, cv\_image):

        hsv = cv2.cvtColor(cv\_image, cv2.COLOR\_BGR2HSV)

        lower\_y = np.array([20, 100, 100])

        upper\_y = np.array([30, 255, 255])

        mask = cv2.inRange(hsv, lower\_y, upper\_y)

        kernel = np.ones((5,5), np.uint8)

        mask = cv2.morphologyEx(mask, cv2.MORPH\_OPEN, kernel)

        mask = cv2.morphologyEx(mask, cv2.MORPH\_CLOSE, kernel)

        contours, \_ = cv2.findContours(mask, cv2.RETR\_EXTERNAL, cv2.CHAIN\_APPROX\_SIMPLE)

        if not contours:

            return -1

        max\_cnt = max(contours, key=cv2.contourArea)

        area = cv2.contourArea(max\_cnt)

        if area < 50:

            return -1

        M = cv2.moments(max\_cnt)

        if M["m00"] == 0:

            return -1

        cx = int(M["m10"] / M["m00"])

        return cx

    # ---------------------------------------------------------------------

    # STOP

    # ---------------------------------------------------------------------

    def stop\_robot(self):

        twist = Twist()

        twist.linear.x = 0.0

        twist.angular.z = 0.0

        self.cmd\_vel\_pub.publish(twist)

    def show\_debug(self, cv\_image, edge):

        cv2.imshow("Camera", cv\_image)

        if edge is not None:

            cv2.imshow("Edge", edge)

        cv2.waitKey(1)

def main(args=None):

    rclpy.init(args=args)

    node = LaneFollower()

    try:

        rclpy.spin(node)

    except KeyboardInterrupt:

        pass

    finally:

        node.destroy\_node()

        rclpy.shutdown()

        cv2.destroyAllWindows()

if \_name\_ == '\_main\_':

    main()

## 5. **Testare și punere în funcțiune**

**Testare în mediu simulare Gazebo**

* **Test 1**: Fără obstacole, doar linie galbenă. Robotul ar trebui să meargă drept pe bandă.
* **Test 2**: Așezarea unui obstacol (om) pe banda robotului. Robotul detectează obstacolul (LiDAR sub prag lidar\_threshold timp de *10* cadre consecutive), virând la stânga și apoi revenind pe bandă.
* **Test 3**: Mai multe obstacole, amplasate aleator. Se verifică dacă robotul continuă să înconjoare fiecare obstacol și să revină.
* **Test 4**: Ajustarea parametrilor angular\_gain, turn\_left\_time, lidar\_threshold, etc., pentru a obține o performanță optimă în scenariile de test.

## **6.Prezentarea firmei (echipei)**

**Firma / echipa dezvoltatoare**: *PathFinders SRL*

* **Membru 1**: Darius Niculescu
  + Rol: Implementarea logicii de urmărire a liniei, procesarea imaginilor (OpenCV).
  + Rol: Testare, simulare Gazebo, lansare pachete, documentare și încercarea (eșuată) de detectare semn STOP.
* **Membru 2**: Alexandru Mladin
  + Rol: Integrarea LiDAR, detectarea obstacolelor și implementarea FSM (finite state machine).
* **Membru 3**: Valentin-Razvan Plesa
  + Rol: Integrarea LiDAR, detectarea obstacolelor și implementarea FSM (finite state machine).

## **Concluzii**

Proiectul demonstrat oferă o bază solidă pentru înțelegerea și implementarea unor algoritmi de **urmărire a benzii** și **ocolire a obstacolelor** pe un robot mobil, folosind ROS2 Humble și Python.

* **Reușite**:
  + Robotul menține banda de mers și evită obstacole printr-o strategie simplă de *FSM*.
  + Codul este ușor extensibil și modular (detecția semnului STOP poate fi reactivată și îmbunătățită).
* **Limitări**:
  + Încercarea de detectare a semnului STOP nu este finalizată. În condiții reale, pot apărea probleme de luminozitate, rezoluție scăzută, multiple semne.
  + Sistemul de control pentru urmărirea liniei este destul de simplu (un “pseudo-PID”). Ar putea fi perfecționat.
* **Perspective de dezvoltare**:
  + Implementarea unui sistem robust de recunoaștere a semnelor de circulație (posibil cu rețele neuronale sau librării specializate).
  + Introducerea mai multor culori/bandă dublă, semafoare, semne rutiere diverse.
  + Migrarea la un sistem complex de planificare a traiectoriei în cazul unor medii cu multe obstacole și rute alternative.

## Bibliografie

1.Documentația oficială ROS2:

<https://docs.ros.org/en/humble/>

2.Tutoriale TurtleBot3:

<https://emanual.robotis.com/docs/en/platform/turtlebot3/quick-start/>

3.OpenCV (detectarea obiectelor, prelucrare de imagine):

<https://docs.opencv.org/>

4.Gazebo Simulator:

<http://gazebosim.org/tutorials>