**OBSTACLE AVOIDANCE**

**ALGORITHM:**

1. Initialize ROS 2 Node:

- Create a ROS 2 node named 'soccer\_obstacle\_avoidance'.

- Set up subscriptions for laser scan data ('laser\_scan') and create a publisher for velocity commands ('cmd\_vel').

- Initialize parameters such as `max\_linear\_speed`, `max\_angular\_speed`, `k\_att`, `k\_rep`, `repulsive\_distance`, and frame names.

2. Define Callback for Laser Scan Data:

- Upon receiving laser scan data:

- Transform the data from the laser frame to the robot's base link frame using TF2.

- Obtain the current robot position in the base link frame.

3. Calculate Attractive Forces:

- Calculate attractive forces toward the opponent and own goals:

- Transform the goal positions to the robot's base link frame.

- Calculate the distance to each goal.

- Calculate attractive forces using the formula

4. Calculate Repulsive Forces:

- Calculate repulsive forces based on laser scan data:

- For each laser scan point:

- If the distance is less than `repulsive\_distance`, calculate repulsive force.

- Sum up repulsive forces from all laser scan points.

5. Combine Forces:

- Combine attractive and repulsive forces to get the total force

6. Convert Forces to Velocities:

- Convert the total force to linear and angular velocities:

- Limit the linear velocity to `max\_linear\_speed`.

7. Publish Velocity Commands:

- Publish the calculated linear and angular velocities to the 'cmd\_vel' topic.

8. Handle Transformations and Exceptions:

- Handle exceptions related to TF (Transform) lookup, connectivity, and extrapolation to ensure robust operation.

- Log warning messages for any encountered issues.

9. Main ROS 2 Loop:

- Initialize the ROS 2 system.

- Start spinning the node to continuously process laser scan data and update velocity commands.

10. Shutdown ROS 2 Node:

- Upon exiting the program, destroy the ROS 2 node and shut down the ROS 2 system.

11. Localization and Transformations:

- Implement the `get\_robot\_position` function to obtain the current robot position:

- Utilize sensors or localization modules to get accurate robot position information.

- Implement the `transform\_point` function to transform a point between different frames:

- Use TF2 to handle the transformations between frames.

12. Testing and Adjustments:

- Test the code in a simulated environment:

- Verify that the robot exhibits expected behavior in terms of obstacle avoidance and goal following.

- Adjust parameters such as `k\_att`, `k\_rep`, `repulsive\_distance`, etc.:

- Fine-tune parameters based on robot dynamics and environmental conditions.

**PSEUDOCODE:**

import rclpy

from rclpy.node import Node

from sensor\_msgs.msg import LaserScan

from geometry\_msgs.msg import Twist, PointStamped, Point

import tf2\_ros

import tf2\_geometry\_msgs

import math

class SoccerObstacleAvoidance(Node):

def \_\_init\_\_(self):

super().\_\_init\_\_('soccer\_obstacle\_avoidance')

self.create\_subscription(LaserScan, '/laser\_scan', self.laser\_scan\_callback, 10)

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

self.tf\_buffer = tf2\_ros.Buffer()

self.tf\_listener = tf2\_ros.TransformListener(self.tf\_buffer, self)

self.max\_linear\_speed = 0.1

self.max\_angular\_speed = 0.5

self.k\_att = 0.1 # Adjust as needed

self.k\_rep = 0.5 # Adjust as needed

self.repulsive\_distance = 0.5 # Adjust as needed

self.laser\_frame = 'laser\_frame'

self.base\_link\_frame = 'base\_link'

self.opponent\_goal\_frame = 'opponent\_goal'

self.own\_goal\_frame = 'own\_goal'

def laser\_scan\_callback(self, msg):

try:

# Transform the laser scan data into the base link frame

laser\_data\_base\_link = self.transform\_laser\_data(msg, self.laser\_frame, self.base\_link\_frame)

# Get the current robot position in the base link frame

current\_position = self.get\_robot\_position(self.base\_link\_frame)

# Calculate attractive and repulsive forces

attractive\_force\_opponent = self.calculate\_attractive\_force(current\_position, self.opponent\_goal\_frame, self.base\_link\_frame)

attractive\_force\_own = self.calculate\_attractive\_force(current\_position, self.own\_goal\_frame, self.base\_link\_frame)

repulsive\_force = self.calculate\_repulsive\_force(laser\_data\_base\_link)

# Combine forces to get total force

total\_force = attractive\_force\_opponent + attractive\_force\_own + repulsive\_force

# Convert force to linear and angular velocities

linear\_velocity = min(total\_force.x, self.max\_linear\_speed)

angular\_velocity = 0.0 # Adjust as needed

# Publish velocities to robots

self.publish\_velocities(linear\_velocity, angular\_velocity)

except (tf2\_ros.LookupException, tf2\_ros.ConnectivityException, tf2\_ros.ExtrapolationException) as e:

self.get\_logger().warn(f"Transform lookup failed: {e}")

def transform\_laser\_data(self, laser\_data, source\_frame, target\_frame):

# Transform laser scan data from the source frame to the target frame

transformed\_laser\_data = LaserScan()

for i in range(len(laser\_data.ranges)):

point\_source\_frame = PointStamped()

point\_source\_frame.header.frame\_id = source\_frame

# Set point coordinates based on laser scan data

angle = laser\_data.angle\_min + i \* laser\_data.angle\_increment

point\_source\_frame.point.x = laser\_data.ranges[i] \* math.cos(angle)

point\_source\_frame.point.y = laser\_data.ranges[i] \* math.sin(angle)

# Perform the transformation

point\_target\_frame = self.tf\_buffer.transform(point\_source\_frame, target\_frame)

# Update the transformed laser data

transformed\_laser\_data.ranges.append(math.sqrt(point\_target\_frame.point.x\*\*2 + point\_target\_frame.point.y\*\*2))

# ...

return transformed\_laser\_data

def calculate\_attractive\_force(self, current\_position, goal\_frame, target\_frame):

# Calculate attractive force towards the goal in the target frame

goal\_position\_target\_frame = self.transform\_point(goal\_frame, target\_frame)

distance\_to\_goal = math.sqrt((goal\_position\_target\_frame.x - current\_position.x)\*\*2 +

(goal\_position\_target\_frame.y - current\_position.y)\*\*2)

attractive\_force\_magnitude = self.k\_att \* distance\_to\_goal # Adjust k\_att as needed

attractive\_force\_direction = math.atan2(goal\_position\_target\_frame.y - current\_position.y,

goal\_position\_target\_frame.x - current\_position.x)

attractive\_force\_x = attractive\_force\_magnitude \* math.cos(attractive\_force\_direction)

attractive\_force\_y = attractive\_force\_magnitude \* math.sin(attractive\_force\_direction)

attractive\_force = Point(x=attractive\_force\_x, y=attractive\_force\_y)

return attractive\_force

def calculate\_repulsive\_force(self, laser\_data):

# Calculate repulsive force based on laser scan data

repulsive\_force = Point(x=0.0, y=0.0)

for i in range(len(laser\_data.ranges)):

if laser\_data.ranges[i] < self.repulsive\_distance:

repulsive\_force\_magnitude = self.k\_rep \* (1.0 / laser\_data.ranges[i] - 1.0 / self.repulsive\_distance)

repulsive\_force\_direction = laser\_data.angle\_min + i \* laser\_data.angle\_increment

repulsive\_force\_x = repulsive\_force\_magnitude \* math.cos(repulsive\_force\_direction)

repulsive\_force\_y = repulsive\_force\_magnitude \* math.sin(repulsive\_force\_direction)

repulsive\_force.x += repulsive\_force\_x

repulsive\_force.y += repulsive\_force\_y

return repulsive\_force

def get\_robot\_position(self, target\_frame):

# Placeholder for obtaining the current position of the robot

# Replace with actual localization code

current\_position = Point(x=0.0, y=0.0)

return current\_position

def transform\_point(self, source\_frame, target\_frame):

# Transform a point from the source frame to the target frame

point\_source\_frame = PointStamped()

point\_source\_frame.header.frame\_id = source\_frame

# Set point coordinates

# Example: Set the coordinates based on the robot's sensors or localization

point\_source\_frame.point.x = 1.0

point\_source\_frame.point.y = 1.0

# Use tf2\_geometry\_msgs to perform the transformation

point\_target\_frame = self.tf\_buffer.transform(point\_source\_frame, target\_frame)

return point\_target\_frame.point

def publish\_velocities(self, linear\_velocity, angular\_velocity):

twist\_msg = Twist()

twist\_msg.linear.x = linear\_velocity

twist\_msg.angular.z = angular\_velocity

self.publisher\_.publish(twist\_msg)

def main(args=None):

rclpy.init(args=args)

obstacle\_avoidance\_node = SoccerObstacleAvoidance()

rclpy.spin(obstacle\_avoidance\_node)

obstacle\_avoidance\_node.destroy\_node()

rclpy.shutdown()

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

main()