Reporte Simulación Turtlesim

Aceves G. Julieta

Garay P. Ángeles

Gaona R. Eduardo

Department: Computación
Course: TE3002B - Implementación de Robótica Inteligente
Instructor: Benito Granados-Rojas, PhD.
Date: June 3, 2023

Abstract

In this project, 4 simulation exercises are carried out in Turtlesim, the ROS simulator for differential type traction robots. Exercises are developed based on the realization of rectangular and curved trajectories, speed changes, collision routes and obstacle avoidance. The functions "`orientate()`", "`go_to_goal()`", the topic pose, Twist() type speed message, and functions such as "`avoid_crash()`" are used, in addition to the Euclidean distance formula which is also used.

1 Introducción

Este proyecto se enfoca en el desarrolllo de 4 ejercicios de simulación en Ros que forman parte del Módulo 1: Robots móviles terrestres. Estos ejercicios están basados principalmente en la evasión de obstáculos y la identificiación de los modelos de cinemática inversa y directa de un robot móvil con tracción de tipo diferencial. Para esto se utilizó Turtlesim, este simulador es una ventana gráfica que muestra un robot en forma de tortuga, los robots utilizados en turtlesim se llaman turtlebots. Turtlesim es un simulador básico de un robot tipo diferencial. Un robot con tracción diferencial es un vehículo que utiliza un sistema de transmisión de dos ruedas independientes, esto quiere decir que cada rueda se encuentra unida a un motor propio, por lo que en consecuencia su locomoción se basa en la diferencia de velocidades de las dos ruedas, aparte, también se coloca una rueda libre o castor, ya sea en la parte trasera o delantera, la cual gira pasivamente y funciona para darle estabilidad al robot. Sobre la cinemática de este tipo de robots, se tiene el modelo cinemático diferencial directo, el cual funciona para determinar la localización del robot móvil, dicho modelo relaciona las velocidades del punto de interés o control, con las velocidades de los actuadores, considerando al vehículo como una masa puntual, es decir, sin analizar las fuerzas que ejercen sobre este, como fuerza de rozamiento e inercia. El desarrollo de las simulaciones 0, 1 y 2 (se explican posteriormente) nos llevan a un objetivo principal, el ejercicio 3, el cual consiste en programar una tortuga capaz de recorrer una trayectoria en la que evite impactar otras tortugas, es decir, la meta general de este proyecto es la evasión de obstáculos.

2 Procedimientos y Resultados

2.1 Simulación 0. Trayectorias rectangulares y curvas

El objetivo de esta simulación es lograr que la tortuga realice una trayectoria rectangular dentro del espacio de trabajo del turtlesim. Para ello, el primer paso fue realizar las siguientes modificaciones en el programa proveído por el profesor:

- En la función "orientate()": Añadir dos condiciones para saber en que sen-

tido era más conveniente que la tortuga girara, teniendo en cuenta su ángulo actual y el ángulo al que debía llegar, esto con el fin de suavizar los giros de la tortuga lo más posible.

- En la función "`go_to_goal()`": También se agregaron las condiciones para determinar en qué sentido era mejor que la tortuga girara, debido a que nos dimos cuenta mientras hacíamos las pruebas, que era necesario que también se agregaran en esta parte para corregir la trayectoria de la tortuga, de manera que se fuera lo más derecho posible.
- Finalmente, en la parte principal del código, el enfoque que decidimos usar fue el de una máquina de estados, definiendo las cuatro coordenadas de las esquinas del rectángulo, de manera que la tortuga primero se orientara con "orientate()", y luego se dirigiera hacia la esquina que le queda enfrente con "`go_to_goal()`", repitiendo este proceso cuatro veces hasta llegar a su posición original.

  Como se sabe que el espacio de trabajo de turtlesim va de 0 a 11 en "x" y en "y". Las coordenadas escogidas para el rectángulo fueron (1,1),(8,1),(8,10) y (1,10). Es importante mencionar que para esta y las siguientes simulaciones, la posición inicial de la tortuga se define en el archivo ".launch".

2.1.1 Código

```
#! /usr/bin/env python

import rospy
from geometry_msgs.msg import Twist
from turtlesim.msg import Pose
import math
import time
from std_srvs.srv import Empty

x = 0
y = 0
z = 0
theta = 0

def poseCallback(pose_message):
    global x
    global y
    global z
    global theta

    x = pose_message.x
```

```
    y = pose_message.y
    theta = pose_message.theta

def orientate (xgoal, ygoal):
    global x
    global y
    global theta

    velocity_message = Twist()
    cmd_vel_topic = '/turtle1/cmd_vel'

    while(True):
        ka = 4.0
        desired_angle_goal = math.atan2(ygoal-y, xgoal-x)
        dtheta = desired_angle_goal-theta
        aux = 2*math.pi-dtheta
        if dtheta > math.pi:
            dtheta = dtheta - 2*math.pi
        elif dtheta < -math.pi:
            dtheta = dtheta = 2*math.pi
        angular_speed = ka * (dtheta)
        velocity_message.angular.z = angular_speed
        velocity_message.linear.x = 0.0
        velocity_publisher.publish(velocity_message)
        print(dtheta)
        if (dtheta < 0.03):
            break

def go_to_goal (xgoal, ygoal):
    global x
    global y
    global theta

    velocity_message = Twist()
    cmd_vel_topic = '/turtle1/cmd_vel'

    while(True):
        kv = 0.5
        distance = abs(math.sqrt(((xgoal-x)**2)+((ygoal-y)**2)))
        linear_speed = kv * distance

        ka = 4.0
        desired_angle_goal = math.atan2(ygoal-y, xgoal-x)
        dtheta = desired_angle_goal-theta
        if dtheta > math.pi:
            dtheta = dtheta - 2*math.pi
        elif dtheta < -math.pi:
            dtheta = dtheta = 2*math.pi
        angular_speed = ka * (dtheta)

        velocity_message.linear.x = linear_speed
        velocity_message.angular.z = angular_speed
        velocity_publisher.publish(velocity_message)
        print(dtheta)
        if (distance < 0.01):
            break
```

```
if __name__ == '__main__':
    try:

        rospy.init_node('turtlesim_motion_pose', anonymous = True)

        cmd_vel_topic = '/turtle1/cmd_vel'
        velocity_publisher = rospy.Publisher(cmd_vel_topic, Twist, queue_size = 10)

        position_topic = "/turtle1/pose"
        pose_subscriber = rospy.Subscriber(position_topic, Pose, poseCallback)

        time.sleep(2.0)
        orientate(8,1)
        time.sleep(1.0)
        go_to_goal(8,1)
        time.sleep(1.0)

        orientate(8,10)
        time.sleep(1.0)
        go_to_goal(8,10)
        time.sleep(1.0)

        orientate(1,10)
        time.sleep(1.0)
        go_to_goal(1,10)
        time.sleep(1.0)

        orientate(1,1)
        time.sleep(1.0)
        go_to_goal(1,1)
        time.sleep(1.0)
        orientate(8,1)

    except rospy.ROSInterruptException:
        pass
```

2.2 Simulación 1. Cambio de velocidad

Esta simulación consiste en programar dos tortugas que se encuentran en franca ruta de colisión, donde una de ellas es capaz de frenar y/o retroceder antes de colisionar con la otra: la tortuga "bot", la cual solo se mueve de un lado a otro. Debido a que esta simulación requiere de dos tortugas, en este caso se programaron 2 nodos.

Para el código en la sección 2.2.1, los cambios que se realizaron con respecto al código de la sección 2.1.1, fue el cambio de las coordenadas que definen la trayectoria de la tortuga, siendo ahora una línea recta de (1,5) a (10,5) y de regreso, y la publicación de su tópico de posición "/turtle4/pose".

Ahora bien, para el código de la sección 2.2.2, lo primero que se hizo fue la subscripción al tópico "/turtle4/pose", para que la tortuga pudiera identificar cuando la tortuga "bot" estuviera cerca, y en nuestro caso, retrocediera. Para lograr que la tortuga retrocediera, la segunda modificación que se hizo fue en la función

"go_to_goal()", donde usando la fórmula de la distancia euclidiana se calcula la distancia entre las dos tortugas, y cuando esta distancia es menor a 1.2 unidades, la velocidad lineal en "x" se multiplica por -0.3; para que retroceda y desacelere, y así publique el mensaje de velocidad tipo "Twist()" durante 1.5 segundos. Finalmente, el último cambio que se hizo fue el de la trayectoria de la tortuga, la cual en este caso va en una línea recta de (5,1) a (5,10) y de regreso.

2.2.1 Código para tortuga bot

```
#! /usr/bin/env python

import rospy
from geometry_msgs.msg import Twist
from turtlesim.msg import Pose
import math
import time
from std_srvs.srv import Empty

x = 0
y = 0
z = 0
theta = 0

def poseCallback(pose_message):
    global x
    global y
    global z
    global theta

    x = pose_message.x
    y = pose_message.y
    theta = pose_message.theta

def orientate (xgoal, ygoal):
    global x
    global y
    global theta

    velocity_message = Twist()
    cmd_vel_topic = '/turtle4/cmd_vel'

    while(True):
        ka = 4.0
        desired_angle_goal = math.atan2(ygoal-y, xgoal-x)
        dtheta = desired_angle_goal-theta
        aux = 2*math.pi-dtheta
        if dtheta > math.pi:
            dtheta = dtheta - 2*math.pi
        elif dtheta < -math.pi:
            dtheta = dtheta = 2*math.pi
        angular_speed = ka * (dtheta)
        velocity_message.angular.z = angular_speed
        velocity_message.linear.x = 0.0
        velocity_publisher.publish(velocity_message)
        print(dtheta)
```

```
        if (dtheta < 0.03):
            break

def go_to_goal (xgoal, ygoal):
    global x
    global y
    global theta

    velocity_message = Twist()
    cmd_vel_topic = '/turtle4/cmd_vel'

    while(True):
        kv = 0.5
        distance = abs(math.sqrt(((xgoal-x)**2)+((ygoal-y)**2)))
        linear_speed = kv * distance

        ka = 3.0
        desired_angle_goal = math.atan2(ygoal-y, xgoal-x)
        dtheta = desired_angle_goal-theta
        if dtheta > math.pi:
            dtheta = dtheta - 2*math.pi
        elif dtheta < -math.pi:
            dtheta = dtheta = 2*math.pi
        angular_speed = ka * (dtheta)

        velocity_message.linear.x = linear_speed
        velocity_message.angular.z = angular_speed
        velocity_publisher.publish(velocity_message)
        print(dtheta)
        if (distance < 0.01):
            break

if __name__ == '__main__':
    try:

        rospy.init_node('turtlesim_motion_pose', anonymous = True)

        cmd_vel_topic = '/turtle4/cmd_vel'
        velocity_publisher = rospy.Publisher(cmd_vel_topic, Twist, queue_size = 10)

        position_topic_pub = "/turtle4/pose"
        pose_subscriber = rospy.Publisher(position_topic_pub, Pose, queue_size=10)

        position_topic_sub = "/turtle4/pose"
        pose_subscriber = rospy.Subscriber(position_topic_sub, Pose, poseCallback)
        time.sleep(2)

        time.sleep(3.0)
        orientate(1,5)
        time.sleep(1.0)
        go_to_goal(1,5)
        time.sleep(1.0)

        orientate(10,5)
        time.sleep(1.0)
        go_to_goal(10,5)
```

```
        time.sleep(1.0)

        orientate(1,1)
        time.sleep(2.0)
        orientate(1,5)
        time.sleep(1.0)
        go_to_goal(1,5)
        time.sleep(1.0)

    except rospy.ROSInterruptException:
        pass
```

2.2.2 Código para tortuga que retrocede

```
#! /usr/bin/env python

import rospy
from geometry_msgs.msg import Twist
from turtlesim.msg import Pose
import math
import time
from std_srvs.srv import Empty

x = 0
y = 0
z = 0
theta = 0

def poseCallback(pose_message):
    global x
    global y
    global z
    global theta

    x = pose_message.x
    y = pose_message.y
    theta = pose_message.theta


def dumbposeCallback(pose_messaged):
    global dumbx
    global dumby
    global dumbz
    global dumbtheta

    dumbx = pose_messaged.x
    dumby = pose_messaged.y
    dumbtheta = pose_messaged.theta

    print("Dumb x:", dumbx)
    print("Dumb y:", dumby)
    print("Dumb theta:", dumbtheta)
```

```
def orientate (xgoal, ygoal):
    global x
    global y
    global theta

    velocity_message = Twist()
    cmd_vel_topic = '/turtle3/cmd_vel'

    while(True):
        ka = 5.0
        desired_angle_goal = math.atan2(ygoal-y, xgoal-x)
        dtheta = desired_angle_goal-theta
        aux = 2*math.pi-dtheta
        if dtheta > math.pi:
            dtheta = dtheta - 2*math.pi
        elif dtheta < -math.pi:
            dtheta = dtheta = 2*math.pi
        angular_speed = ka * (dtheta)
        velocity_message.angular.z = angular_speed
        velocity_message.linear.x = 0.0
        velocity_publisher.publish(velocity_message)
        print(dtheta)
        if (dtheta < 0.03):
            break

def go_to_goal (xgoal, ygoal):
    global x
    global y
    global theta

    velocity_message = Twist()
    cmd_vel_topic = '/turtle3/cmd_vel'

    while(True):
        kv = 0.5
        distance = abs(math.sqrt(((xgoal-x)**2)+((ygoal-y)**2)))
        linear_speed = kv * distance

        ka = 4.0
        desired_angle_goal = math.atan2(ygoal-y, xgoal-x)
        dtheta = desired_angle_goal-theta

        if dtheta > math.pi:
            dtheta = dtheta - 2*math.pi
        elif dtheta < -math.pi:
            dtheta = dtheta = 2*math.pi
        angular_speed = ka * (dtheta)

        dumbdist = math.sqrt(((dumbx- x)**2)+((dumby-y)**2))
        rospy.loginfo(dumbdist)

        if  dumbdist < 1.2:
            velocity_message.linear.x = linear_speed * -0.3
```

```
            velocity_publisher.publish(velocity_message)
            time.sleep(1.5)

        velocity_message.linear.x = linear_speed
        velocity_message.angular.z = angular_speed
        velocity_publisher.publish(velocity_message)
        if (distance < 0.01):
            break

if __name__ == '__main__':
    try:

        rospy.init_node('turtlesim_motion_pose', anonymous = True)

        cmd_vel_topic = '/turtle3/cmd_vel'
        velocity_publisher = rospy.Publisher(cmd_vel_topic, Twist, queue_size = 10)

        position_topic = "/turtle3/pose"
        pose_subscriber = rospy.Subscriber(position_topic, Pose, poseCallback)

        position_topic_pub = "/turtle3/pose"
        pose_subscriber = rospy.Publisher(position_topic_pub, Pose, queue_size=10)

        dummy_position_topic = "/turtle4/pose"
        pose_subscriber = rospy.Subscriber(dummy_position_topic, Pose, dumbposeCallback)

        time.sleep(2.0)
        orientate(5,1)
        time.sleep(1.0)
        go_to_goal(5,1)
        time.sleep(1.0)

        orientate(5,10)
        time.sleep(1.0)
        go_to_goal(5,10)
        time.sleep(1.0)

        orientate(10,10)
        time.sleep(2.0)
        orientate(5,1)
        time.sleep(1.0)
        go_to_goal(5,1)
        time.sleep(1.0)


    except rospy.ROSInterruptException:
        pass
```

2.3 Simulación 2. Evasión de Obstáculos

El objetivo de esta simulación es programar dos tortugas de manera que sigan una trayectoria en línea recta. Teniendo como condiciones que la velocidad lineal de la tortuga "bot" debe ser determinada a partir de una "kv"

con un valor aleatorio dentro de un rango adecuado para la simulación; en nuestro caso de 0.1 a 0.8, y que la tortuga principal debe generar una trayectoria alternativa para evitar la colisión.

Siendo la trayectoria de la tortuga "bot" de (5,10) a (5,1), y la de la tortuga principal de (5,1) a (5,10). Por ello, el único cambio que se le hizo al código de la sección 2.3.1 con respecto al de la sección 2.2.1, fue el de la trayectoria. Mientras que, al código de la sección 2.3.2, con respecto al de la sección 2.2.2 se le añadieron dos nuevas funciones "avoid()" y "`avoid_crash()`". Además, mediante las simulaciones de prueba se fueron ajustando los valores de "kv" y "ka" para cada tortuga.

- avoid(): Se manda a llamar cuando la distancia; también se calcula con la fórmula de distancia euclidiana, entre las tortugas es menor a 2.7 unidades, y genera coordenadas para una trayectoria en forma triangular, usando como referencia las posiciones en "x" y "y" de la tortuga principal, y manda a llamar "`avoid_crash()`".
- `avoid_crash()`: Controla el moviento de la tortuga de manera similar a "`go_to_goal()`", cuando la tortuga sigue la trayectoria alternativa.

2.3.1 Código para tortuga bot

```
#! /usr/bin/env python

import rospy
from geometry_msgs.msg import Twist
from turtlesim.msg import Pose
import math
import time
from std_srvs.srv import Empty
import random

x = 0
y = 0
z = 0
theta = 0

def poseCallback(pose_message):
    global x
    global y
    global z
    global theta

    x = pose_message.x
    y = pose_message.y
    theta = pose_message.theta

def orientate (xgoal, ygoal):
    global x
    global y
    global theta

    velocity_message = Twist()
```

```
    cmd_vel_topic = '/turtle6/cmd_vel'

    while(True):
        ka = 5.0
        desired_angle_goal = math.atan2(ygoal-y, xgoal-x)
        dtheta = desired_angle_goal-theta
        aux = 2*math.pi-dtheta
        if dtheta > math.pi:
            dtheta = dtheta - 2*math.pi
        elif dtheta < -math.pi:
            dtheta = dtheta = 2*math.pi
        angular_speed = ka * (dtheta)
        velocity_message.angular.z = angular_speed
        velocity_message.linear.x = 0.0
        velocity_publisher.publish(velocity_message)
        print(dtheta)
        if (dtheta < 0.03):
            break

def go_to_goal (xgoal, ygoal):
    global x
    global y
    global theta

    velocity_message = Twist()
    cmd_vel_topic = '/turtle6/cmd_vel'

    while(True):
        kv = random.uniform(0.1,0.8)
        distance = abs(math.sqrt(((xgoal-x)**2)+((ygoal-y)**2)))
        linear_speed = kv * distance

        ka = 4.0
        desired_angle_goal = math.atan2(ygoal-y, xgoal-x)
        dtheta = desired_angle_goal-theta
        if dtheta > math.pi:
            dtheta = dtheta - 2*math.pi
        elif dtheta < -math.pi:
            dtheta = dtheta = 2*math.pi
        angular_speed = ka * (dtheta)

        velocity_message.linear.x = linear_speed
        velocity_message.angular.z = angular_speed
        velocity_publisher.publish(velocity_message)
        print(dtheta)
        if (distance < 0.01):
            break

if __name__ == '__main__':
    try:

        rospy.init_node('turtlesim_motion_pose', anonymous = True)

        cmd_vel_topic = '/turtle6/cmd_vel'
        velocity_publisher = rospy.Publisher(cmd_vel_topic, Twist, queue_size = 10)
```

```
        position_topic_pub = "/turtle6/pose"
        pose_subscriber = rospy.Publisher(position_topic_pub, Pose, queue_size=10)

        position_topic_sub = "/turtle6/pose"
        pose_subscriber = rospy.Subscriber(position_topic_sub, Pose, poseCallback)
        time.sleep(2)

        orientate(5,10)
        time.sleep(1.0)
        go_to_goal(5,10)
        time.sleep(1.0)

        orientate(5,1)
        time.sleep(1.0)
        go_to_goal(5,1)
        time.sleep(1.0)

    except rospy.ROSInterruptException:
        pass
```

2.3.2 Código para tortuga principal

```
#! /usr/bin/env python

import rospy
from geometry_msgs.msg import Twist
from turtlesim.msg import Pose
import math
import time
from std_srvs.srv import Empty

x = 0
y = 0
z = 0
theta = 0

def poseCallback(pose_message):
    global x
    global y
    global z
    global theta

    x = pose_message.x
    y = pose_message.y
    theta = pose_message.theta


def dumbposeCallback(pose_messaged):
    global dumbx
    global dumby
    global dumbz
    global dumbtheta
```

```
    dumbx = pose_messaged.x
    dumby = pose_messaged.y
    dumbtheta = pose_messaged.theta

    print("Dumb x:", dumbx)
    print("Dumb y:", dumby)
    print("Dumb theta:", dumbtheta)


def avoide_crash (xgoal, ygoal):
    global x
    global y
    global theta

    velocity_message = Twist()

    while(True):
        kv =2
        distance = abs(math.sqrt(((xgoal-x)**2)+((ygoal-y)**2)))
        linear_speed = kv * distance

        ka = 8.0
        desired_angle_goal = math.atan2(ygoal-y, xgoal-x)
        dtheta = desired_angle_goal-theta
        if dtheta > math.pi:
            dtheta = dtheta - 2*math.pi
        elif dtheta < -math.pi:
            dtheta = dtheta = 2*math.pi
        angular_speed = ka * (dtheta)
        dumbdist = math.sqrt(((dumbx- x)**2)+((dumby-y)**2))

        velocity_message.linear.x = linear_speed
        velocity_message.angular.z = angular_speed
        velocity_publisher.publish(velocity_message)
        print(dtheta)
        if (distance < 0.19):
            break

def avoid():
    dist=2
    orientate(x+dist,y+dist+0.5)
    time.sleep(0.3)
    avoide_crash(x+dist,y+dist+0.5)
    time.  sleep(5)
    orientate(x-dist,y+0.5)
    time.sleep(0.1)
    avoide_crash(x-dist,y+0.5)

def orientate (xgoal, ygoal):
    global x
    global y
    global theta

    velocity_message = Twist()
    cmd_vel_topic = '/turtle5/cmd_vel'
```

```
    while(True):
        ka = 5.0
        desired_angle_goal = math.atan2(ygoal-y, xgoal-x)
        dtheta = desired_angle_goal-theta
        if dtheta > math.pi:
            dtheta = dtheta - 2*math.pi
        elif dtheta < -math.pi:
            dtheta = dtheta = 2*math.pi
        angular_speed = ka * (dtheta)
        velocity_message.angular.z = angular_speed
        velocity_message.linear.x = 0.0
        velocity_publisher.publish(velocity_message)
        print(dtheta)
        if (dtheta < 0.03):
            break

def go_to_goal (xgoal, ygoal):
    global x
    global y
    global theta

    velocity_message = Twist()
    cmd_vel_topic = '/turtle5/cmd_vel'

    while(True):
        kv = 0.5#0.5
        distance = abs(math.sqrt(((xgoal-x)**2)+((ygoal-y)**2)))
        linear_speed = kv * distance

        ka = 5.0
        desired_angle_goal = math.atan2(ygoal-y, xgoal-x)
        dtheta = desired_angle_goal-theta

        if dtheta > math.pi:
            dtheta = dtheta - 2*math.pi
        elif dtheta < -math.pi:
            dtheta = dtheta = 2*math.pi
        angular_speed = ka * (dtheta)
        dumbdist = math.sqrt(((dumbx- x)**2)+((dumby-y)**2))

        if  dumbdist < 2.7:
            avoid()
        velocity_message.linear.x = linear_speed
        velocity_message.angular.z = angular_speed
        velocity_publisher.publish(velocity_message)
        print(dtheta)
        if (distance < 0.01):
            break

if __name__ == '__main__':
    try:

        rospy.init_node('turtlesim_motion_pose', anonymous = True)

        cmd_vel_topic = '/turtle5/cmd_vel'
        velocity_publisher = rospy.Publisher(cmd_vel_topic, Twist, queue_size = 10)
```

```
        position_topic = "/turtle5/pose"
        pose_subscriber = rospy.Subscriber(position_topic, Pose, poseCallback)

        position_topic_pub = "/turtle5/pose"
        pose_subscriber = rospy.Publisher(position_topic_pub, Pose, queue_size=10)

        dummy_position_topic = "/turtle6/pose"
        pose_subscriber = rospy.Subscriber(dummy_position_topic, Pose, dumbposeCallback)

        time.sleep(2.0)
        orientate(5,1)
        time.sleep(1.0)
        go_to_goal(5,1)
        time.sleep(1.0)

        orientate(5,10)
        time.sleep(1.0)
        go_to_goal(5,10)
        time.sleep(1.0)
    except rospy.ROSInterruptException:
        pass
```

2.4 Simulación 3. Entorno cooperativo

El objetivo de esta simulación es que se programen 4 tortugas, cada una con diferentes trayectorias, y que no choquen entre ellas ni colisionen con las "paredes" del espacio de trabajo del turtlesim.

Para lograr esto, se hicieron los nodos correspondientes para cada tortuga, asegurándonos de que cada una publicara su nodo de posición. Ahora bien, la tortuga con la que decidimos trabajar fue la 1; referida como "/turtle0" en los códigos, por lo que en su nodo, mostrado en la sección 2.4.1, nos aseguramos de suscribirla al tópico de posición de cada tortuga. De igual manera, la distancia hacia las otras tres tortugas se calculó utilizando la fórmula de distancia euclidiana, y para evitar las colisiones se implementaron las funciones "`direction_vector()`" y "`avoid_crash()`".

- `avoid_crash()`: Funciona de manera similar a "`go_to_goal()`", controlando el movimiento de la tortuga mientras esta se encuentra en la ruta alternativa.
- `direction_vector()`:Se manda a llamar cuando la distancia entre la tortuga 1 y alguna de las otras tres tortugas es menor a 2 unidades. Obtiene el vector de movimiento de la tortuga principal y el de la tortuga con la que va a chocar, posteriormente genera un nuevo vector de movimiento para la tortuga 1, sumando el vector de movimiento de la tortuga con la que va a colisionar y el vector opuesto de la tortuga 1, finalmente manda las componentes "x" y "y" del nuevo vector de movimiento a la función "`avoid_crash()`".

Para emplear el código para las demás tortugas solo hay que poner la trayectoria correspondiente y hacer las subscripciones correpondientes.

2.4.1 Código para tortuga

```python
#! /usr/bin/env python

import rospy
from geometry_msgs.msg import Twist
from turtlesim.msg import Pose
import math
import time
from std_srvs.srv import Empty

x = 0
y = 0
z = 0
theta = 0

def poseCallback(pose_message):
    global x
    global y
    global z
    global theta
    global cf1lin
    global cf1ang

    x = pose_message.x
    y = pose_message.y
    theta = pose_message.theta
    cf1lin = pose_message.linear_velocity
    cf1ang = pose_message.angular_velocity

def cf2Callback(pose_message):
    global cf2x
    global cf2y
    global cf2z
    global cf2theta
    global cf2lin
    global cf2ang

    cf2x = pose_message.x
    cf2y = pose_message.y
    cf2theta = pose_message.theta
    cf2lin = pose_message.linear_velocity
    cf2ang = pose_message.angular_velocity

def cf3Callback(pose_message):
    global cf3x
    global cf3y
    global cf3z
    global cf3theta
    global cf3lin
    global cf3ang

    cf3x = pose_message.x
    cf3y = pose_message.y
    cf3theta = pose_message.theta
    cf3lin = pose_message.linear_velocity
    cf3ang = pose_message.angular_velocity
```

```
def cf4Callback(pose_message):
    global cf4x
    global cf4y
    global cf4z
    global cf4theta
    global cf4lin
    global cf4ang

    cf4x = pose_message.x
    cf4y = pose_message.y
    cf4theta = pose_message.theta
    cf4lin = pose_message.linear_velocity
    cf4ang = pose_message.angular_velocity

def direction_vector(menace):
    linear_vel = cf1lin
    angular_vel = cf1ang
    turtle_heading = theta #heading angle

    if angular_vel != 0 :
        dt=0.3
        turtle_heading = theta+angular_vel * dt
        direction_vec = [math.cos(turtle_heading)+x, math.sin(turtle_heading)+y]#direction vector
        new_vect = [menace[0]*-1 + direction_vec[0], menace[1]*-1 + direction_vec[1]]
        print("direction:",direction_vec)
        print("menace:", menace)
        print("new_vect:",new_vect)
        avoide_crash(new_vect[0], new_vect[1])

def avoide_crash (xgoal, ygoal):
    global x
    global y
    global theta

    velocity_message = Twist()

    while(True):
        kv =0.6
        distance = abs(math.sqrt(((xgoal-x)**2)+((ygoal-y)**2)))
        linear_speed = kv * distance

        ka = 8.0
        desired_angle_goal = math.atan2(ygoal-y, xgoal-x)
        dtheta = desired_angle_goal-theta

        if dtheta > math.pi:
            dtheta = dtheta - 2*math.pi
        elif dtheta < -math.pi:
            dtheta = 2*math.pi
        angular_speed = ka * (dtheta)
        velocity_message.linear.x = linear_speed
        velocity_message.angular.z = angular_speed
        velocity_publisher.publish(velocity_message)
        if (distance < 0.1):
            break
```

```
def orientate (xgoal, ygoal):
    global x
    global y
    global theta

    velocity_message = Twist()

    while(True):
        ka = 6.0
        desired_angle_goal = math.atan2(ygoal-y, xgoal-x)
        dtheta = desired_angle_goal-theta
        if dtheta > math.pi:
            dtheta = dtheta - 2*math.pi
        elif dtheta < -math.pi:
            dtheta = 2*math.pi
        angular_speed = ka * (dtheta)
        velocity_message.angular.z = angular_speed
        velocity_message.linear.x = 0.0
        velocity_publisher.publish(velocity_message)
        if (dtheta < 0.03):
            break

def go_to_goal (xgoal, ygoal):
    global x
    global y
    global theta

    velocity_message = Twist()

    while(True):
        kv = 0.8
        distance = abs(math.sqrt(((xgoal-x)**2)+((ygoal-y)**2)))
        linear_speed = kv * distance

        ka = 6.0
        desired_angle_goal = math.atan2(ygoal-y, xgoal-x)
        dtheta = desired_angle_goal-theta
        if dtheta > math.pi:
            dtheta = dtheta - 2*math.pi
        elif dtheta < -math.pi:
            dtheta = 2*math.pi
        angular_speed = ka * (dtheta)

        cf2dist = math.sqrt(((cf2x- x)**2)+((cf2y-y)**2))
        cf3dist = math.sqrt(((cf3x- x)**2)+((cf3y-y)**2))
        cf4dist = math.sqrt(((cf4x- x)**2)+((cf4y-y)**2))
        down_wall = math.sqrt(((x-x)**2) + ((0.5-y)**2))
        print("down:",down_wall)
        up_wall = math.sqrt(((x-x)**2) + ((10.5-y)**2))
        print("up:",up_wall)
        r_wall = math.sqrt(((10.5-x)**2) + ((y-y)**2))
        print("right:",r_wall)
        l_wall = math.sqrt(((0.5-x)**2) + ((y-y)**2))
        print("left:",l_wall,"\n")

        if  (cf2dist < 2):
```

```
        heading = cf2theta
        t_h = heading + cf2ang * 0.1
        menace = [(math.cos(t_h))/cf2lin,(math.sin(t_h))/cf2lin]
        direction_vector(menace)

    elif (cf3dist < 2):
        if cf3lin != 0:
            heading = cf3theta
            t_h = heading + cf3ang * 0.1
            menace = [(math.cos(t_h))/cf3lin,(math.sin(t_h))/cf3lin]
            direction_vector(menace)
            print("cf3dist:", cf3dist)
        else:
            heading = cf3theta
            t_h = heading + cf3ang * 0.1
            menace = [(math.cos(t_h)),(math.sin(t_h))]
            direction_vector(menace)
            print("cf3dist:", cf3dist)
            pass

    elif (cf4dist < 2):

        heading = cf4theta
        t_h = heading + cf4ang * 0.1
        menace = [(math.cos(t_h))/cf4lin,(math.sin(t_h))/cf4lin]
        direction_vector(menace)
    elif (down_wall < 0.5):

        menace=[(x,1)]
        direction_vector(menace)

    elif (up_wall < 0.5):

        menace=[x,1.5]
        direction_vector(menace)

    elif (r_wall < 0.5):
        print("alv vas a CCHOCAR")
        time.sleep(1)
        menace=[10.5,y]
        direction_vector(menace)

    elif (l_wall < 0.5):

        menace=[-10,y]
        direction_vector(menace)

    else:

        velocity_message.linear.x = linear_speed
        velocity_message.angular.z = angular_speed
        velocity_publisher.publish(velocity_message)
    if (distance < 0.01):
        break
```

```
if __name__ == '__main__':
    try:

        rospy.init_node('turtlesim_motion_pose', anonymous = True)

        cmd_vel_topic = '/turtle0/cmd_vel'
        velocity_publisher = rospy.Publisher(cmd_vel_topic, Twist, queue_size = 10)

        position_topic = "/turtle0/pose"
        pose_subscriber = rospy.Subscriber(position_topic, Pose, poseCallback)

        position_topic_pub = "/turtle0/pose"
        pose_subscriber = rospy.Publisher(position_topic_pub, Pose, queue_size=10)

        cf2_position_topic = "/turtle2/pose"
        pose_subscriber = rospy.Subscriber(cf2_position_topic, Pose, cf2Callback)

        cf3_position_topic = "/turtle3/pose"
        pose_subscriber = rospy.Subscriber(cf3_position_topic, Pose, cf3Callback)

        cf4_position_topic = "/turtle4/pose"
        pose_subscriber = rospy.Subscriber(cf4_position_topic, Pose, cf4Callback)

        time.sleep(2.0)
        orientate(10,10)
        time.sleep(0.5)
        go_to_goal(10,10)
        time.sleep(0.5)

        orientate(1,5.5)
        time.sleep(0.5)
        go_to_goal(1,5.5)
        time.sleep(0.5)

        orientate(10,5.5)
        time.sleep(0.5)
        go_to_goal(10,5.5)
        time.sleep(0.5)

        orientate(1,1)
        time.sleep(0.5)
        go_to_goal(1,1)
        time.sleep(0.5)
        orientate(9,10)
        time.sleep(0.1)

    except rospy.ROSInterruptException:
        pass
```

2.5 Resultados

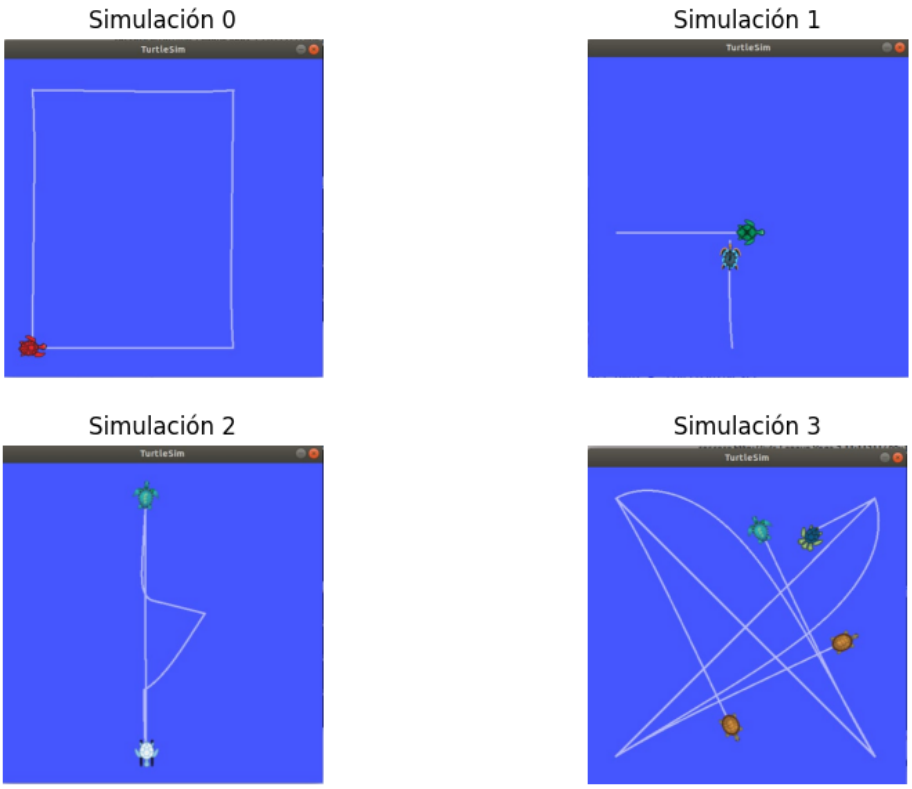

Figure 1: Simulaciones en turtlesim

.

La trayectoria rectangular de la simulación 0 se completó de manera correcta, aunque para las vueltas para orientar la tortuga tuvimos que modificar el valor de "ka" según lo observado en las simulaciones de prueba.

En la simulación 1, para lograr que la tortuga principal frenara y retrocediera en el momento adecuado, primero calculamos la distancia entre las tortugas cuando estas estuvieran en (5,5) para el bot, y (5,4) para la tortuga principal; la distancia es igual a 1 unidad, y a partir de este cálculo y las simulaciones de prueba fuimos adaptando la tolerancia a partir de la cual la tortuga principal retrocedería, la cual quedó fijada en 1.2 unidades.

El resultado de la simulación 2, para evasión de obstáculos fue satisfactorio, y aplicando una trayectoria alterna con forma triangular se logró evitar la colisión, al mismo tiempo que la tortuga principal se mantuvo en la región central del espacio de trabajo del turtlesim.

En la simulación 3, mientras se evitaron las colisiones entre las tortugas cuando todas tenían el método de evasión que decidimos implementar; con la función "`direction_vector()`", cuando pasamos al entorno colaborativo con los demás equipos del salón, notamos que había ciertos problemas con nuestra implementación, pues si la tortuga con la que va a colisionar se frena, su velocidad lineal es 0, lo que ocasiona un error en la normalización de su vector de movimiento.

3 Conclusión

Las simulaciones 0,1 y 2 se completaron de manera satisfactoria, sin embargo, en el entorno cooperativo de la simulación 3, nos percatamos de que nos faltó tomar en cuenta ciertos escenarios que se podrían presentar, como el que las tortugas se pararan, pues debido a que usamos la velocidad lineal para normalizar los vectores, esto generaba un error en nuestro código. Lo pasado, se podría haber prevenido si hubiéramos conversado con los demás equipos, para poder tomar en cuenta todos los posibles factores.

4 Repositorio

`https://github.com/AngelesGarayP/Turtlesim`