

Práctica 3

Programación de tareas en robots móviles



Realizado por:

Jorge Juan De Gomar Pina

Carlos Eduardo Fernández García

Índice

Archivos	1
Proyecto Probado	2
Adquisición de imágenes	2
Entorno de simulación	2
Detección de líneas	3
Seguimiento de líneas	4
Vídeo demostración	5
Proyecto Realizado	6
Mapa empleado	6
Señales detectadas	6
Inicialización y adquisición de imágenes	7
Función principal	7
Detección de líneas	8
Detección de señales	10
Control de movimientos	12
Vídeo proyecto	13
Pruebas en laboratorio	14
Práctica subida a GitHub	14

Archivos

- **follower_p.py** → archivo empleado en la prueba del “proyecto probado”
- **follower_modificado.py** → archivo empleado en el “proyecto realizado”
- **turtlebot3_autorace.world** → mapa empleado

Proyecto Probado

El proyecto que se probó fue el del capítulo 12 del libro “*Programming Robots with ROS*”. En dicho capítulo se muestran los pasos para que el robot siga una línea pintada en el suelo. Cabe destacar que se han realizado algunas modificaciones para que fuese compatible.

Adquisición de imágenes

La adquisición de imágenes se realiza mediante la suscripción al topic “**camera/image**” el cual proporciona la imagen de la cámara del turtlebot3 en ese momento.

```
class Follower:  
  
    def __init__(self):  
        self.bridge = cv_bridge.CvBridge()  
        cv2.namedWindow("window", 1)  
        self.image_sub = rospy.Subscriber('camera/image', Image, self.image_callback)  
        self.cmd_vel_pub = rospy.Publisher('cmd_vel', Twist, queue_size=1)  
        self.twist = Twist()
```

Imagen 1: adquisición de imágenes del robot

Entorno de simulación

Dado que no fue posible emplear el mismo entorno que en el proyecto, se empleó un entorno similar que contaba con una línea amarilla en el suelo (**autorace**)

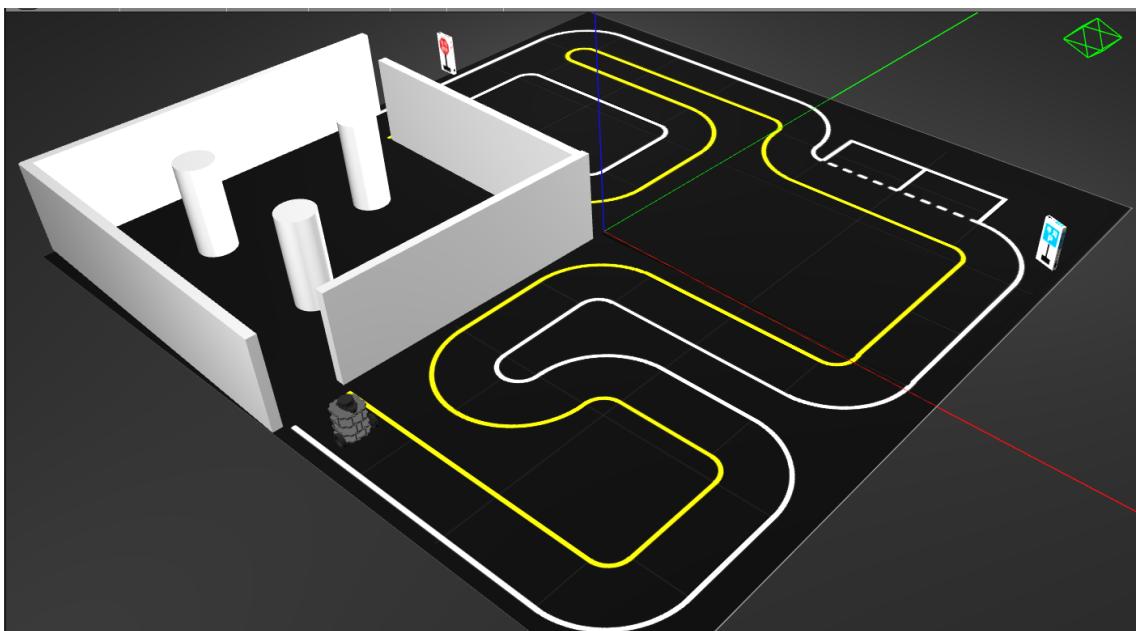


Imagen 2: entorno de simulación empleado

Detección de líneas

Para la detección de la línea amarilla se realiza el siguiente procedimiento:

- **Líneas 18-19** → se transforma la imagen obtenida de la cámara del turtlebot3 al formato de color “rgb” y seguidamente se realiza una copia en el espacio de color “hsv”.
- **Líneas 21-24** → se determina el rango de amarillo que se quiere detectar (lower_yellow y upper_yellow) y se crea una máscara binaria (mask) en la que los píxeles que se encuentren dentro del rango se pondrán en blanco y el resto negro
- **Líneas 26-32** → se extraen las dimensiones de la imagen (h: height y w: width), se determinan las zonas de la imágen en las que se quiere buscar la línea (search_top y search_bot) y se ignora todo lo que no esté dentro del espacio de búsqueda (se pone a “0”).
- **Líneas 34-40** → por último, se extraen los momentos de la zona de búsqueda, se determina el centro (cx y cy) y se dibuja un punto rojo.

```
16 def image_callback(self, msg):  
17     image = self.bridge.imgmsg_to_cv2(msg, desired_encoding='bgr8')  
18     hsv = cv2.cvtColor(image, cv2.COLOR_BGR2HSV)  
19  
20     lower_yellow = numpy.array([ 10,  10,  10])  
21     upper_yellow = numpy.array([255, 255, 250])  
22  
23     mask = cv2.inRange(hsv, lower_yellow, upper_yellow)  
24  
25     h, w, d = image.shape  
26  
27     search_top = 3*h/4  
28     search_bot = 3*h/4 + 20  
29  
30     mask[0:int(search_top), 0:w] = 0  
31     mask[int(search_bot):h, 0:w] = 0  
32  
33     M = cv2.moments(mask)  
34     if M['m00'] > 0:  
35         cx = int(M['m10']/M['m00'])  
36         cy = int(M['m01']/M['m00'])  
37         cv2.circle(image, (cx, cy), 20, (0,0,255), -1)  
38     else:  
39         cx = 0
```

Imagen 3: detección de la línea amarilla

Seguimiento de líneas

Una vez obtenidas las coordenadas del centro de la línea (“cx” y “cy”), dentro del espacio de búsqueda deseado, se calcula el error entre dichas coordenadas y la posición en la que se desearía que se encontrasen. En este caso, dado que se desea que el robot siga la línea de forma centrada, el error será la diferencia entre el centro de la línea y el centro de la imagen (línea 43).

```
42  # BEGIN CONTROL
43  err = cx - w/2
44  self.twist.linear.x = 0.1
45  self.twist.angular.z = -float(err) / 100
46  self.cmd_vel_pub.publish(self.twist)
47
48  # END CONTROL
49  cv2.imshow("window", image)
50  cv2.waitKey(3)
```

Imagen 4: control para el seguimiento de la línea

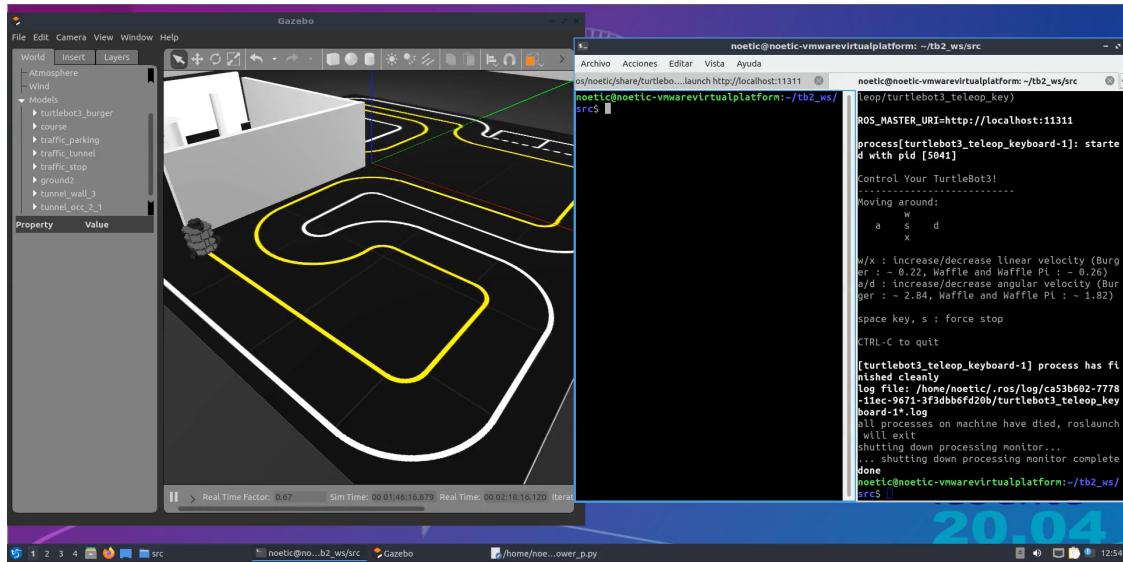
Seguidamente se determina la velocidad lineal del robot y la angular (teniendo en cuenta el error) para corregir la trayectoria y se publica en el topic “*cmd_vel*”. Por último se visualiza la imagen captada por la cámara del robot con los dibujos incorporados (punto rojo sobre línea amarilla).



Imagen 5: imagen final proporcionada

Vídeo demostración

A continuación se muestra un vídeo de demostración del funcionamiento del robot al ejecutar el código anteriormente explicado.



Enlace 1: [turtlebot3_seguimiento_linea_amarilla.mp4](#)

En un principio, dado que en el espacio de búsqueda aparecen dos líneas amarillas, una aproximadamente en el centro de la imagen y otra en el margen izquierdo, el momento calculado se sitúa entre ambas líneas, por lo que el robot pasa entre ambas hasta que una de ellas deja de ser detectada.

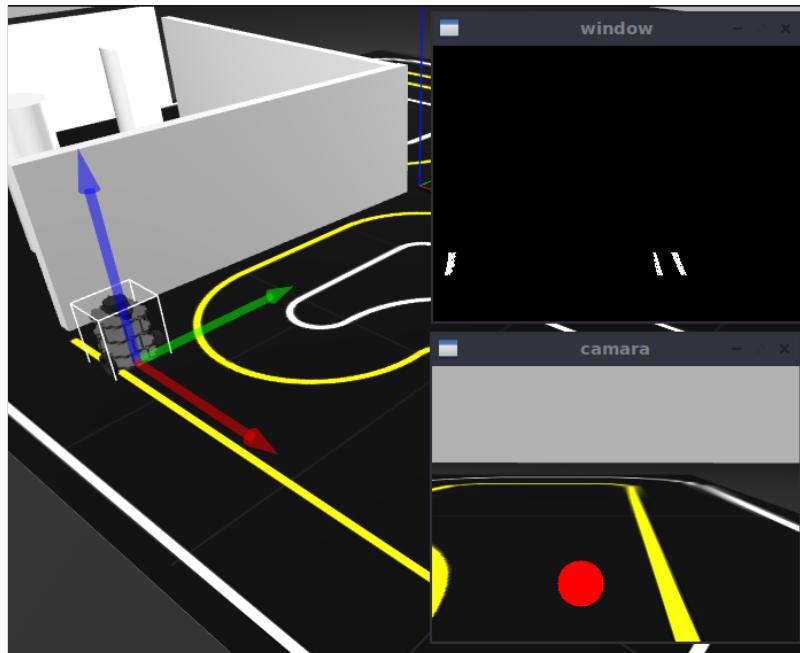


Imagen 6: fallo en el seguimiento de la línea amarilla

Este fallo en el seguimiento de la línea se debe a que el mapa empleado tiene partes en las que las líneas están cerca y que el espacio de búsqueda es demasiado ancho.

Proyecto Realizado

El proyecto realizado consiste en una modificación del proyecto anteriormente probado, pero incorporando una serie de tareas más complejas: detección de dos líneas, circular entre ambas y detección de señales.

El proyecto consiste en una simulación de un taxi autónomo que recoge a un pasajero en un parking y lo lleva hasta el final del camino. El parking y el final del camino están indicados con unas señales.

Mapa empleado

El mapa empleado es el mismo que en proyecto anterior (**autorace**). Cuenta con un camino delimitado por unas líneas (amarilla y blanca) y con unas señales.

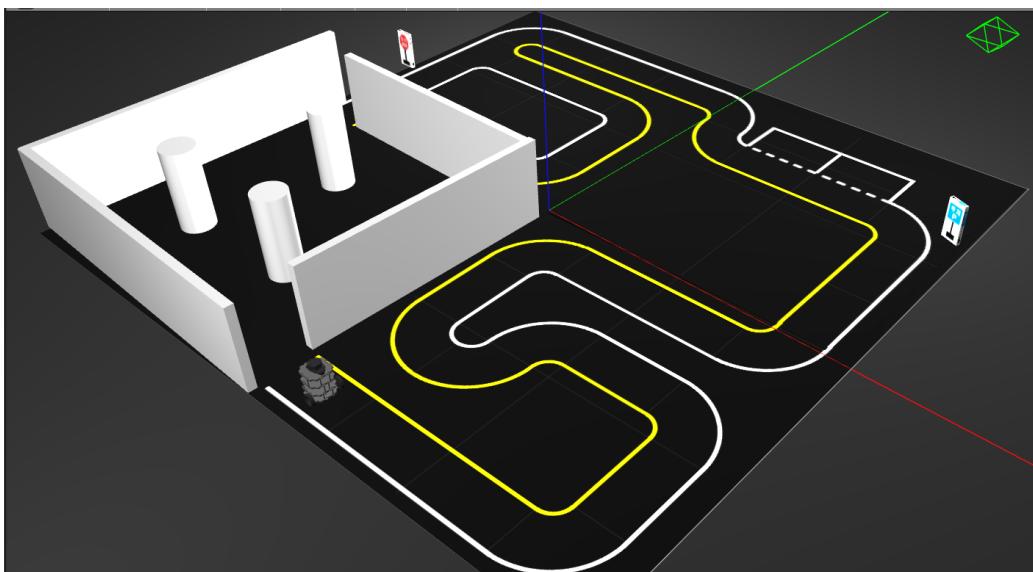


Imagen 7: entorno de simulación empleado en el proyecto

Señales detectadas

Inicialmente, las variables ***parking_state*** y ***stop_state***, determinan si se han encontrado las señales de *parking* y *stop*, por lo que inicialmente son falsas.

```
10 def __init__(self):
11
12     self.bridge = cv_bridge.CvBridge()
13     cv2.namedWindow("window", 1)
14
15     # Indicadores de detección de señales
16     parking_state = False
17     stop_state = False
18     self.parking_state(parking_state)
19     self.stop_state(stop_state)
```

Imagen 8: inicialización de *parking_state* y *stop_state*

Las funciones ***parking_state()*** y ***stop_state()*** almacenan de forma global el estado de las variables que determinan si se han detectado las señales

```
25 def parking_state (self, state):
26     global parking_detected
27     parking_detected = state
28
29 def stop_state (self, state):
30     global stop_detected
31     stop_detected = state
```

Imagen 9: funciones *parking_state()* y *stop_state*

Inicialización y adquisición de imágenes

Al igual que en el “Proyecto Probado”, las imágenes se obtienen al suscribirse al topic **“camera/image”**. Seguidamente, se ejecuta la función “principal”

```
10 def __init__(self):
11
12     self.bridge = cv_bridge.CvBridge()
13     cv2.namedWindow("window", 1)
14
15     # Indicadores de detección de señales
16     parking_state = False
17     stop_state = False
18     self.parking_state(parking_state)
19     self.stop_state(stop_state)
20
21     self.image_sub = rospy.Subscriber('camera/image', Image, self.principal)
22     self.cmd_vel_pub = rospy.Publisher('cmd_vel', Twist, queue_size=1)
23     self.twist = Twist()
```

Imagen 10: adquisición de las imágenes de la cámara del turtlebot3

Función principal

La función principal realiza secuencialmente llamadas a funciones secundarias y en función de la respuesta determina las acciones a realizar. La función principal se divide en los siguientes bloques:

1. **Preparación** → inicialmente, se transforma la imagen obtenida de la cámara del robot al espacio de color “rgb” y al “hsv”
2. **Detección de líneas** → se detectan las líneas amarilla y blanca que definen el camino
3. **Detección de señales** → se comprueba si hay alguna señal y si está cerca
4. **Control del movimiento** → en función de los resultados de los bloques anteriores, se determina el movimiento a realizar.

Detección de líneas

Como se ha podido comprobar anteriormente, la línea que delimita el camino por la izquierda es amarilla y por la derecha es blanca. Por ello se llama dos veces a la función para detectar ambas líneas, cada una con sus rangos de color correspondientes.

```
# Detección de Lineas
left = self.detect_line(image, numpy.array([10,10,10]), numpy.array([255,255,250]), 40, h, w, hsv)
right = self.detect_line(image, numpy.array([0,0,250]), numpy.array([0,0,255]), 280, h, w, hsv)
```

Imagen 11: llamadas a la función de detección de líneas

Se han probado diferentes rangos de colores hasta dar con los que mejores resultados proporcionan. La función crea un máscara binaria, dejando a en blanco los píxeles que se encuentren dentro del rango y en negro el resto. A continuación se determina la zona de búsqueda, poniendo a “0” (negro) todo lo que quede fuera de dicha zona, se obtienen las coordenadas del centro de la figura que forman los píxeles en blanco (mediante el cálculo de sus momentos) y se dibuja un punto rojo en dichas coordenadas. Tras realizar las dos llamadas, una para detectar cada línea, se obtienen dos coordenadas “x” (left, right).

```
84 # Detección de Linea
85 def detect_line (self, image, lower, upper, ideal_position, h, w, hsv):
86
87     mask = cv2.inRange(hsv, lower, upper)
88
89     x = ideal_position
90     y = 190
91
92     # Zona de búsqueda
93     search_top = 3*h/4
94     search_bot = 3*h/4 + 20
95
96     mask [0:int(search_top), 0:w] = 0
97     mask[int(search_bot):h, 0:w] = 0
98     M = cv2.moments(mask)
99     if M['m00'] > 0:
100         x = int(M['m10']/M['m00'])
101         y = int(M['m01']/M['m00'])
102         cv2.circle(image, (x, y), 20, (0,0,255), -1)
103     else:
104         cv2.circle(image, (x, y), 20, (0,255,0), -1)
105
106 return x
```

Imagen 12: función de detección de líneas

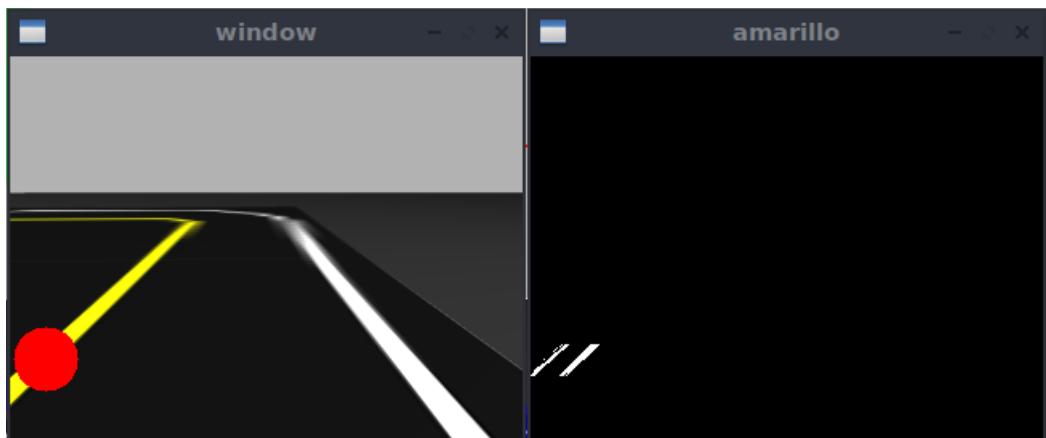


Imagen 13: detección de la línea amarilla

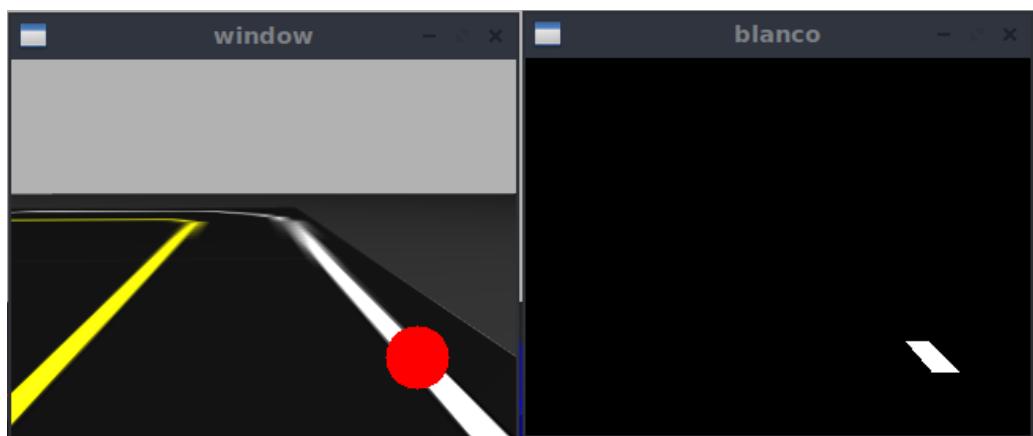


Imagen 14: detección de la línea blanca

En caso de no detectar una línea, como puede suceder al tomar una curva, se devuelve un valor por defecto (*ideal_position*) que permitirá posteriormente no perder el control del robot. El valor por defecto de la línea amarilla es 40 y de la blanca es 280.

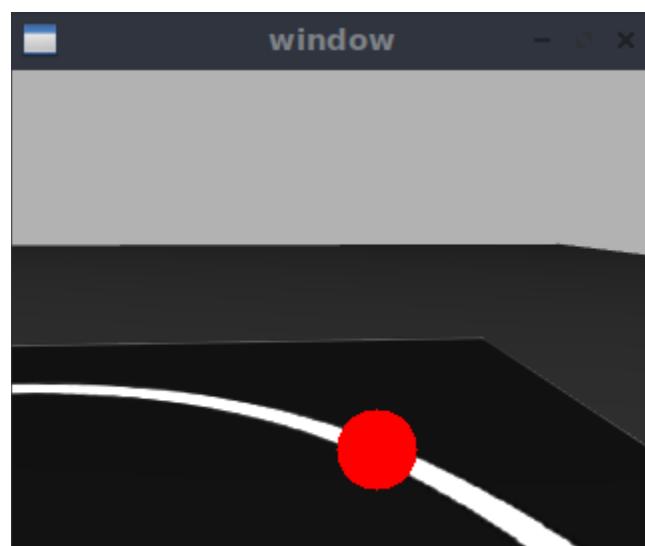


Imagen 15: detección de una única línea

Detección de señales

El robot reconoce dos señales, la señal de parking y la de stop. Cada señal le indica una orden al robot:



Imagen 16: señales que el robot busca

- **parking** → detiene el robot hasta que se presiona la tecla “ENTER”

```
Bienvenido, por favor pulse 'ENTER' para comenzar  
el trayecto ...
```

Imagen 17: mensaje al detectar la señal de parking

- **stop** → marca el final del trayecto

```
Detectado 'STOP', final del trayecto. Gracias por  
viajar con nosotros
```

Imagen 18: mensaje al detectar la señal de stop

Las señales se detectan en función de su color al igual que la detección de las líneas. Si la señal se encuentra en un rango de azul será la de parking y en un rango de rojo será la de stop. La función crea una máscara binaria dejando en blanco los píxeles que se encuentren dentro del rango de color deseado. La zona de búsqueda se limita únicamente al margen superior derecho. En dicho margen, se realizan unas transformaciones morfológicas empleando un elemento estructurante y se localiza el contorno de la figura. Posteriormente se calcula el área que forma ese contorno y se devuelve.

```
106 # Detección de Señales  
107 def detect_signal (self, image, lower, upper, h, w, hsv):  
108     mask = cv2.inRange(hsv, lower, upper)  
109  
110     # Zona de búsqueda: margen superior derecho  
111     mask[0:int(h/2), 0:int(w/2)] = 0  
112     mask[int(h/2):h, 0:w] = 0  
113  
114     # Detección del contorno y Cálculo del área  
115     kernel = cv2.getStructuringElement(cv2.MORPH_RECT, (3,3))  
116     opening = cv2.morphologyEx(mask, cv2.MORPH_OPEN, kernel, iterations=1)  
117  
118     contours = cv2.findContours(opening, cv2.RETR_EXTERNAL, cv2.CHAIN_APPROX_SIMPLE)  
119     contours = contours[0] if len(contours) == 2 else contours[1]  
120  
121     area = 0  
122     for pixel in contours:  
123         area += cv2.contourArea(pixel)  
124  
125     return area
```

Imagen 19: función de detección de señales

El principal problema de detectar las señales en función del color es que se puede dar el caso de detectarla a mucha distancia y realizar erróneamente la tarea predefinida. La solución para tomar en consideración la señal únicamente cuando esté cerca es calcular su área y considerarla detectada si supera un determinado valor.

```

48  # Detección de Señales
49  if parking_detected == False:
50      parking = self.detect_signal (image, numpy.array([90,50,70]), numpy.array([128,255,255]), h, w, hsv)
51  else:
52      parking = 0
53
54  # Señal de parking cerca
55  if parking > 3600 and parking_detected == False:
56      self.parking_state(True)
57      self.control (left, right, w, 0.0, 0.0)
58      print()
59      input("Bienbenido, por favor pulse 'ENTER' para comenzar el trayecto ...")
```

Imagen 20: consideración o no de la detección de la señal de parking

```

64  if stop_detected == False:
65      stop = self.detect_signal (image, numpy.array([159,50,70]), numpy.array([180,255,255]), h, w, hsv)
66  else:
67      stop = 0
68
69  # Señal de STOP CERCA
70  if stop > 1300 and stop_detected == False:
71      self.stop_state(True)
72      self.control (left, right, w, 0.0, 0.0)
73      print()
74      print ("Detectado 'STOP', final del trayecto. Gracias por viajar con nosotros")
```

Imagen 21: consideración o no de la detección de la señal de stop

En caso de que el área sea superior de un determinado valor y que no se haya detectado previamente la señal, se realizará la acción correspondiente. Una vez detectada una señal, en caso de volverla a detectar será ignorada. Esto evita realizar detecciones consecutivas no deseadas de la misma señal y repetir la acción correspondiente varias veces seguidas.



Imagen 22: comparación entre señal de parking no considerada y sí considerada

Enlace 2: [video_deteccion_señal_parking.mp4](#)

Enlace 3: [video_deteccion_señal_stop.mp4](#)

Control de movimientos

El robot varía sus movimientos en función de los resultados obtenidos en los bloques anteriores. Principalmente se pueden dividir los movimientos en dos tipos: parado y desplazándose. La diferencia entre ambos movimientos es que cuando se deseé que pare el robot, sus velocidades lineales y angulares serán 0.

Inicialmente, se calcula el error de angular como la diferencia entre la media de las posiciones “x” de las líneas del camino menos la posición deseada por el robot. Esto quiere decir que, dado que se quiere que el centro del robot esté entre las dos líneas y el centro del robot se corresponde con la mitad del ancho de la imagen ($w/2$), la distancia entre ambas será el error angular. Dicho error se representa a continuación con la línea morada.

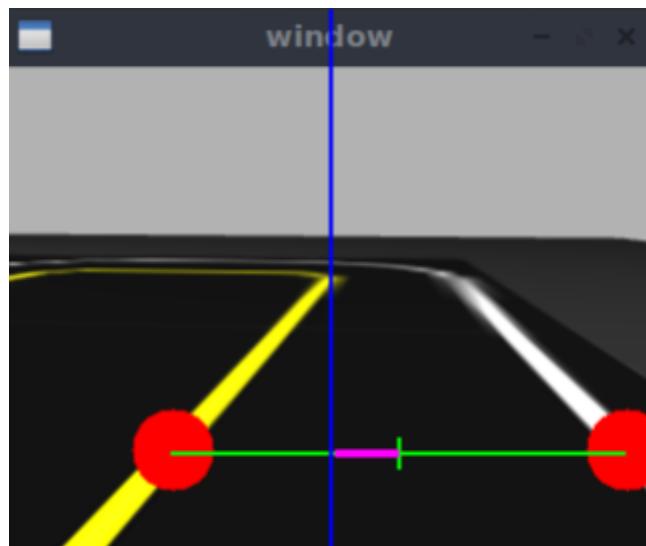


Imagen 23: ejemplo del error de angular

El robot irá corrigiendo su posición angular continuamente para situarse dentro del camino en todo momento. La velocidad lineal será constante mientras el robot se esté desplazando.

```
128  # Control del Robot
129  def control (self, left, right, w, vel, ang):
130
131      err = ((left+right)/2) - (w/2)
132      self.twist.linear.x = vel
133      self.twist.angular.z = -float(err) / 100 * ang
134      self.cmd_vel_pub.publish(self.twist)
```

Imagen 24: función de control del movimiento

Cabe destacar que el robot se desplazará mientras que la señal de stop no sea detectada, en caso contrario, el robot permanecerá permanentemente parado.

```
76      if stop_detected == True:
77          self.control (left, right, w, 0.0, 0.0)
78      else:
79          self.control (left, right, w, 0.1, 1)
```

Imagen 25: condición para que el robot se desplace

Dado que la posición angular se calcula teniendo en cuenta ambas líneas, en caso de no detectar una de ellas, como puede suceder en una curva, el robot se saldría del camino al calcular el error angular y girar. Para evitar que esto suceda, en caso de no detectar una línea, se establece una “posición imaginaria fija” de la línea (representada con un punto verde), de modo que el robot determinará el giro en función de la posición de la línea verdaderamente detectada (punto rojo). La posición imaginaria de la línea amarilla es 40 y de la blanca es 280.

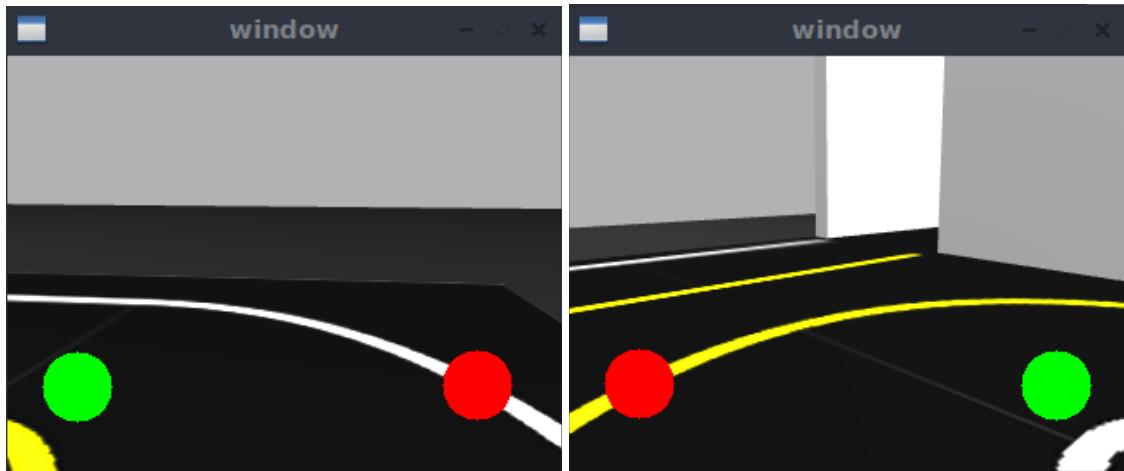
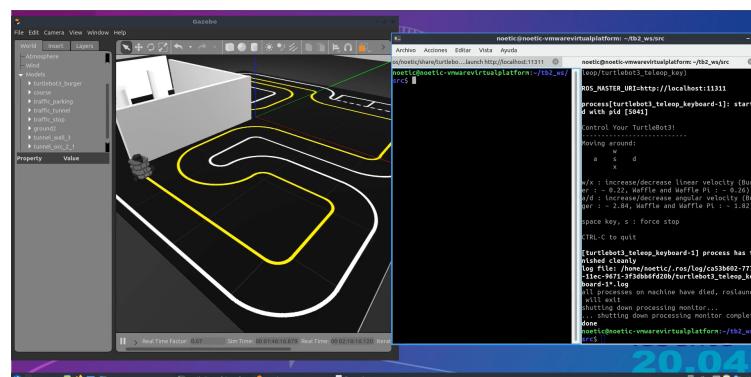


Imagen 26: posiciones imaginarias de las líneas no detectadas

Los valores de las posiciones imaginarias se han establecido teniendo en cuenta las coordenadas de las líneas (puntos rojos) en línea recta y con un error angular muy bajo, lo que se aproximaría a unos valores ideales. La posición imaginaria desaparece en cuanto se vuelve a detectar la línea que reemplaza.

Vídeo proyecto

En el siguiente enlace se encuentra el vídeo completo del funcionamiento del proyecto:



Enlace 4: [video_completo_proyecto.mp4](#)

Pruebas en laboratorio

Para realizar pruebas en el laboratorio, se preparó un código en el que el robot se desplazaba hacia adelante mientras no detectara una señal roja. Como señal roja se empleó una silla de este color. La detección de la señal se realizó de manera similar al empleado en simulación.

```
# Detección de Señales
def detect_signal (self, image, lower, upper, h, w, hsv):

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

    # Zona de búsqueda: margen superior derecho
    mask[0:int(h/2), 0:int(w/2)] = 0
    mask[int(h/2):h, 0:w] = 0

    # Detección del contorno y Cálculo del área
    kernel = cv2.getStructuringElement(cv2.MORPH_RECT, (3,3))
    opening = cv2.morphologyEx(mask, cv2.MORPH_OPEN, kernel, iterations=1)

    contours = cv2.findContours(opening, cv2.RETR_EXTERNAL, cv2.CHAIN_APPROX_SIMPLE)
    contours = contours[0] if len (contours) == 2 else contours[1]

    area = 0
    for pixel in contours:
        area += cv2.contourArea(pixel)

    return area
```

Imagen 27: función de detección de señales

La mayoría de las pruebas realizadas en los robots reales no fueron exitosas, ya que el robot en ocasiones detectaba la señal cuando no la había. Esto se pudo deber a que los rangos de detección del color rojo no fueron los correctos, ya que los rangos que mejores resultados producían en simulación se obtuvieron tras asistir al laboratorio, por lo que no se pudo confirmar que este fuera el motivo. El siguiente vídeo muestra una de las pruebas que se realizaron con éxito.



Enlace 5: [video_lab_proyecto.mp4](#)

Práctica subida a GitHub

La práctica ha sido subida a GitHub en el siguiente enlace: https://github.com/cefq1/P3_Moviles

