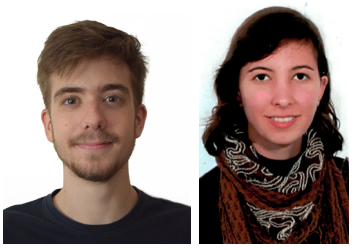
Universidad de Huelva

Grado en Ingeniería Informática

Robótica

Memoria Práctica

Autores:
Alberto Fernández Merchán
Alba Márquez-Rodríguez



Profesores:
Fernando Gómez Bravo
Rafael López de Ahumada Gutiérrez

Asignatura: Robótica

Año: 2022/23

Índice

Índice de figuras

1. Introducción al Sistema de Control y Sensores del EV3

Para las prácticas será utilizado el robot Ev3, el esquema del robot utilizado en las prácticas es el siguiente:

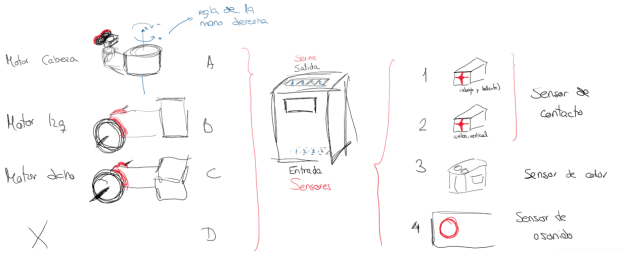


Figura 1: Esquema del robot LEGO utilizado en prácticas

Durante la primera sesión el objetivo fue la familiarización con el Robot y su estructura. Realizando programas sencillos.

El programa realizado fue uno para mover la cabeza del robot a través del motor de la cabeza (A). Posteriormente se añadió controlar esta rotación con un Sensor de Contacto (2).

2. Control de movimiento Simulados de un Manipulador de 6 grados de Libertad

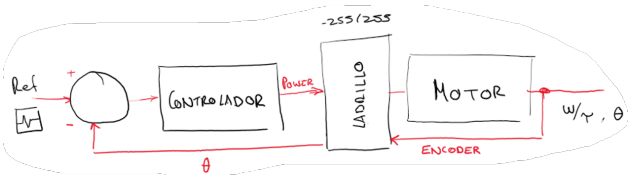


Figura 2: Esquema de un controlador de bucle cerrado

En esta sesión se introdujeron los controladores y el *encoder*.
Se estudiaron 3 tipos de controladores:

- Proporcional (P):
- Proporcional-Integral (PI)
- Proporcional-Integral-Derivativo (PID)

En nuestro caso utilizaremos el proporcional, en el que el error se multiplica por una constante de proporcionalidad ($error_t \cdot kp$). Esta constante puede ser encontrada a través del método de sintonía.

Los resultados fueron los siguientes:

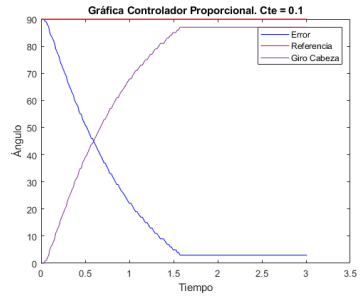


Figura 3: Gráfica del controlador principal con $cte = 0{,}1$

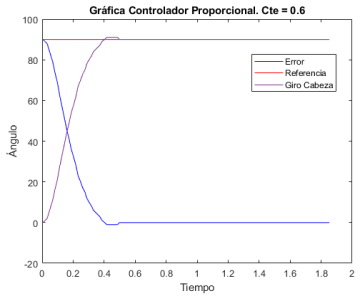


Figura 4: Gráfica del controlador principal con $cte = 0{,}6$

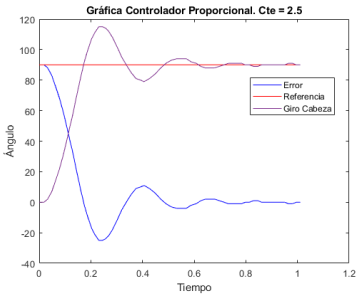


Figura 5: Gráfica del controlador principal con $cte = 2{,}5$

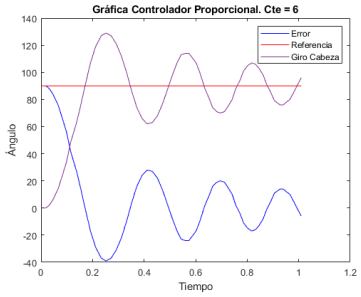


Figura 6: Gráfica del controlador principal con $cte = 6$

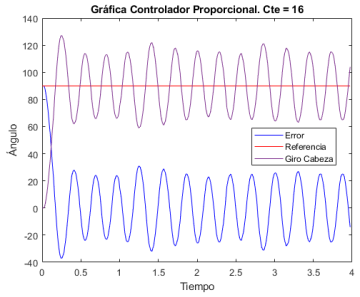


Figura 7: Gráfica del controlador principal con $cte = 16$

Con las gráficas anteriores hemos observado como elegir una buena constante de proporcionalidad es esencial para el correcto funcionamiento del robot. Si elegimos una demasiado baja, no llegará a la referencia, sin embargo, si elegimos una muy alta se produce lo que se conoce como: **sobreoscilación** y se ve reflejado en que la cabeza del robot comienza a oscilar entorno al punto de referencia.

La última parte consistió en enlazar el movimiento de la rueda izquierda con la cabeza. Por lo que si se giraba la rueda izquierda manualmente la cabeza giraría automáticamente.

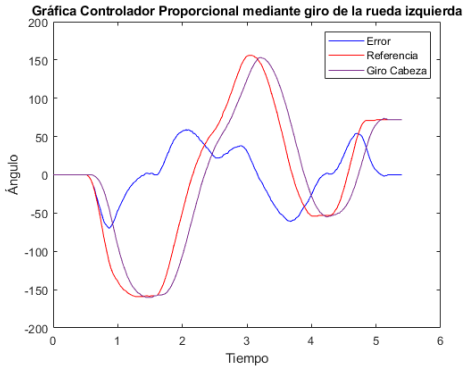


Figura 8: Controlador Proporcional mediante giro de la rueda izquierda.

El movimiento de la cabeza se podía visualizar desde la siguiente gráfica:

Figura 9: Simulación del robot girando la cabeza.

3. Introducción al entorno de ROS

En esta práctica se describen y trabajan los conceptos básicos del sistema operativo ROS, las principales instrucciones básica para ROS, así como la introducción a la programación de los robots móviles en el entorno de ROS.

En esta sesión vimos algunos conceptos como:

- **Nodos**: Son programas que realizan cálculos. ROS es modular, por lo que utiliza un nodo para cada tarea.
- **Maestro**: El maestro es la parte más importante de ROS ya que, sin él, los demás nodos no podrían comunicarse entre ellos.
- **Mensajes**: Es el elemento que utilizan todos los nodos para comunicarse entre ellos.
- **Topics**: Son un tipo de mensaje que utilizan un sistema de transporte basado en la subscripción y publicación.

Para utilizar ROS ejecutaremos la máquina virtual que nos proporcionan en Moodle que tiene instalado el sistema operativo Ubuntu y el software necesario para poder ejecutar las instrucciones de ROS con su versión **Indigo**.

Para comenzar a usar ROS debemos lanzar primero el nodo maestro mediante la instrucción *roscore*.

Para lanzara la ejecución de nuevos nodos utilizaremos la instrucción *rosrun*. En primer lugar iniciaremos el simulador del robot turtlebot con la instrucción:

```
turtlebot@rosindigo:~$ rosrun turtlesim turtlesim_node
```

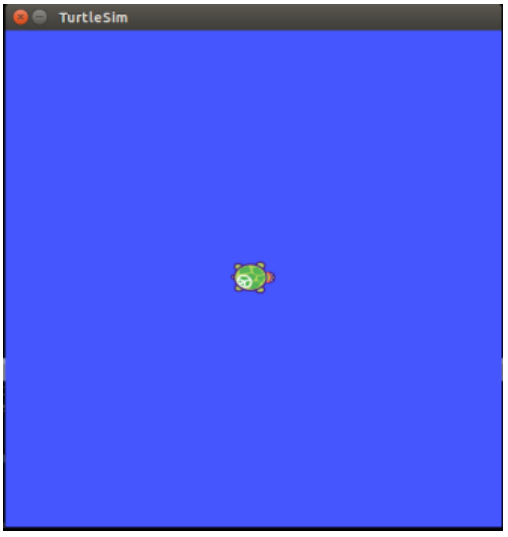


Figura 10: Simulador turtlesim

Podemos ejecutar un nodo llamado *turtle_teleop_key* que permite controlar el robot mediante las flechas del teclado. Lo ejecutamos utilizando:

```
turtlebot@rosindigo:~$ rosrun turtlesim turtle_teleop_key
```

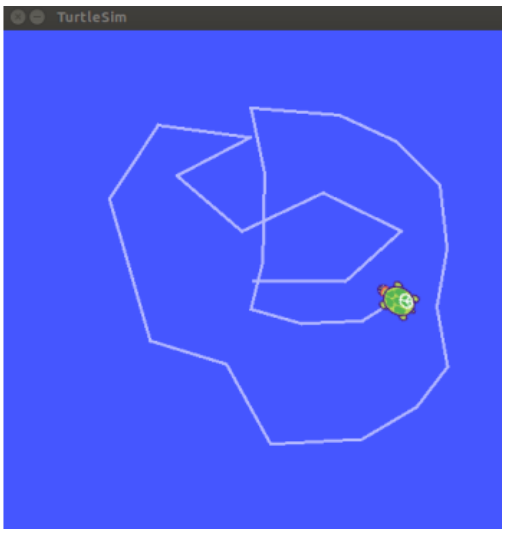


Figura 11: Simulador de turtlesim con el nodo *turtle_teleop_key*

También vimos que podemos consultar la información que publican los *topics* en ROS con el comando:

```
turtlebot@rosindigo:~$ rostopic list
turtlebot@rosindigo:~$ rostopic info /turtle1/pose
turtlebot@rosindigo:~$ rostopic echo /turtle1/pose
```

Con los comandos anteriores podemos ver como cambia la posición de la tortuga en el simulador.

También podemos hacer movimientos predefinidos publicando en el nodo de velocidad del simulador:

```
turtlebot@rosindigo:~$ rostopic pub -r 1 /turtle1/cmd_vel geometry_msgs/Twist '[2.0,0.0,0.0]' '[0.0,0.0,2]'
```

Con la instrucción anterior haríamos una circunferencia de radio 1.

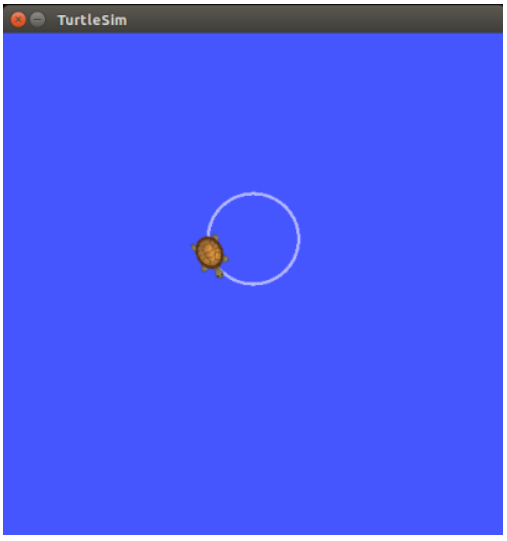


Figura 12: Circunferencia generada en el simulador

Por último visualizamos las variables de estado del robot utilizando *rqt_plot*.

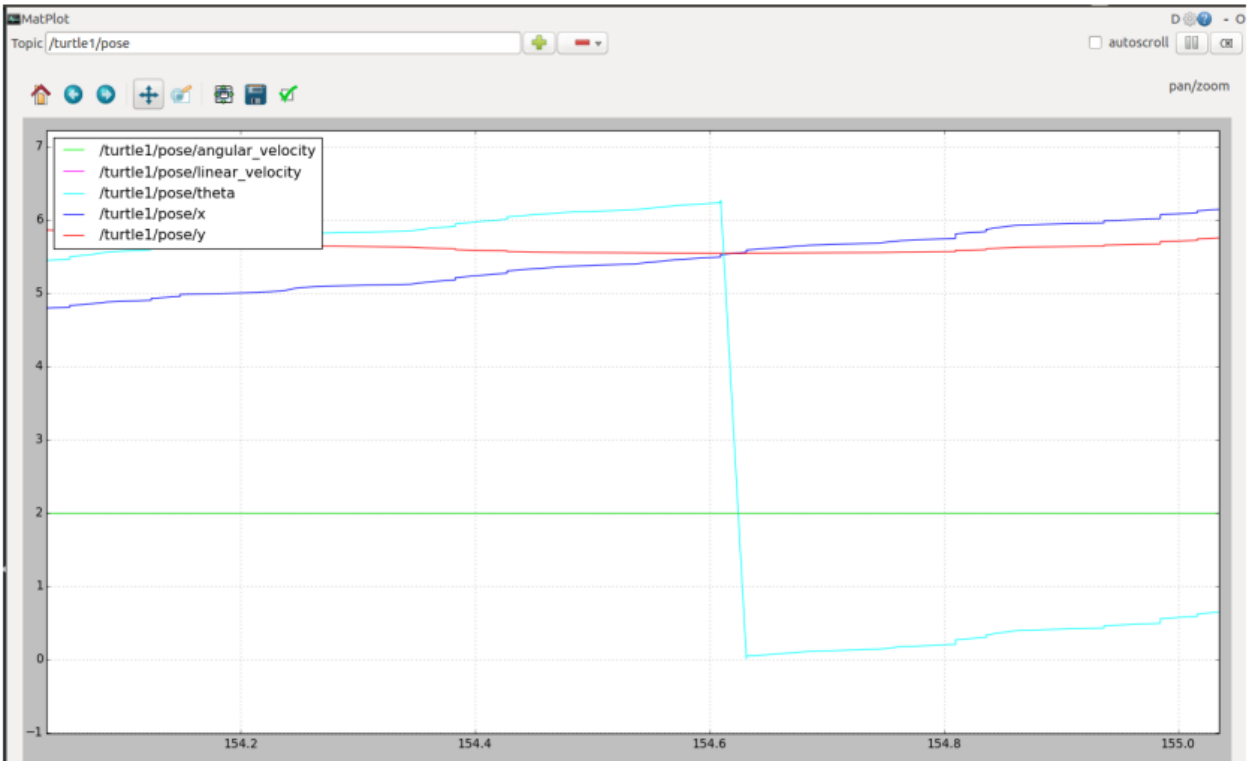


Figura 13: Gráficas de la posición y velocidad del robot

4. ROS: RVIZ y ROSBAG

En esta práctica se utilizará la herramienta **rviz**, un software de visualización en 3 dimensiones diseñada para ROS. Se ejecuta como un nodo del entorno, pero requiere ciertas condiciones para representar correctamente los elementos.

Se puede representar el simulador del robot en rviz. Para ello se utilizan nodos llamados *broadcasters*. Se utilizará el paquete **TF** para mostrar la representación del robot. Seguiremos los siguientes pasos:

1. Lanzar el maestro ROS:

```
turtlebot@rosindigo:~$ rosrun turtlesim turtlesim_node
```

2. Lanzar el nodo de control del turtlebot mediante teclado:

```
turtlebot@rosindigo:~$ rosrun turtlesim turtle_teleop_key
```

3. Crear un nodo broadcaster para traducir los movimientos de la tortufa a un sistema de coordenadas global.

```
turtlebot@rosindigo:~$ rosrun turtle_tf turtle_tf_broadcaster "turtle1"
```

4. Iniciar rviz:

```
turtlebot@rosindigo:~$ rosrun rviz rviz -d `rospack find turtle_tf`/rviz/turtle_rviz.rviz
```

Si desplazamos la tortuga con el teclado veremos como se mueve la representación en rviz.

5. ROS: GAZEBO Y TURTLEBOT

Primero abrimos el mapa original con el que comenzaremos la práctica.

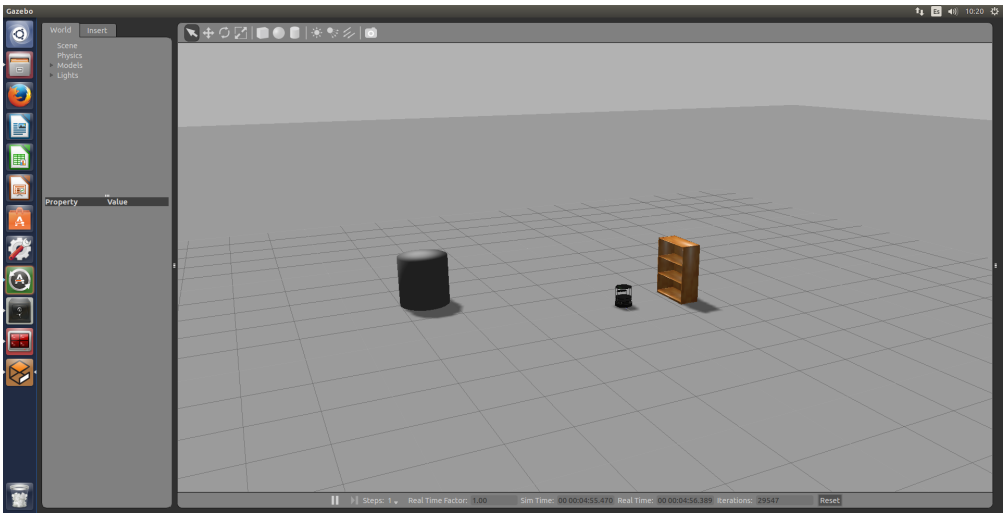


Figura 14: Mapa inicial y original.

En el fichero control_turtlebot spline_v4.py

- Se realizará el envío de datos a topics, visualizando la trayectoria generada, los puntos de paso que estamos generando y el camino que utiliza el algoritmo de control para hacer el seguimiento.

Abriremos la aplicación **rviz** con el comando *rosrun rviz*. En rviz tendremos que realizar los siguientes pasos:

1. Cambiar sistema de referencia a 'odom' para poder visualizar el robot correctamente
2. Añadir un nodo tipo marker para visualizar la trayectoria generada.
3. Añadir un markerarray para generar los puntos rojos que son los puntos de paso definidos para generar la spline.
4. Añadir un nodo de tipo path para poder visualizar el camino que realiza el robot.

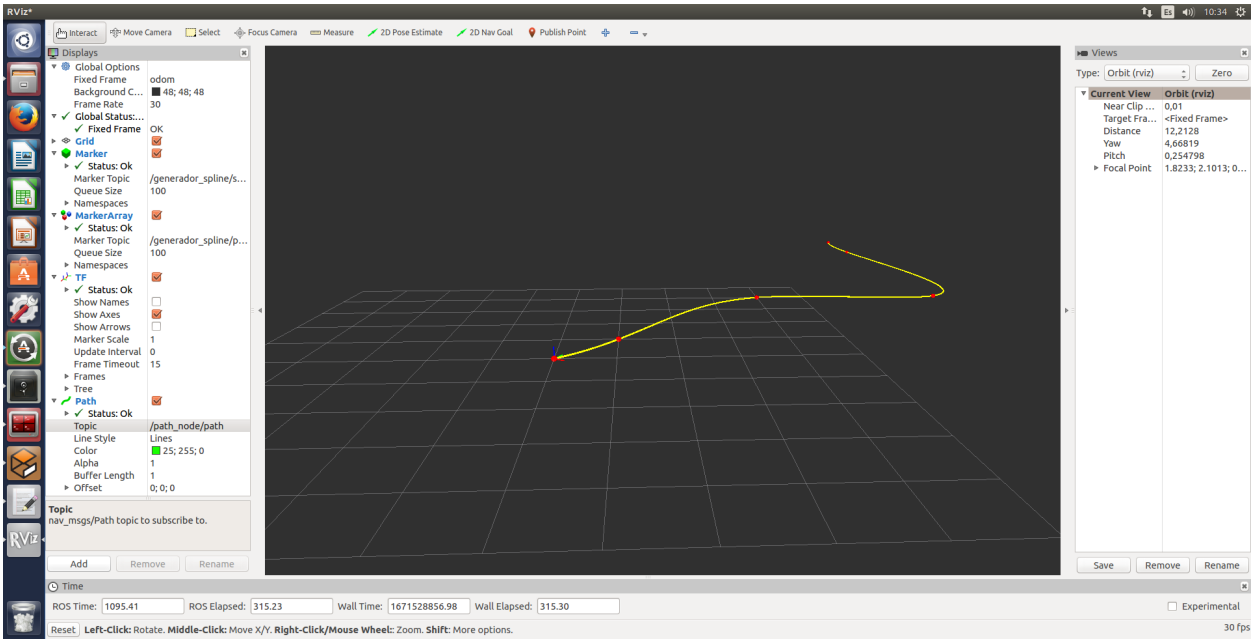


Figura 15: rviz con todos los nodos añadidos

El fichero *control_alumno.py* está parcialmente programado, habrá que modificarlo para que siga la spline generada por nosotros, ya que, en la versión original, genera una circunferencia:

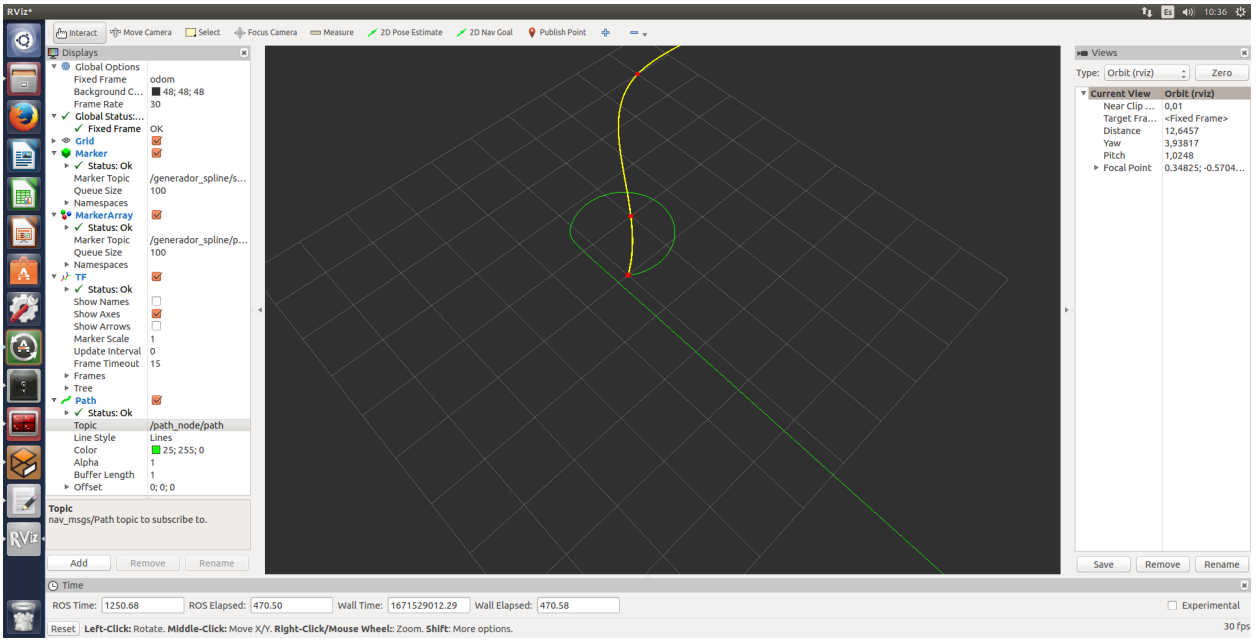


Figura 16: Camino inicial del robot

La línea recta que aparece sucede porque el robot colisiona con un objeto en la simulación sobre la aplicación **gazebo**. Cuando colisiona pierde el control y comienza a andar recto. Para solucionarlo debemos dividir la velocidad lineal entre 3. Esto hará que la circunferencia sea mayor y no colisione con el objeto.

Para no tener que escribir siempre los mismos comandos en la consola, existen ficheros llamados *ficheros launch*. Estos ficheros permiten lanzar varios ficheros sin necesidad de escribirlos uno a uno.

- Crear fichero: control_alumnos.launch:
  - Abrir gazebo
  - Nodo add path
  - Nodo spline
  - Nodo rviz (abrir rviz)

6. ROS: Maestro Remoto y TURTLEBOT Real

En esta práctica trabajamos con los robots físicos que había en el laboratorio.

En la primera sesión nos tocó al robot llamado **Getrud** y tuvimos que modificar el fichero del sistema operativo */etc/hosts* para incluir el nombre de nuestro robot junto con su IP: 192.168.2.62.

Para acceder a los servicios del robot, tendremos que ejecutar los siguientes comandos:

```
turtlebot@rosindigo:~$ export ROS_MASTER_URI=http://Gertrud:11311
turtlebot@rosindigo:~$ export ROS_HOSTNAME=192.168.2.28
```

Utilizaremos la herramienta **Filezilla** para acceder al contenido del robot y poder descargar y cargar ficheros mediante una conexión sftp. Utilizaremos una conexión ssh para acceder a la consola del robot:

```
turtlebot@rosindigo:~$ ssh turtlebot@Gertrud
```

Para poder controlar el robot mediante un joystick tendremos que ejecutar el siguiente nodo que se proporciona en moodle.

```
rosrun control_turtlebot control_joy.py
```

Debemos ejecutar el comando lanzador:

```
roslaunch turtlebot_bringup minimal.launch
```

Finalmente, utilizando la función realizada en prácticas anteriores que genera un camino de splines, intentamos hacer que el robot siguiese un camino formado por una línea recta. Sin embargo, debido a problemas técnicos no conseguimos que el robot se comunicase correctamente con el programa.

7. Actividades

7.1. Actividad 3. Simulación de Modelos Cinemáticos

En esta actividad hemos resuelto el problema del modelo inverso. Hemos utilizado MatLab para programar un simulador en el que podemos simular tanto un robot diferencial como un triciclo.

Nos centraremos en el robot de triciclo, ya que tiene un control más simple. En esta actividad jugaremos con la velocidad para poder realizar una circunferencia con el triciclo:

```
volante=-0.1416;
velocidad=2;
conduccion=[velocidad volante];
```

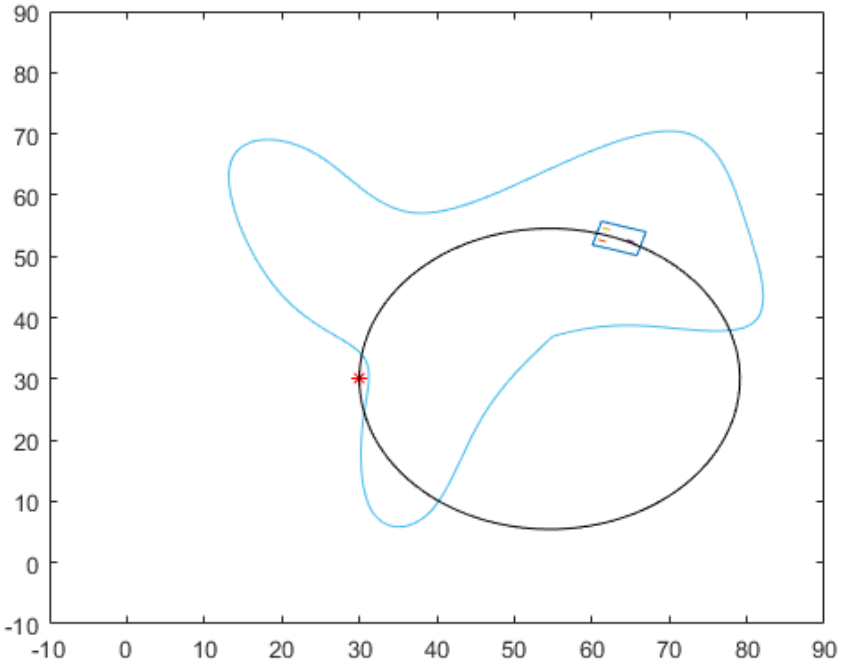


Figura 17: Circunferencia hecha con kuta_triciclo

7.2. Actividad 4. Simulación de Control Geométrico

En esta práctica se diseñará un algoritmo que permitirá al vehículo diferencial situarse en un punto concreto. Para ello se utilizarán las siguientes fórmulas:

$$\Delta = (x_0 - x_f)\sin(\theta_0) - (y_0 - y_f)\cos(\theta_0)$$
$$L_H = \sqrt{(x_0 - x_f)^2 + (y_0 - y_f)^2}$$
$$\rho = \frac{2\Delta}{L_H^2}$$

En el código podemos verlo:

```
delta = (pose(1,k) - punto(1))*sin(pose(3,k))-(pose(2,k) - punto(2))*cos(pose(3,k));
LH = sqrt( (pose(1,k) - punto(1))^2 + (pose(2,k) - punto(2))^2 );
rho=(2*delta)/(LH^2);
velocidad_lineal = 10;
```

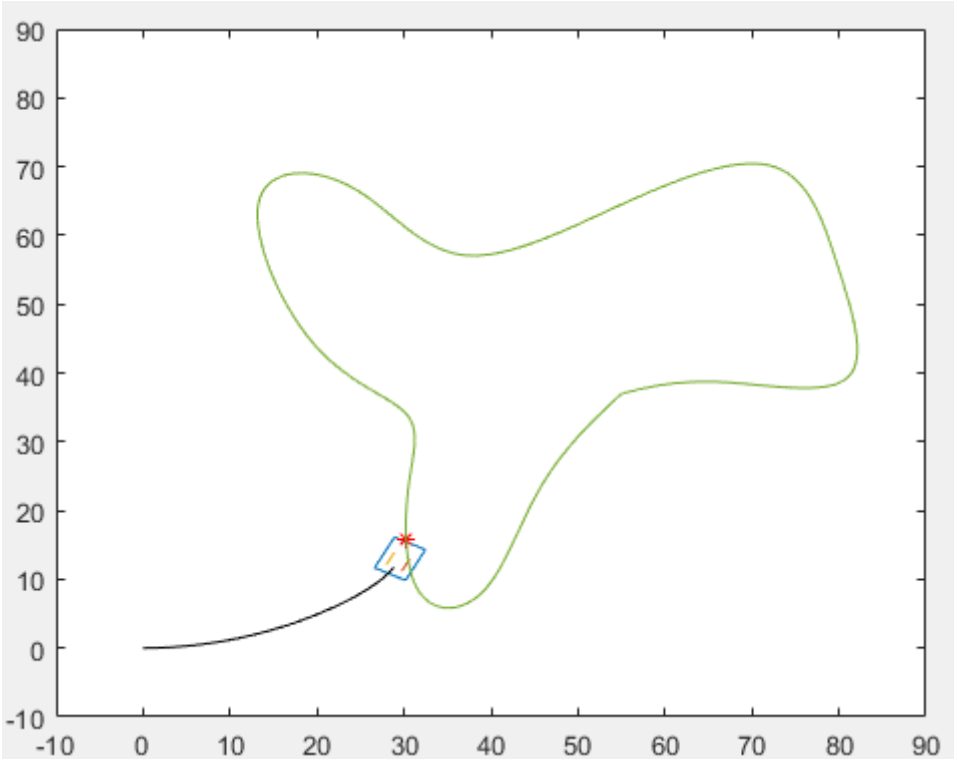


Figura 18: Vehículo diferencial aproximándose al punto objetivo

Podemos hacer que disminuya la velocidad con forme se vaya acercando al punto. De esta forma la transición entre el movimiento y la para será más suave:

```
kv = 1;
velocidad_lineal = kv*LH;
```

donde kv es una constante de proporción.

7.3. Actividad 5. Introducción al Path Following

En esta actividad se implementa una función en la que se devuelve el punto más cercano al robot y, dentro del bucle de control del programa principal, el robot irá aproximándose al siguiente punto más cercano.

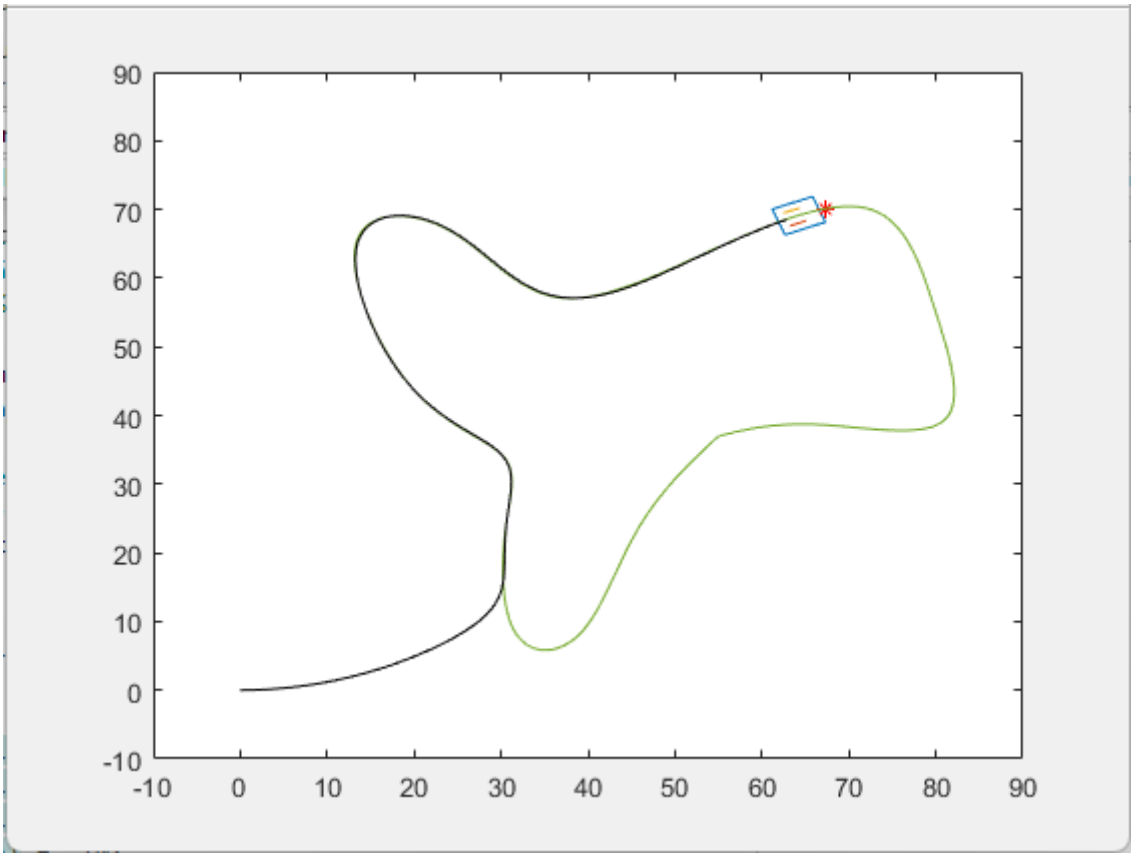


Figura 19: Robot diferencial siguiendo el punto más cercano

7.4. Actividad 6. Definición de Caminos con Splines

En esta actividad, el simulador tratará que el vehículo diferencial siga un camino **cinemáticamente admisible**, es decir, que cumpla las siguientes características:

- Que sea continuo.
- Que la tangente debe coincidir con la dirección de la velocidad.
- Que sea libre de colisiones.
- Los caminos deben ser suaves (*smooth*). Esta característica está vinculada con dos conceptos matemáticos:

  1. **Derivabilidad**: La curva del camino debe ser derivable en todos sus puntos.
  2. **Segunda Derivada**: La segunda derivada debe ser continua en todos sus puntos.

  Por tanto, podemos concluir que cuanto más derivadas continuas tenga una función, esta será más suave.

Existen muchas formas de representar curvas. Nosotros usaremos las **splines cúbicas**, es decir, representaremos las curvas como polinomios de grado 3.

$$X = Ax + Bxt + Cxt^2 + Dxt^3$$
$$Y = Ay + Byt + Cyt^2 + Dyt^3$$

Donde la t es un parámetro que se puede interpretar como un *tiempo virtual*.
Podemos plantear 8 ecuaciones con 8 incógnitas y encontrar los distintos coeficientes que definen un camino. Asignando sub-metas al camino final, obtenemos tantos tramos de *splines* como parejas de puntos tenemos.

En **Matlab** usaremos la función *funcion_spline_cubica_varios_puntos* para generar el camino a partir de las distancias intermedias.

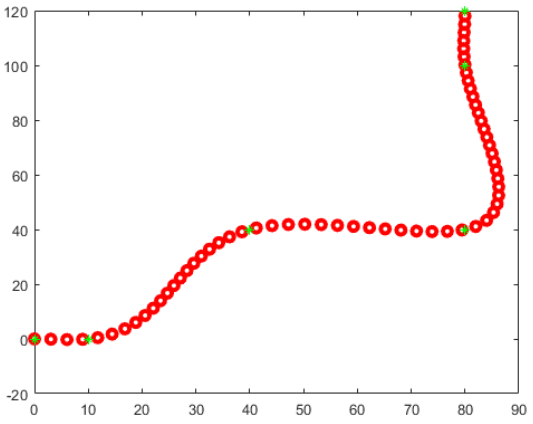


Figura 20: Ejemplo de camino hecho por splines

Podemos adaptar el camino de la actividad anterior al que hemos generado con la función:

```
% En lugar de utilizar el camino del fichero, generaremos el camino con la
% función funcion_spline
xc=[0 10 40 80 80 80];
yc=[0 0 40 40 100 120];
ds=3; %distancia entre puntos en cm.
camino = funcion_spline_cubica_varios_puntos(xc,yc,ds)';
```

Figura 21: Fragmento de código para cambiar el camino

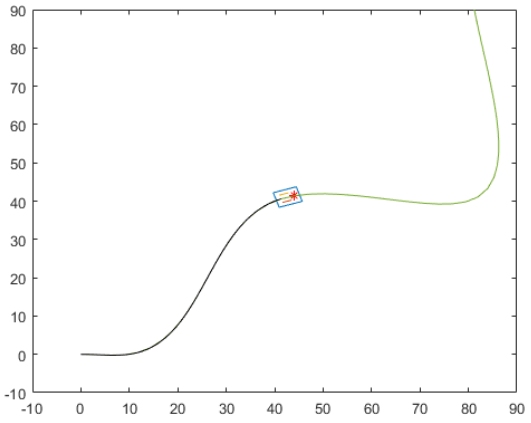


Figura 22: Vehículo derivativo siguiendo el camino generado anteriormente

Matlab da fallo al terminar de recorrer el camino porque el robot intenta seguir al punto siguiente al final del camino que, obviamente, no existe. Para solucionar este problema tendremos que asignar el punto final del camino como punto de seguimiento cuando el punto que vaya a seguir el vehículo sea superior a la longitud del camino.

```
orden_minimo = minima_distancia(camino,[pose(1,k),pose(2,k)]);
look_ahead = 3;

if(orden_minimo+look_ahead >= length(camino))
    punto = camino(end,:);
else
    punto = camino(orden_minimo+look_ahead,:);
end
```

Figura 23: Fragmento de código que soluciona el error

Para hacer que la curva sea tangente a la configuración de salida y llegada hay que tener en cuenta el ángulo que el vehículo tiene en ambas configuraciones. Para ello, se deben fijar dos puntos adicionales. Uno de ellos Pd (punto de despegue), se fijará posterior a la configuración inicial, separado del mismo una distancia arbitraria dd (distancia de despegue). El otro, Pa (punto de aterrizaje) será definido anterior al correspondiente a la configuración final, separado del mismo una distancia arbitraria da (distancia de aterrizaje). El cálculo de los mismos será de la forma: *(Ejercicio 2)*

$$Pdx = X_0 + dd * \cos(\theta_0)$$
$$Pdy = Y_0 + dd * \sin(\theta_0)$$
$$Pax = X_f - da * \cos(\theta_f)$$
$$Pay = Y_f - da * \sin(\theta_f)$$

El código de Matlab sería el siguiente:

```
%%%%%%%%%%%%%%%%%%%%%%%% Ejercicio 2 %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
configuracion_inicial = [10,15,-pi/4]; % [X0,Y0,theta0]
configuracion_final = [80,80,(2/3)*pi]; % [Xf, Yf, tehtaf]
dd = 5; % Distancia de despegue
da = 5; % Distancia de aterrizaje

% Posición de Despegue
Pdx = configuracion_inicial(1) + dd*cos(configuracion_inicial(3));
Pdy = configuracion_inicial(2) + dd*sin(configuracion_inicial(3));

% Posición de aterrizaje
Pax = configuracion_final(1) - da*cos(configuracion_final(3));
Pay = configuracion_final(2) - da*sin(configuracion_final(3));
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

Figura 24: Fragmento de código para obtener los puntos de despegue y aterrizaje.

7.5. Actividad 7: Planificando rutas con A*

En esta actividad utilizaremos el algoritmo A* para trazar el camino que deberá seguir nuestro robot en un mapa con obstáculos.

Si ponemos el tamaño de la celda demasiado pequeño, el robot podría pegarse demasiado a los obstáculos y el algoritmo tardaría demasiado en ejecutarse. Sin embargo, si ponemos el tamaño demasiado grande, el robot no considerará algunos caminos debido a que las celdas están parcialmente ocupadas.

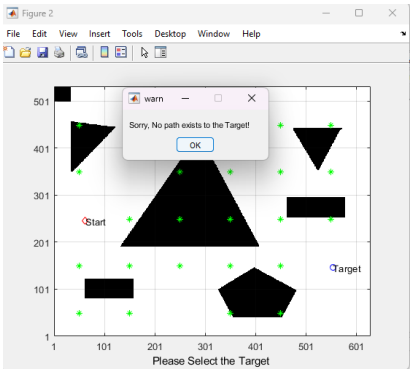


Figura 25: Cuadrícula del A* demasiado grande

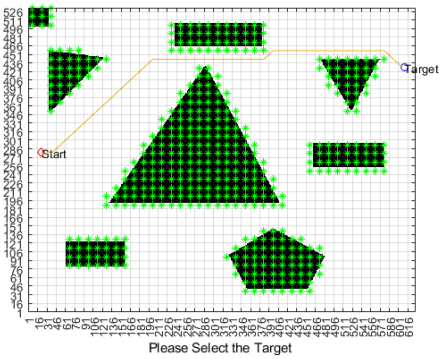


Figura 26: Cuadrícula del A* de 15

Los caminos que genera el algoritmo A* no son caminos admisibles. Tendremos, por tanto, que aplicar la función de generar el camino mediante splines. El resultado sería:

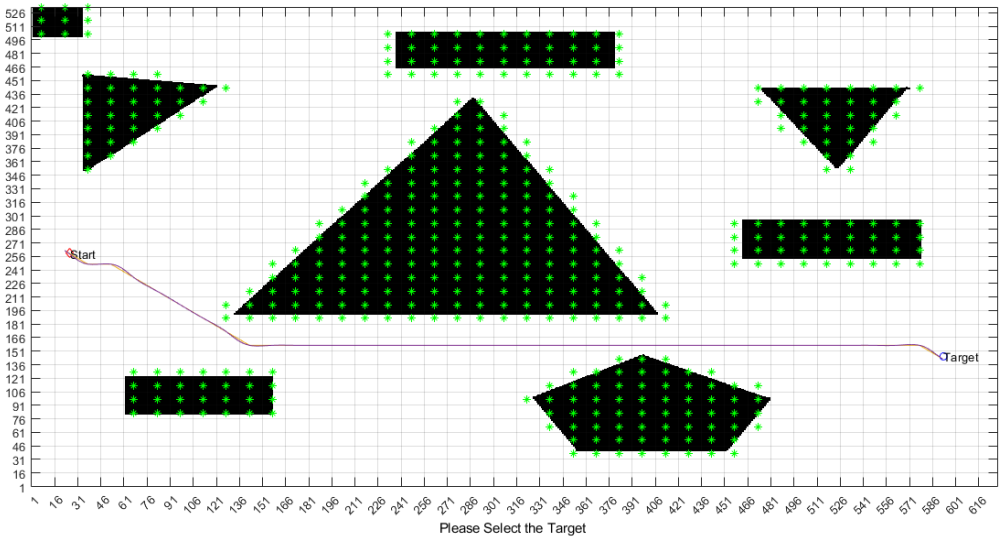


Figura 27: Camino A* con splines

A continuación tendremos que dibujar un mapa de nuestra casa para pasárselo al algoritmo. También tendremos que integrar el robot diferencial con el algoritmo A* para que trace el camino seleccionado.

7.5.1. Mapa de la casa

Para esta actividad se ha diseñado un mapa que representa mi casa:

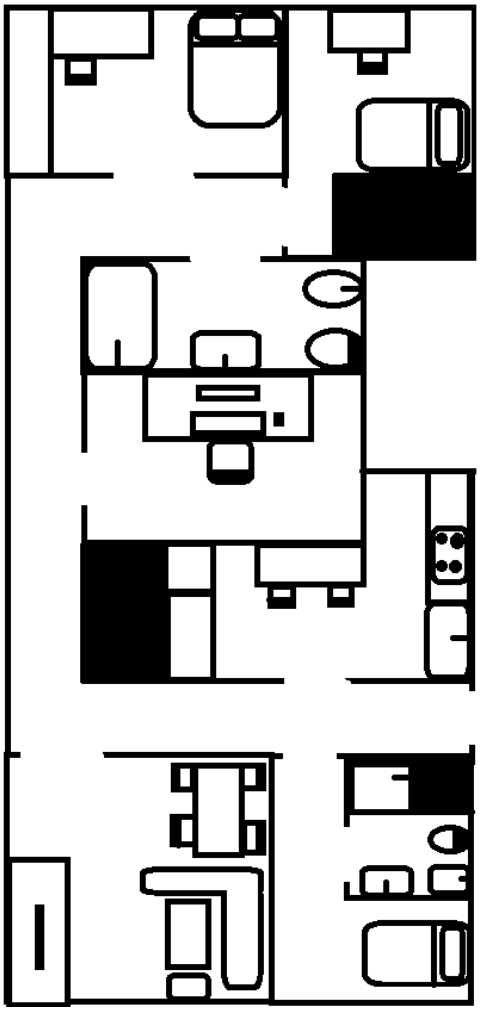


Figura 28: Representación de una casa

7.5.2. Integración del A* con el robot

El punto de origen se introduce a mano y el punto de destino se selecciona con la función de matlab. A partir de la primera iteración ya se podrá seleccionar el punto destino cada vez que queramos y el camino se irá acumulando:

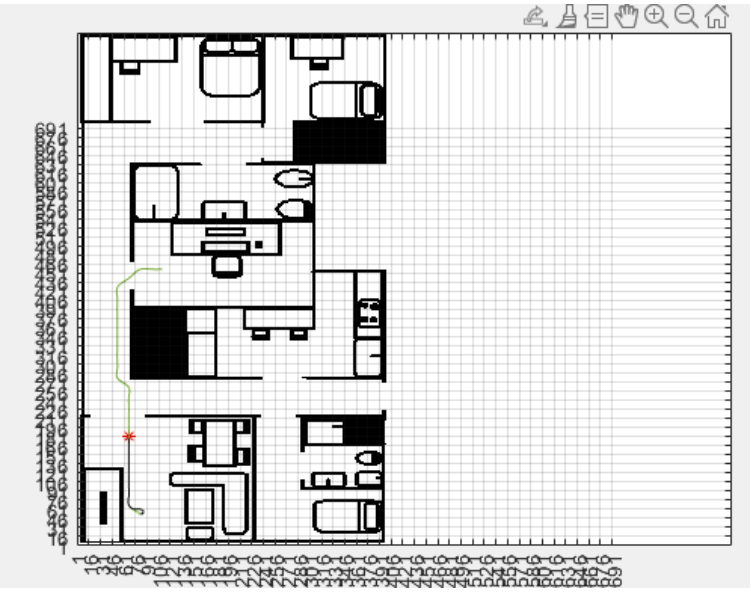


Figura 29: Primera iteración del algoritmo de seguimiento

El algoritmo se repite hasta que el usuario cierre el programa.

Figura 30: Opción de volver a elegir el camino