

# PRÁCTICA FINAL

## SISTEMAS DE CONTROL INTELIGENTE

Diseño de Controladores Borrosos y Neuronales  
para la maniobra de aparcamiento de un vehículo

Grado en Ingeniería Informática

David Barreiro Zambrano

Natalia Montoya Gómez

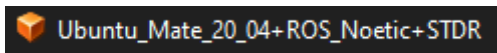
## *ÍNDICE*

<i>Instalación y configuración del entorno .....</i>	<i>3</i>
Desarrollo de la práctica.....	8
PARTE I. Diseño manual de un control borroso de tipo MAMDANI .....	8
PARTE II. Diseño automático de un controlador neuronal .....	16
Conclusiones .....	21

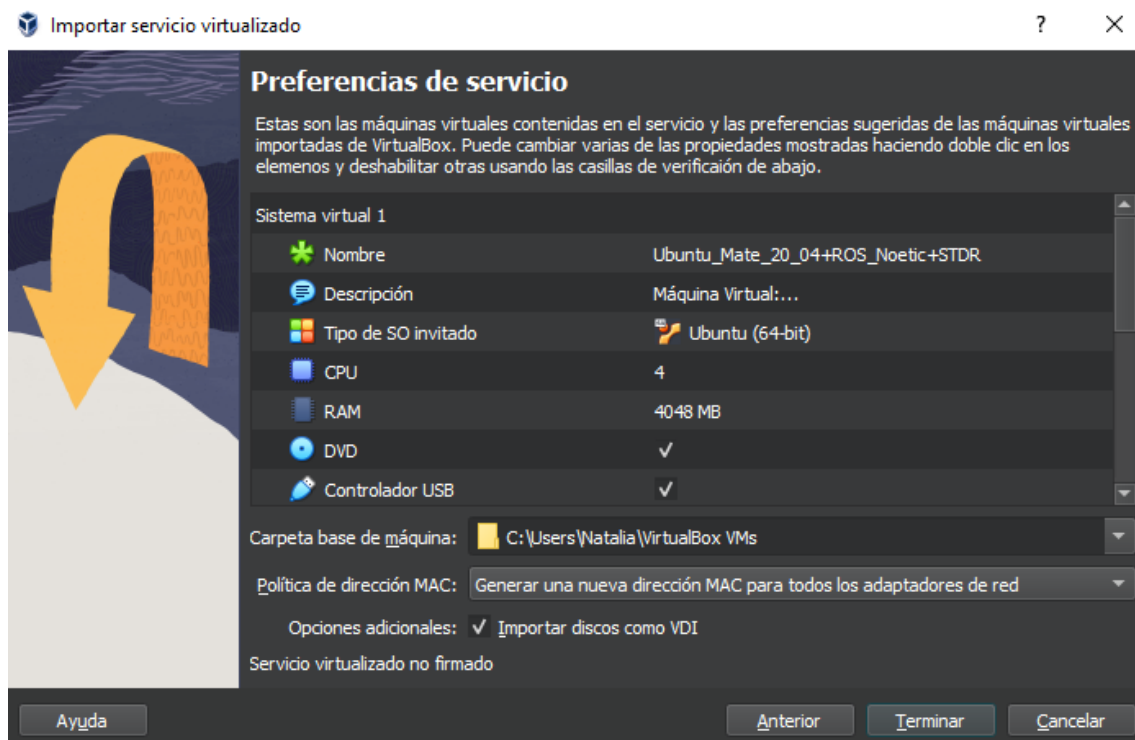
## Instalación y configuración del entorno

A continuación, se muestra el procedimiento que se ha tomado para poner a punto la máquina, en cuanto a instalación y configuración de la misma, para un correcto funcionamiento del entorno.

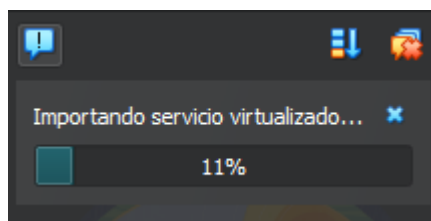
En primer lugar, se descarga el fichero .ova, ubicado en el espacio del aula virtual de la asignatura. Esta máquina viene preparada por los profesores de la asignatura con los programas necesarios para el desarrollo de esta práctica final:



Una vez descargado el fichero que contiene la imagen de la máquina virtual, desde VirtualBox, se importa de tal manera que solo es necesario determinar la política de dirección de MAC, la cual es: *Generar una nueva dirección MAC para todos los adaptadores de red*, tal y como aparece en la siguiente figura:

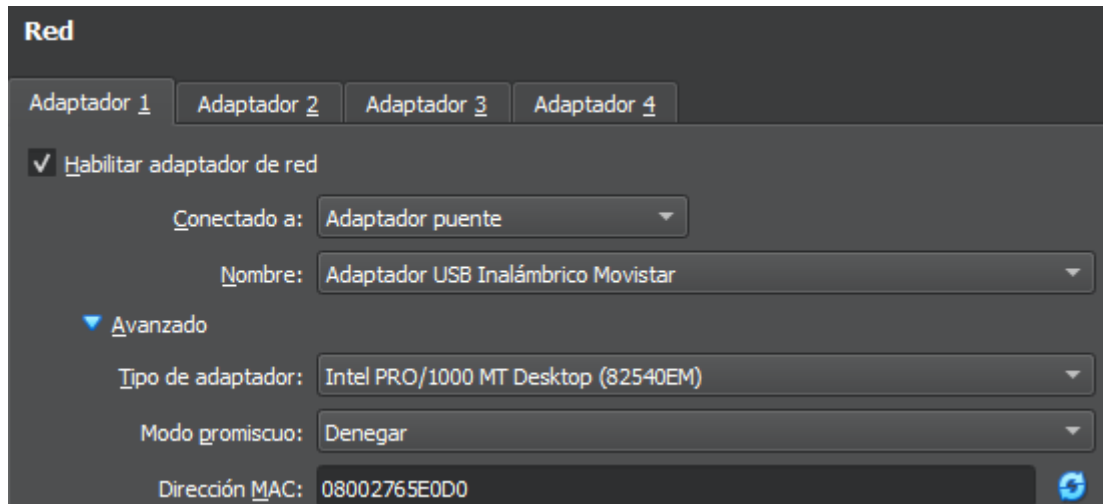


Presionamos el botón de *Terminar* y la máquina empieza a importarse a nuestro equipo:



Una vez instalada la máquina, el nombre de usuario y contraseña es *alumno*. Con lo cual, podremos entrar a la máquina, pero antes de ello, es necesario configurar algunos detalles más.

En la ventana de *VirtualBox*, se clicca en *Configuración* y se cambia, en caso de que tuviese otra configuración, a la que aparece en la figura siguiente, *Adaptador puente*, lo cual permite que haya una conexión con el host principal del equipo, Internet y otras máquinas virtuales. Con esto se consigue que la información viaje desde la máquina virtual hacia el host y viceversa, para ejecutar *Matlab* mientras en la máquina se ejecuta *ROS*.



Aprovechamos que estamos con esta ventana abierta, para generar distintas direcciones MAC por si hubiese problemas con alguna conexión. Por lo que, clicamos varias veces sobre las flechas azules que aparecen al lado de la dirección MAC.

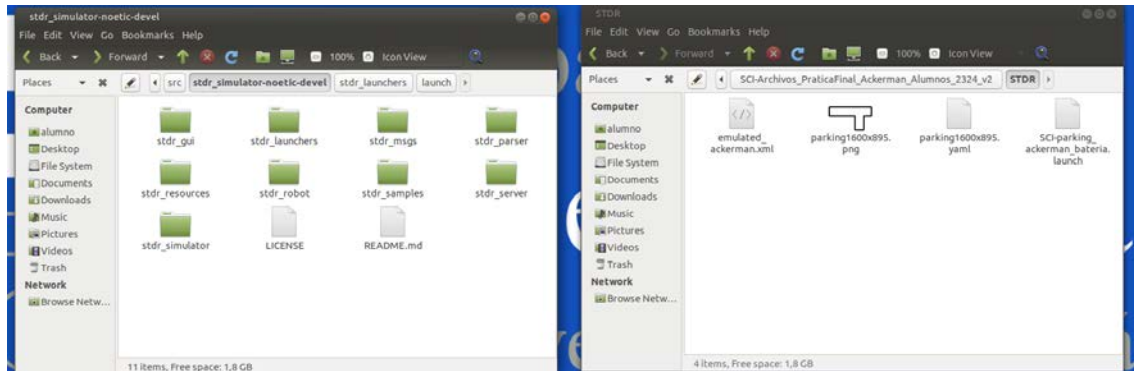
Estando dentro de la máquina, se abre una terminal, desde donde lanzaremos los siguientes comandos y podremos acceder a ciertas carpetas:

```
alumno@alumno-MV:~$ mkdir -p ~/robotica_movil_ws/src
alumno@alumno-MV:~$ cd ~/robotica_movil_ws/src/
alumno@alumno-MV:~/robotica_movil_ws/src$ catkin_init_workspace
File "/home/alumno/robotica_movil_ws/src/CMakeLists.txt" already exists
alumno@alumno-MV:~/robotica_movil_ws/src$ cd ~/robotica_movil_ws
alumno@alumno-MV:~/robotica_movil_ws$ catkin_make
Base path: /home/alumno/robotica_movil_ws
Source space: /home/alumno/robotica_movil_ws/src
Build space: /home/alumno/robotica_movil_ws/build
Devel space: /home/alumno/robotica_movil_ws/devel
Install space: /home/alumno/robotica_movil_ws/install
####
#### Running command: "make cmake_check_build_system" in "/home/alumno/robotica_movil_ws/build"
####
####
#### Running command: "make -j3 -l3" in "/home/alumno/robotica_movil_ws/build"
####
[ 0%] Built target _std_msgs_generate_messages_check_deps_SpawnRobotFeedback
```

Añadimos las siguientes líneas de código al fichero `.bashrc`:

```
source ~/robotica_movil_ws/devel/setup.bash
export ROS_PACKAGE_PATH=$ROS_PACKAGE_PATH:~/robotica_movil_ws/
```

Como últimos pasos, y para tener el entorno preparado, se mueven los archivos descargados desde el aula virtual en la carpeta de descargables de la asignatura, donde se almacenan los archivos necesarios para STDR, entre los que se encontrarán el mapa, el fichero `.launch` y el `.xml`.



El fichero `.launch`, el cual contiene la posición donde estará posicionado el robot en el mapa, está ubicado en la carpeta `stdr_launchers/launch`. En la cual, para cambiar al entorno definitivo de la práctica, se necesitan cambiar los parámetros que se señalan en amarillo en la figura siguiente:

```
SCI-parking_ackerman_bateria.launch ✕
<launch>

  <include file="$(find stdr_robot)/launch/robot_manager.launch" />

  <node type="stdr_server_node" pkg="stdr_server" name="stdr_server"
output="screen" args="$(find stdr_resources)/maps/parking1600x895.yaml"/>

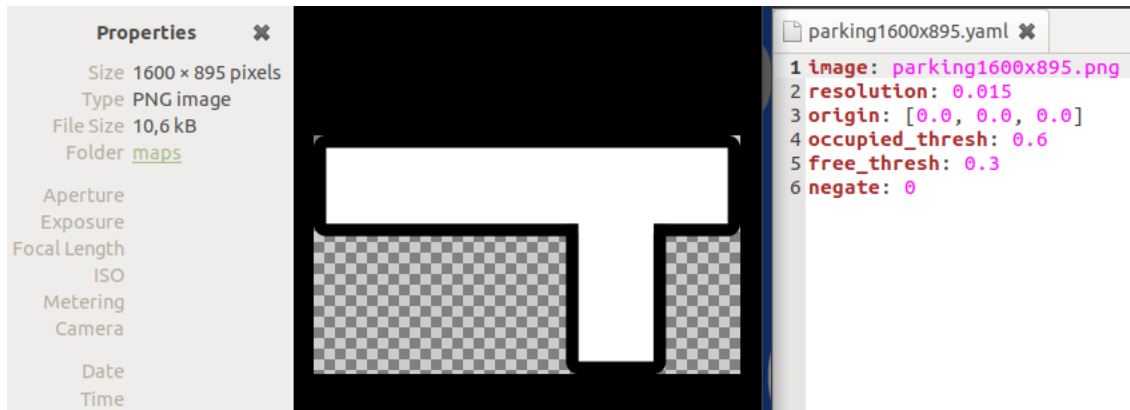
  <node pkg="tf" type="static_transform_publisher" name="world2map"
args="0 0 0 0 0 0 world map 100" />

  <include file="$(find stdr_gui)/launch/stdr_gui.launch"/>

  <node pkg="stdr_robot" type="robot_handler" name="$(anon
robot_spawn)" args="add $(find stdr_resources)/resources/robots/-
emulated_ackerman.xml 5 10.5 3.14159" />

</launch>
```

Los ficheros que configuran el mapa en ROS, son el *.png* y el *.yaml* mostrados en la siguiente figura, los cuales están ubicados en la carpeta de *stdr\_resources/maps*.



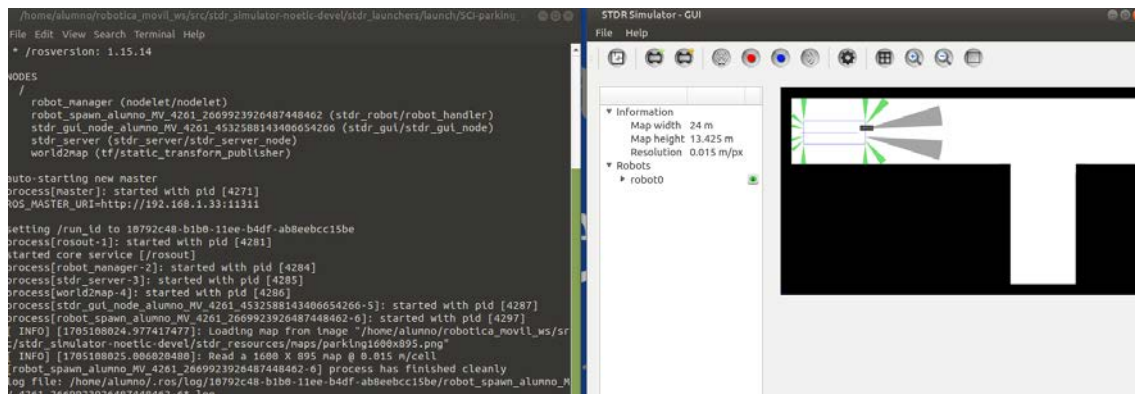
El fichero que contiene las especificaciones del robot, tales como las posiciones de los sónares, y otras características, están en el fichero *.xml*, ubicado en la carpeta *stdr\_resources/resources/robots*.

```
<robot>
  <robot_specifications>
    <initial_pose>
      <x>0</x>
      <y>0</y>
      <theta>0</theta>
    </initial_pose>
    <footprint>
      <footprint_specifications>
        <radius>0.3</radius>
        <points>
          <point>
            <x>-0.5</x>
            <y>-0.75</y>
          </point>
          <point>
            <x>3.5</x>
            <y>-0.75</y>
          </point>
          <point>
            <x>3.5</x>
            <y>0.75</y>
          </point>
          <point>
            <x>-0.5</x>
            <y>0.75</y>
          </point>
        </points>
      </footprint_specifications>
    </footprint>
  </robot_specifications>
</robot>
```

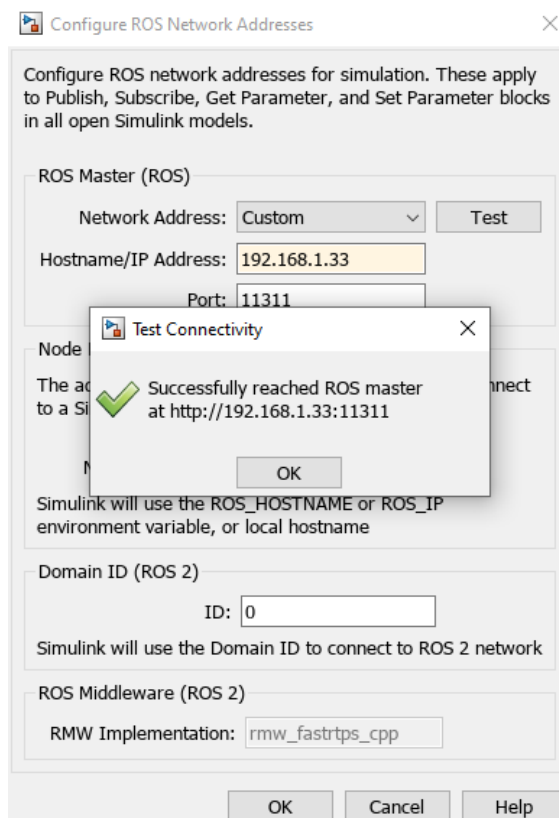
Una vez ubicados todos los ficheros en sus respectivas carpetas, se lanza la siguiente instrucción para poder abrir el programa, con el robot ubicado en la posición en la que ha sido especificado, y el mapa correspondiente.

```
alumno@alumno-MV: ~
File Edit View Search Terminal Help
alumno@alumno-MV:~$ roslaunch stdr_launchers SCI-parking_ackerman_bateria.launch
```

Al ejecutar la instrucción anterior, la interfaz del simulador de STDR con ROS se abre, y desde la terminal, podemos observar que se encuentra en espera de algún mandato para poder empezar a mover el robot.



Por último, y con Matlab abierto en nuestro host, podemos hacer una comprobación de que máquina y host están conectados, dando al botón de *Test* con el cual se abrirá una ventana emergente en la que comunicará si ha habido éxito en la conexión o no.



Es probable que el *Firewall* esté activado, por lo que deberá dar permisos a su versión de Matlab en redes públicas y privadas.

Con este último paso, termina la instalación y configuración del sistema.

A continuación, se describe el desarrollo de la práctica.

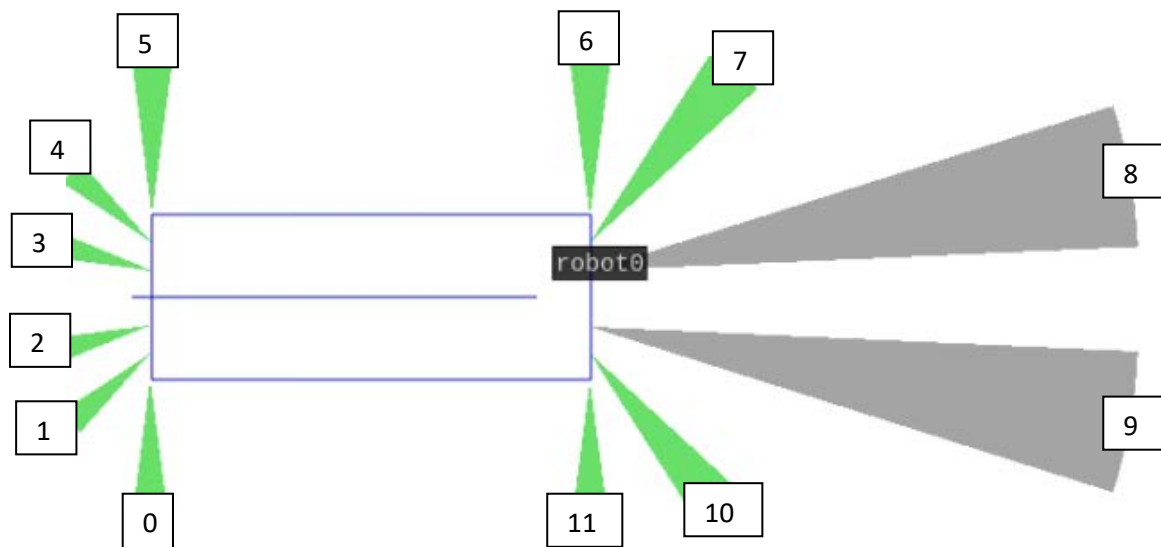
## Desarrollo de la práctica

### PARTE I. Diseño manual de un control borroso de tipo MAMDANI

Se comienza con el diseño del controlador borroso, para ello, abrimos la consola de Matlab y ejecutamos la instrucción *fuzzy* tal y como se especifica en el enunciado de la práctica. Con esto, podemos crear un controlador borroso básico inicial. Este comando permite abrir la ventana emergente de Fuzzy Logic Design, desde donde se controlan las entradas y salidas, funciones de pertenencia y reglas de un sistema de control borroso.

En nuestro caso, a modo de idea para englobar el problema que se plantea, hemos tomado la ayuda del Control Manual que se proporcionaba en los ficheros de ayuda descargables. De esta manera, hemos ido haciéndonos una idea de los sensores que eran o no necesarios para este modelo de problema.

En la siguiente figura se visualiza el robot con todos los sensores numerados:



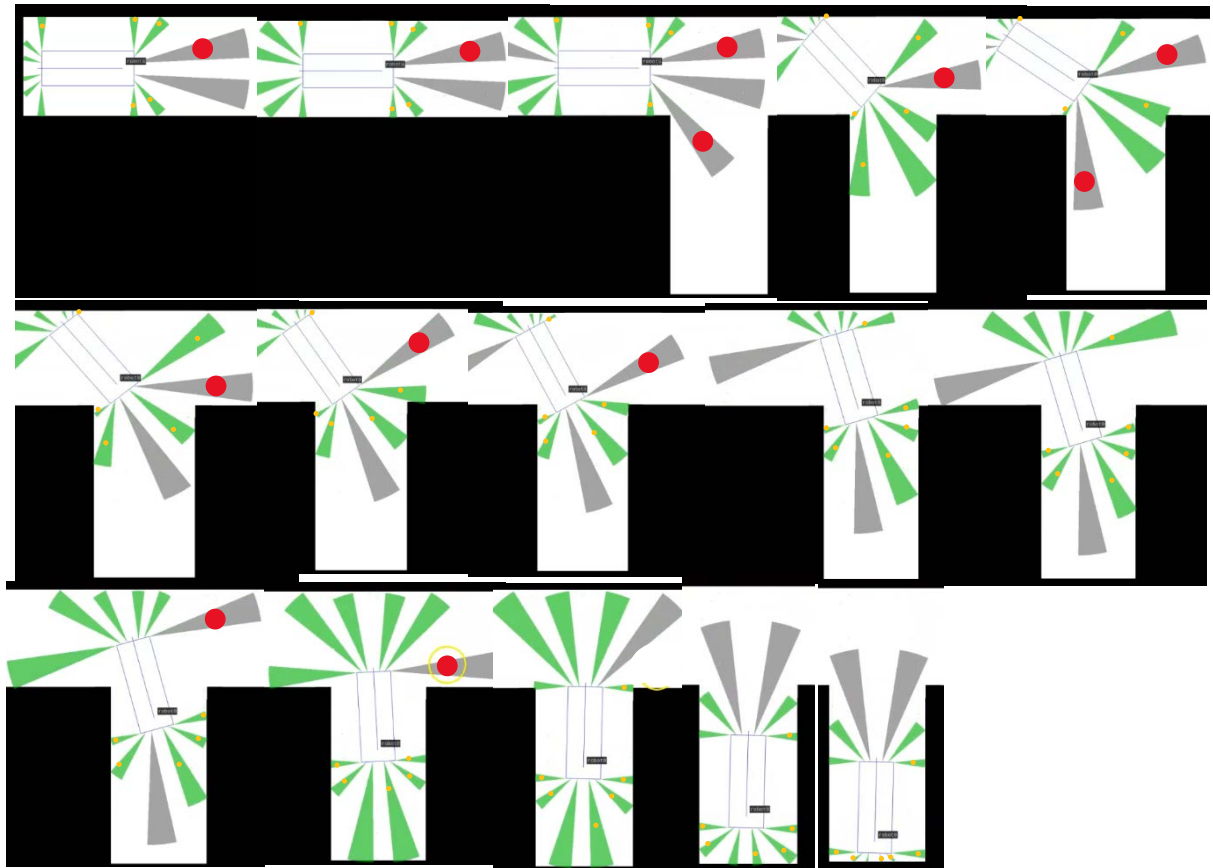
Tras varias maniobras de aparcamiento con el controlador manual 'ControlManual\_Ackerman\_12\_sensores\_V\_steering' hemos llegado a la conclusión que solo es necesario usar 6 sensores para que la maniobra solicitada se ejecute de forma correcta.

Notamos que cuando ciertos sensores tienen como entrada su máxima distancia (cercano a 5 o superior) nos sirven para referenciar el giro del robot, junto con los que tienen entradas más pequeñas, para el aparcamiento.

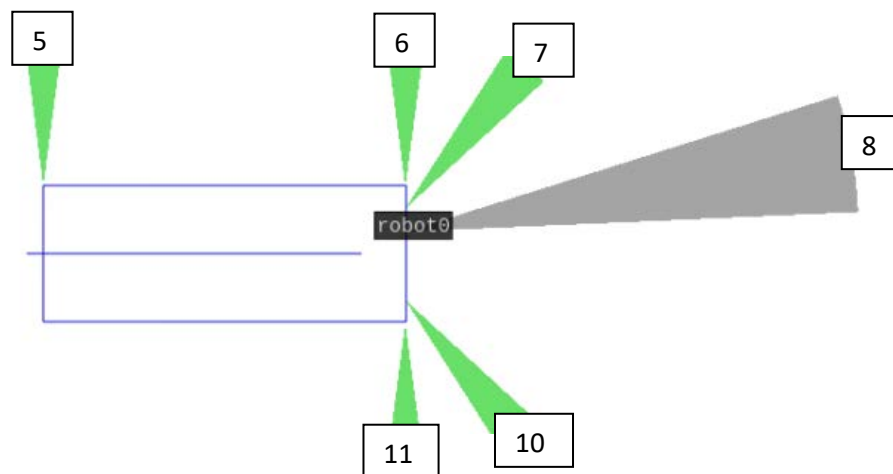
Así como para saber si el robot debe seguir moviéndose sin hacer ningún giro. Por ejemplo cuando empieza a moverse y se va acercando a la zona de aparcamiento o bien cuando ya está colocado dentro de la misma con la orientación correcta.

A continuación, demostraremos gráficamente el estudio sobre el que se ha basado el análisis de la maniobra y la decisión del uso de los sensores, tomando los mínimos e imprescindibles que aporten información. Con puntos amarillos denotamos los sensores que tienen una distancia menor a 5 y con puntos rojos los sensores que tienen una distancia superior o próxima a 5.



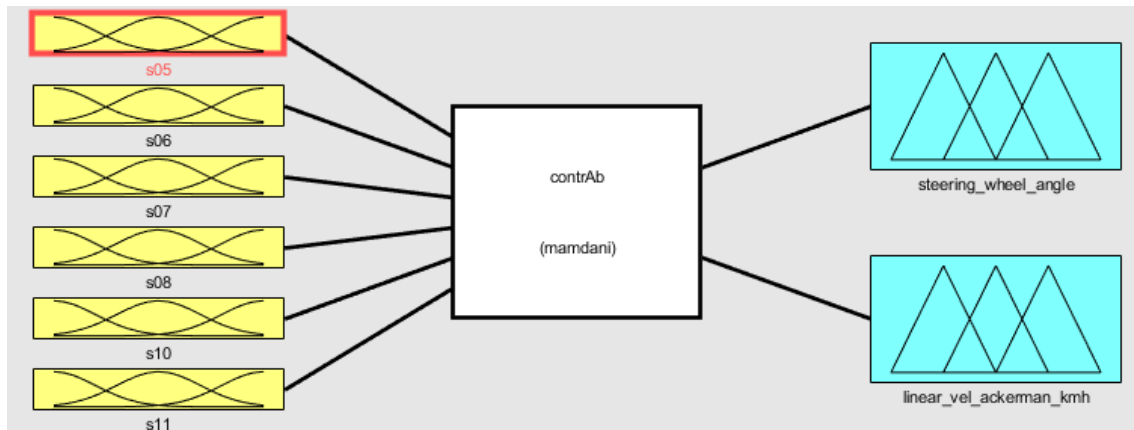


De los mostrados en la imagen, como hemos comentado, solo usamos 6 de ellos, ya que son los que entendemos necesarios para la maniobra. Estos son el 5, 6, 7, 8, 10 y 11:



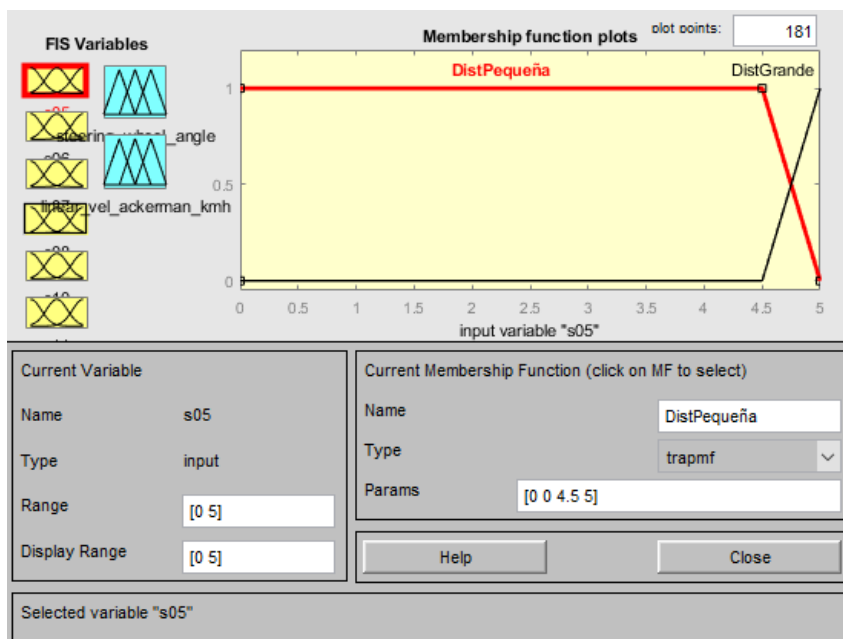
El robot está orientado hacia atrás, por lo que el sensor 8 y 10 son los sensores que tiene en la parte trasera. De esta forma, la plaza de aparcamiento se encuentra a su izquierda, con lo que hemos considerado que solo serán necesarios los sensores que se mostraban en la imagen anterior.

Estos sónares se ven reflejados como entradas en el sistema de control borroso que se ha creado para resolver este problema, los cuales se pueden visualizar en la siguiente figura. El sistema tiene como salidas, la velocidad y el ángulo.



Se puede observar que las funciones de pertenencia de los sónares son las mismas para todos ellos, puesto que hemos considerado dos estados, la distancia grande y la distancia pequeña. Cuando se trata de una distancia grande, se entiende que el sónar no ve ningún obstáculo en una distancia cercana a 5m. Sin embargo, se considera que una distancia es pequeña cuando empieza a ser menor de 4.5m.

Esta figura es aplicable para todos los sónares, pues tienen los mismos parámetros:



Cuando se trata de las funciones del ángulo de giro del volante y de la velocidad del robot, las funciones de pertenencia serán distintas.

En cuanto a la velocidad, tenemos 5 funciones.

DriveG: El robot va hacia delante con velocidad superior a 2.5.

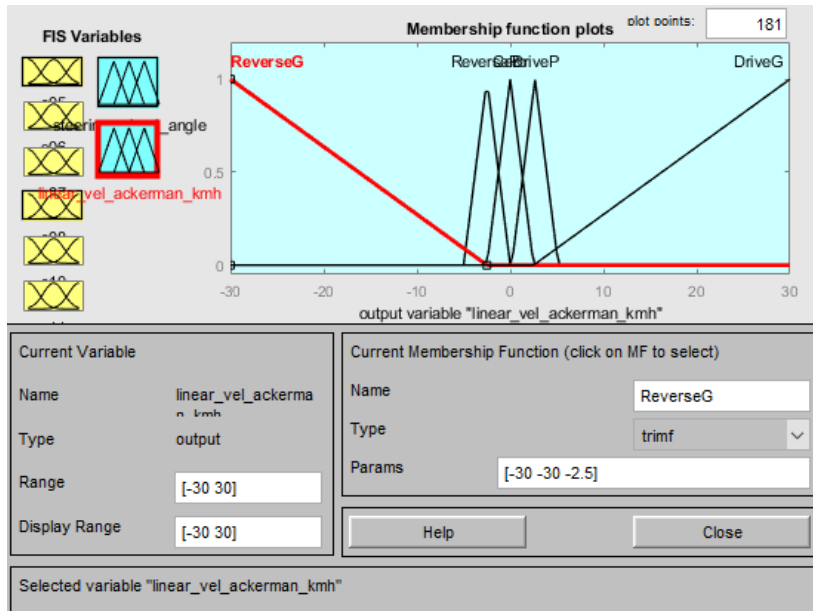
DriveP: El robot va hacia delante con una velocidad entre 0 y 5.

Cero: El robot toma una velocidad entre -2.5 y 2.5.

ReverseP: El robot va marcha atrás con velocidad entre 0 y -5.

ReverseG: El robot va marcha atrás con velocidad superior a -2.5.

Pero lo más predominante será el uso de la función de reversa, ya que nuestro robot solo maniobra marcha atrás durante toda la maniobra, pues según como está posicionado, la forma más rápida de realizar el aparcamiento es de esta manera y girando el ángulo perfecto como para no tener que moverse hacia delante durante dicha maniobra.



Además, hemos visto que lo más eficiente es que lleve una velocidad pequeña hacia atrás (ReverseP) para que pueda reconocer y reaccionar a tiempo a las entradas de los sensores.

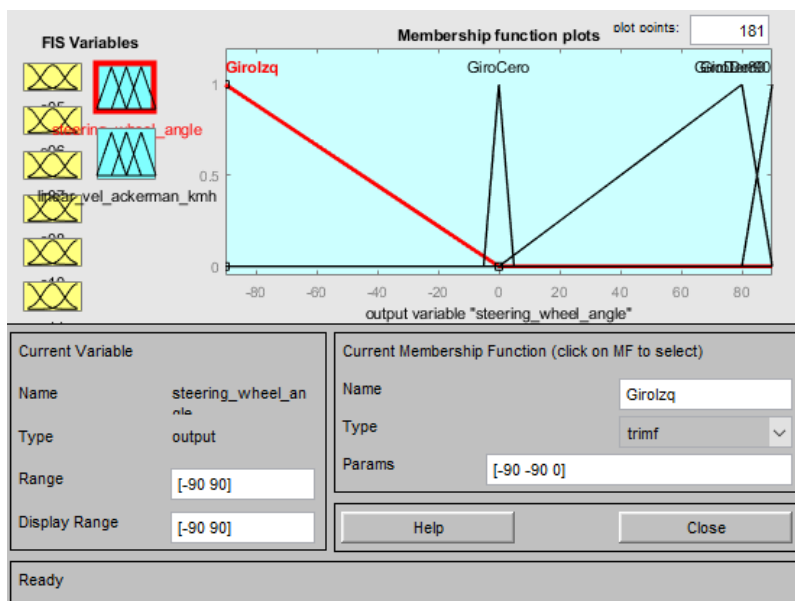
En cuanto al ángulo, tenemos 4 funciones:

GiroIzq: El robot hace un giro entre superior a 0 hasta -90.

GiroCero: El robot hace un giro entre -5 y 5.

GiroDer80: El robot hace un giro entre 0 y 90 con su valor máximo en 80.

GiroDer90: El robot hace un giro entre 80 y 90.



La función más usada es la de giro de 90 grados, pues en el momento en el que el s010 tiene una distancia grande, debería hacer un giro radical de 90 grados mientras se va moviendo. Exceptuando, cuando el sonar 5 y el 8 tienen una distancia grande que es el momento de corregir el robot con un ángulo de 80 grados para que el robot se coloque correctamente en la plaza de aparcamiento.

El giro Cero es utilizado cuando el robot no tiene que girar, simplemente avanzar en una dirección recta. Es decir, cuando inicia la maniobra o cuando el robot está ubicado en la zona de aparcamiento con la orientación adecuada.

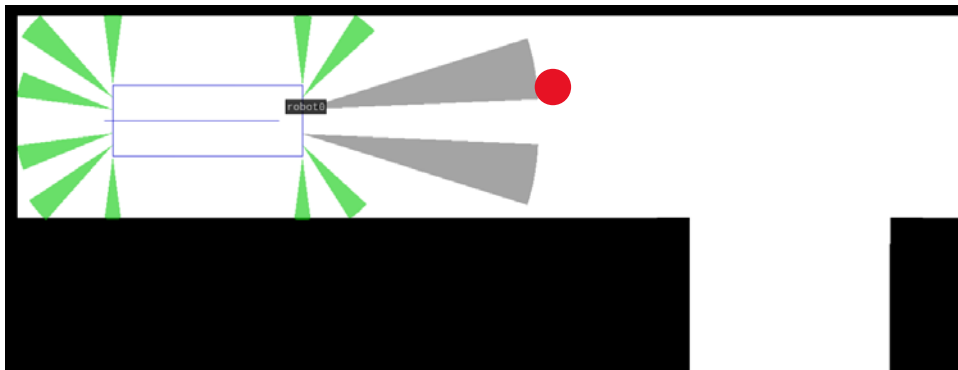
Una vez definidos el rango, los parámetros de las variables de entradas y salidas, y habiendo creado las funciones de pertenencia, se diseñan las reglas del controlador borroso. Para establecer las reglas, solo hemos definido las reglas inferidas del análisis inicial.

```

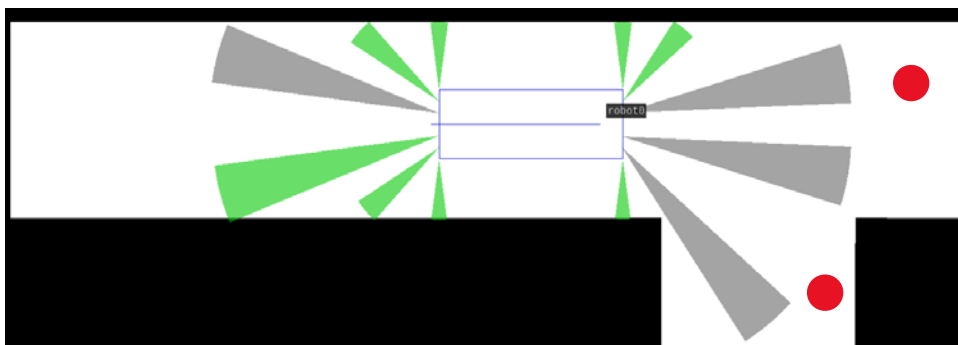
1. If (s05 is DistPequeña) and (s06 is DistPequeña) and (s07 is DistPequeña) and (s08 is DistGrande) and (s10 is DistPequeña) and (s11 is DistPequeña) then (steering_wheel_angle is GiroCero)(linear_vel_ackerman_kmh is ReverseP) (1)
2. If (s05 is DistPequeña) and (s06 is DistPequeña) and (s07 is DistPequeña) and (s08 is DistGrande) and (s10 is DistGrande) and (s11 is DistPequeña) then (steering_wheel_angle is GiroDer90)(linear_vel_ackerman_kmh is ReverseP) (1)
3. If (s05 is DistPequeña) and (s06 is DistGrande) and (s07 is DistGrande) and (s08 is DistPequeña) and (s10 is DistGrande) and (s11 is DistPequeña) then (steering_wheel_angle is GiroDer90)(linear_vel_ackerman_kmh is ReverseP) (1)
4. If (s05 is DistPequeña) and (s06 is DistGrande) and (s07 is DistPequeña) and (s08 is DistPequeña) and (s10 is DistPequeña) and (s11 is DistPequeña) then (steering_wheel_angle is GiroCero)(linear_vel_ackerman_kmh is ReverseP) (1)
5. If (s05 is DistPequeña) and (s06 is DistPequeña) and (s07 is DistPequeña) and (s08 is DistPequeña) and (s10 is DistPequeña) and (s11 is DistPequeña) then (steering_wheel_angle is GiroCero)(linear_vel_ackerman_kmh is ReverseP) (1)
6. If (s05 is DistPequeña) and (s06 is DistPequeña) and (s07 is DistGrande) and (s08 is DistPequeña) and (s10 is DistPequeña) and (s11 is DistGrande) then (steering_wheel_angle is GiroDer90)(linear_vel_ackerman_kmh is ReverseP) (1)
7. If (s05 is DistPequeña) and (s06 is DistPequeña) and (s07 is DistPequeña) and (s08 is DistPequeña) and (s10 is DistPequeña) and (s11 is DistGrande) then (steering_wheel_angle is GiroDer90)(linear_vel_ackerman_kmh is ReverseP) (1)
8. If (s05 is DistPequeña) and (s06 is DistPequeña) and (s07 is DistPequeña) and (s08 is DistGrande) and (s10 is DistPequeña) and (s11 is DistPequeña) then (steering_wheel_angle is GiroDer90)(linear_vel_ackerman_kmh is ReverseP) (1)
9. If (s05 is DistGrande) and (s06 is DistPequeña) and (s07 is DistPequeña) and (s08 is DistPequeña) and (s10 is DistPequeña) and (s11 is DistPequeña) then (steering_wheel_angle is GiroDer90)(linear_vel_ackerman_kmh is ReverseP) (1)
10. If (s05 is DistPequeña) and (s06 is DistPequeña) and (s07 is DistPequeña) and (s08 is DistGrande) and (s10 is DistPequeña) and (s11 is DistPequeña) then (steering_wheel_angle is GiroCero)(linear_vel_ackerman_kmh is ReverseP) (1)
11. If (s05 is DistGrande) and (s06 is DistPequeña) and (s07 is DistPequeña) and (s08 is DistGrande) and (s10 is DistPequeña) and (s11 is DistPequeña) then (steering_wheel_angle is GiroDer80)(linear_vel_ackerman_kmh is ReverseP) (1)
12. If (s05 is DistPequeña) and (s06 is DistGrande) and (s07 is DistPequeña) and (s08 is DistPequeña) and (s10 is DistGrande) and (s11 is DistPequeña) then (steering_wheel_angle is GiroDer90)(linear_vel_ackerman_kmh is ReverseP) (1)

```

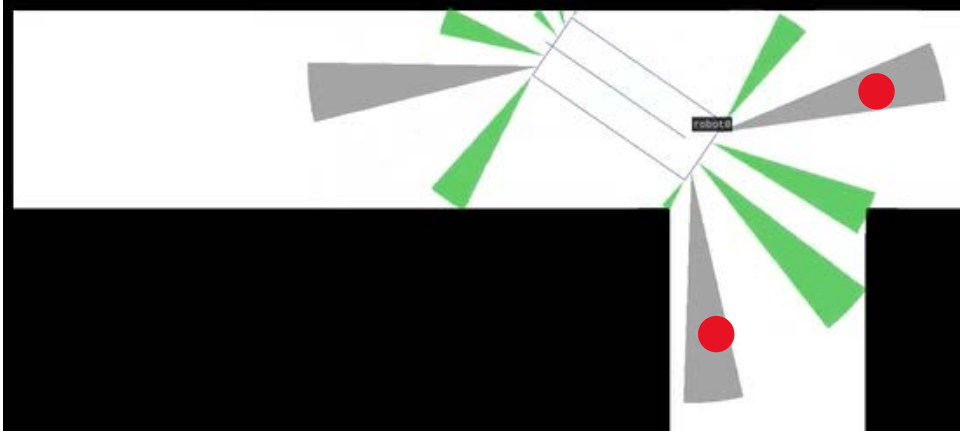
1ª REGLA: Todos los s010ares tienen distancia pequeña, exceptuando el s010ar 8 que tiene una distancia grande, significa que el robot debe ir avanzando en reversa sin realizar ningún giro.



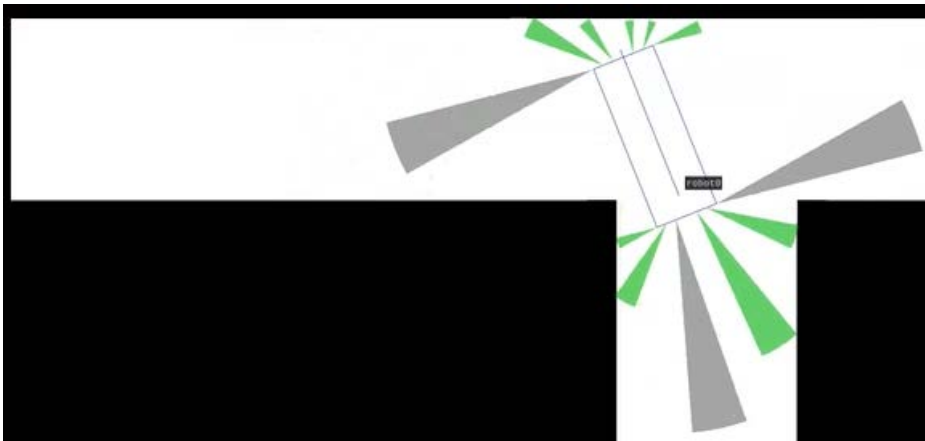
2ª REGLA: Todos los s010ares tienen distancia pequeña exceptuando los s010ares 8 y 10, quiere decir que el robot debería empezar a girar, con un ángulo de 90, pues la plaza de aparcamiento está próxima.



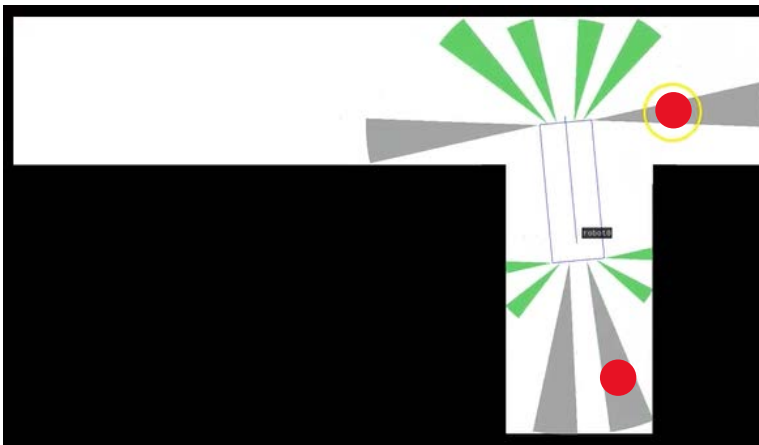
3ª REGLA: Todas las distancias son pequeñas, excepto 7 y 10, significa que tiene posibilidad de giro aún, debe seguir girando mientras sigue avanzando.



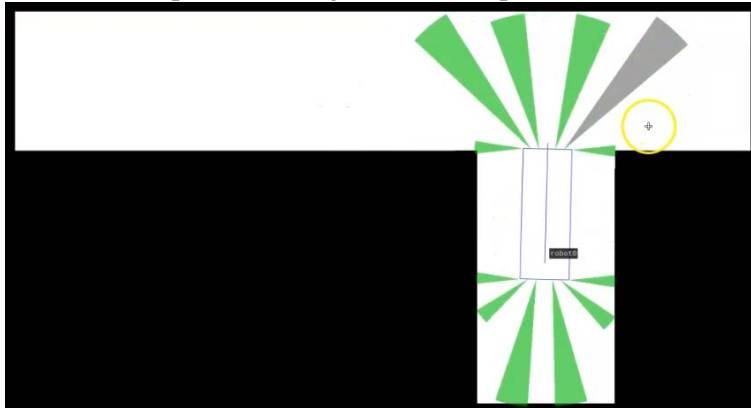
4ª REGLA: Debe seguir girando un poco más, todos los sónares tienen distancia pequeña excepto el 7, lo que significa que falta muy poco espacio de giro.



A continuación, se agruparán las siguientes reglas: 6ª, 7ª, 8ª, 9ª, 10ª, 12ª, puesto que el robot en estas reglas necesita seguir girando. Hasta el momento en que el sensor 5 tiene una distancia grande junto al sensor 8 (regla 11ª), es el momento de aplicar un giro más suave cercano a 80.

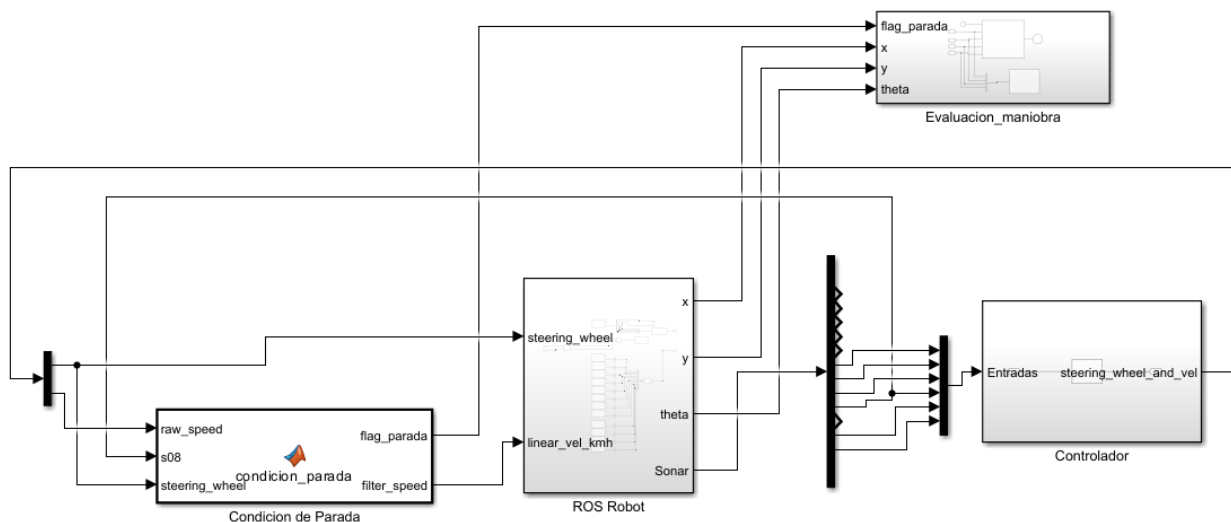


Finalmente, el robot vuelve a tener los 6 sensores con una distancia pequeña, por ello es momento de aplicar la 5ª regla. Es decir, aplicar un GiroCero, muy pequeño o cercano a cero.



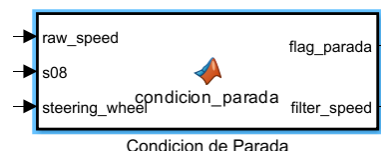
El robot seguirá avanzando hasta que el flag de parada se ponga a 1 con los criterios que explicaremos más adelante.

Finalmente, el sistema de Simulink queda de la siguiente manera:

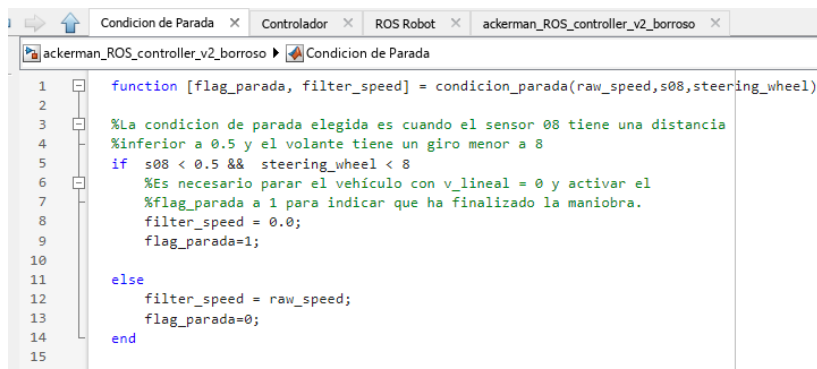


Tomamos 6 sensores del ROS Robot (s05,s06,s07,s08,s10,11). Añadimos a Controlador el controlador borroso.

Se puede observar una condición de parada que conecta con el bloque de ROS Robot, que a su vez conecta con un bloque de evaluación de maniobra especificado por los profesores de la asignatura, el cual no debe modificarse, y, por último, un multiplexor que acaba en el controlador visto anteriormente.



La condición de parada está pensada para que cuando el sónar 8 tenga un campo de visión menor a 0.5 y el giro del volante sea menor a 8, sabremos entonces que el robot está cercano a la pared de la plaza de aparcamiento y lleva un giro muy pequeño, por lo que en ese momento se efectuaría la parada.



The screenshot shows a MATLAB script editor with the following code:

```
1 function [flag_parada, filter_speed] = condicion_parada(raw_speed,s08,steering_wheel)
2
3 %La condicion de parada elegida es cuando el sensor 08 tiene una distancia
4 %inferior a 0.5 y el volante tiene un giro menor a 8
5 if s08 < 0.5 && steering_wheel < 8
6     %Es necesario parar el vehículo con v_lineal = 0 y activar el
7     %flag_parada a 1 para indicar que ha finalizado la maniobra.
8     filter_speed = 0.0;
9     flag_parada=1;
10
11 else
12     filter_speed = raw_speed;
13     flag_parada=0;
14 end
15
```

Por último, en la presentación de la práctica, se puede ver el funcionamiento de este sistema borroso de tipo Mamdani, con los entornos definitivo y original.

## PARTE II. Diseño automático de un controlador neuronal

Esta segunda parte de la práctica final se trata de solventar el mismo problema anterior, pero utilizando una red neuronal, la cual es necesaria de entrenar para su correcto funcionamiento.

Para el entrenamiento de la red neuronal hemos optado por la opción de usar nuestro controlador borroso, ya que ejecuta correctamente la maniobra para los dos entornos proporcionados.

Para ello, creamos el script en Matlab 'Entrenamiento', que se apoya en el script de Matlab `maniobra_park_ackerman_datos_entrenamiento_alumnos.m` (incluido en los archivos de la práctica)

Para la ejecución del control manual, tal y como se especifica en el enunciado, ha sido necesario hacer, en la máquina virtual, un *ifconfig*, de tal manera que conozcamos la IP del ordenador, para poder conectarlo a Matlab y lo necesario.

```
alumno@alumno-MV:~$ ifconfig
enp0s3: flags=4163<UP,BROADCAST,RUNNING>
    inet 192.168.1.33 netmask 255.255.255.0
    inet6 fe80::4437:72a1:21f:959
```

De esta manera, la variable `ROS_MASTER_IP` tiene que ser igual a la IP señalada en la imagen anterior en amarillo.

Antes de comenzar con la generación de datos y el entrenamiento de la red neuronal, se establece la conexión con el entorno ROS. Se limpian las variables existentes, y se apaga cualquier instancia ROS previa.

A continuación, se inicia la conexión ROS con la dirección IP '192.168.1.33'. Se declaran variables globales para almacenar datos relevantes de los sensores y el control del robot.

```
roshutdown

rosinit('192.168.1.33')

global steering_wheel_angle;
global vel_lineal_ackerman_kmh;
global s00;
global s01;
global s02;
global s03;
global s04;
global s05;
global s06;
global s07;
global s08;
global s09;
global s10;
global s11;
```



Después de realizar la conexión, se inicializan los suscriptores y publicadores ROS, así como se generan mensajes para el control de velocidad del robot. Se espera hasta recibir un mensaje relacionado con el robot para asegurar una inicialización adecuada.

```
%DECLARACIÓN DE SUBSCRIBERS
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%Odometria
sub_odom=rossubscriber('/robot0/odom');
%Laser
sub_laser = rossubscriber('/robot0/laser_1', rostype.sensor_msgs_LaserScan);
%Sonars

sonar_0 = rossubscriber('/robot0/sonar_0', rostype.sensor_msgs_Range);
sonar_1 = rossubscriber('/robot0/sonar_1', rostype.sensor_msgs_Range);
sonar_2 = rossubscriber('/robot0/sonar_2', rostype.sensor_msgs_Range);
sonar_3 = rossubscriber('/robot0/sonar_3', rostype.sensor_msgs_Range);
sonar_4 = rossubscriber('/robot0/sonar_4', rostype.sensor_msgs_Range);
sonar_5 = rossubscriber('/robot0/sonar_5', rostype.sensor_msgs_Range);
sonar_6 = rossubscriber('/robot0/sonar_6', rostype.sensor_msgs_Range);
sonar_7 = rossubscriber('/robot0/sonar_7', rostype.sensor_msgs_Range);
sonar_8 = rossubscriber('/robot0/sonar_8', rostype.sensor_msgs_Range);
sonar_9 = rossubscriber('/robot0/sonar_9', rostype.sensor_msgs_Range);
sonar_10 = rossubscriber('/robot0/sonar_10', rostype.sensor_msgs_Range);
sonar_11 = rossubscriber('/robot0/sonar_11', rostype.sensor_msgs_Range);

%DECLARACIÓN DE PUBLISHERS
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%Velocidad
pub_vel=rospublisher('/robot0/cmd_vel', 'geometry_msgs/Twist');

%GENERACION DE MENSAJES
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
msg_vel=rosmesssage(pub_vel);

%Definimos la periodicidad del bucle
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
r=robotics.Rate(10);

%Nos aseguramos de recibir un mensaje relacionado con el robot
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
while (strcmp(sub_odom.LatestMessage.ChildFrameId,'robot0')~=1)
    sub_odom.LatestMessage
end

disp('Inicialización ACKERMAN finalizada correctamente');
```

Se crea un bucle que realiza maniobras de aparcamiento simuladas. Utilizando N como numero de iteraciones sobre los entornos.

```
74     training_data=[];
75
76     %numero de maniobras para entreno
77     N=10
78     %Vectores para almacenar los datos de entrenamiento
79     Sensores_vec=[];
80     angVol_vec=[];
81     velLin_vec=[];
```

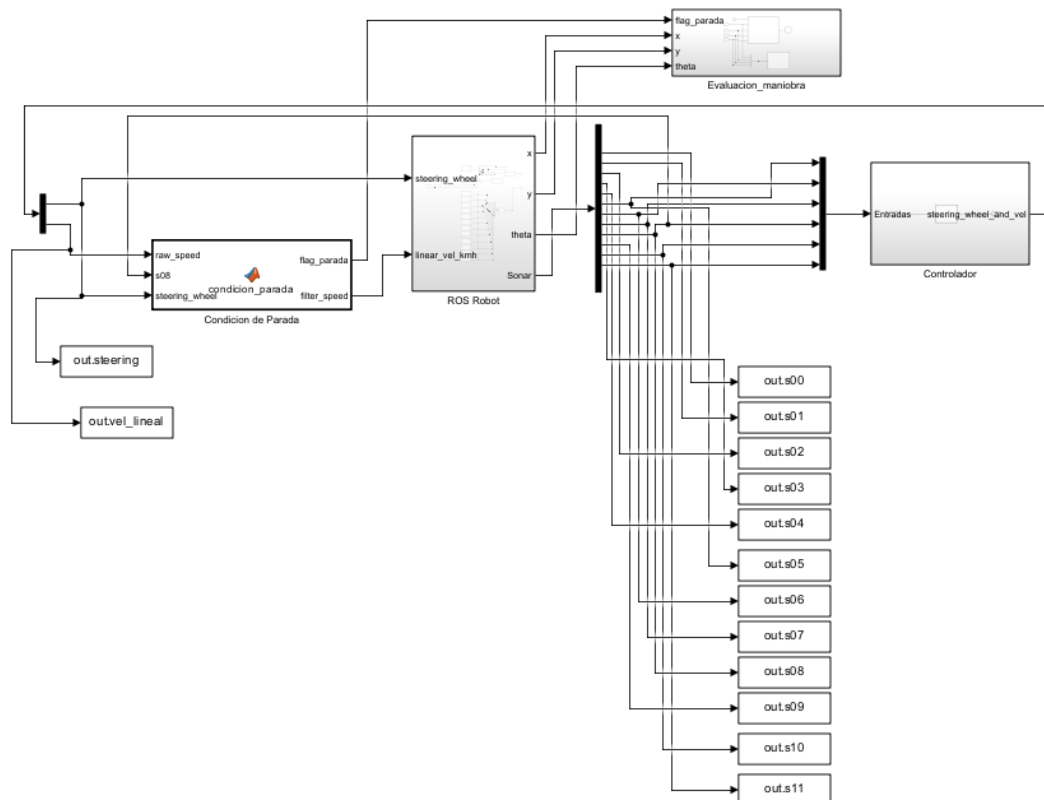
Y declaramos las matrices que irán almacenando los datos en cada una de ellas.

```

82 %Bucle para crear los datos de entrenamiento
83 for i=0:N
84
85 % Recorrido de aparcamiento para obtener datos de entrenamiento con
86 % controlador borroso.
87
88 sim('ackerman_ROS_controller_v2_training.slx')
89
90 %recogida de los datos desde simulink
91
92 %velocidad lineal y giro de volante
93 vel_lineal_ackerman_kmh = ans.steering.signals.values; %(km/h)
94 steering_wheel_angle = ans.vel_lineal.signals.values; % desde -90 a 90 grados.
95
96 %sensores del robot
97 s00=ans.s00.signals.values;
98 s01=ans.s01.signals.values;
99 s02=ans.s02.signals.values;
100 s03=ans.s03.signals.values;
101 s04=ans.s04.signals.values;
102 s05=ans.s05.signals.values;
103 s06=ans.s06.signals.values;
104 s07=ans.s07.signals.values;
105 s08=ans.s08.signals.values;
106 s09=ans.s09.signals.values;
107 s10=ans.s10.signals.values;
108 s11=ans.s11.signals.values;

```

Empezamos por lanzar nuestro controlador borroso implementado en Simulink y con las modificaciones necesarias para capturar los valores de salida necesarios, como podemos ver en la siguiente imagen.



Los datos de velocidad lineal, ángulo de dirección y lecturas de sensores son recopilados y almacenados en matrices.

```

110      %Matriz que recoge los valores de todos los sensores
111      medidas_sonar = [s00, s01, s02, s03, s04, s05, s06, s07, s08, s09, s10, s11];
112      medidas_sonar(isinf(medidas_sonar)) = 5.0;
113
114      %pausa para cerrar y reiniciar el mapa de entrenamiento
115      disp('inicio pausa');
116      pause(10)
117      disp('fin pausa');
118
119      %Acotamos el numero de filas para que siempre sean las mismas
120      medidas_sonar = medidas_sonar(1:271,:);
121      steering_wheel_angle = steering_wheel_angle(1:271,:);
122      vel_lineal_ackerman_kmh = vel_lineal_ackerman_kmh(1:271,:);
123
124      %Almacenamos todos los valores de los sensores a los anteriores
125      Sensores_vec=[Sensores_vec;medidas_sonar];
126
127      %Almacenamos todos los valores de velocidad y giro de volantes a los
128      %ya almacenados
129      angVol_vec=[angVol_vec;steering_wheel_angle];
130      vellin_vec=[vellin_vec;vel_lineal_ackerman_kmh];
131
132      end

```

Luego haremos una pausa de 10 segundos para reiniciar el robot con el mapa deseado y preparar la siguiente iteracion en ROS.

Finalmente se acotan las matrices de datos, puesto que se producen pequeñas variaciones en el tiempo de la maniobra al capturarlas y esto nos podría dar una o dos filas mas. Generando un problema en el momento de concatenarlas con las matrices creadas anteriormente.

Seguidamente, concatenamos los matrices con las de las iteraciones anteriores y empezamos con la siguiente iteracion.

Los datos recopilados se utilizan para entrenar una red neuronal de alimentación directa con 8 neuronas en la capa oculta. La red se configura, entrena y luego se genera un bloque de Simulink para implementar el control neuronal. Está entrenada con 10 maniobras, 5 en cada entorno, original y definitivo, de manera alterna.

```

%Se almacenan todos las matrices de valores en training data
training_data=[Sensores_vec,vellin_vec,angVol_vec];

save datos_entrenamiento training_data

%Generamos los valores de entrada de la nn con los sensores
inputs = training_data(:,[1:12])';
%Generamos los valores de salida de la nn con la velocidad y el giro
outputs = training_data(:,[13:14])';
inputs(isinf(inputs)) = 5.0;
inputs = double(inputs);
outputs = double(outputs);

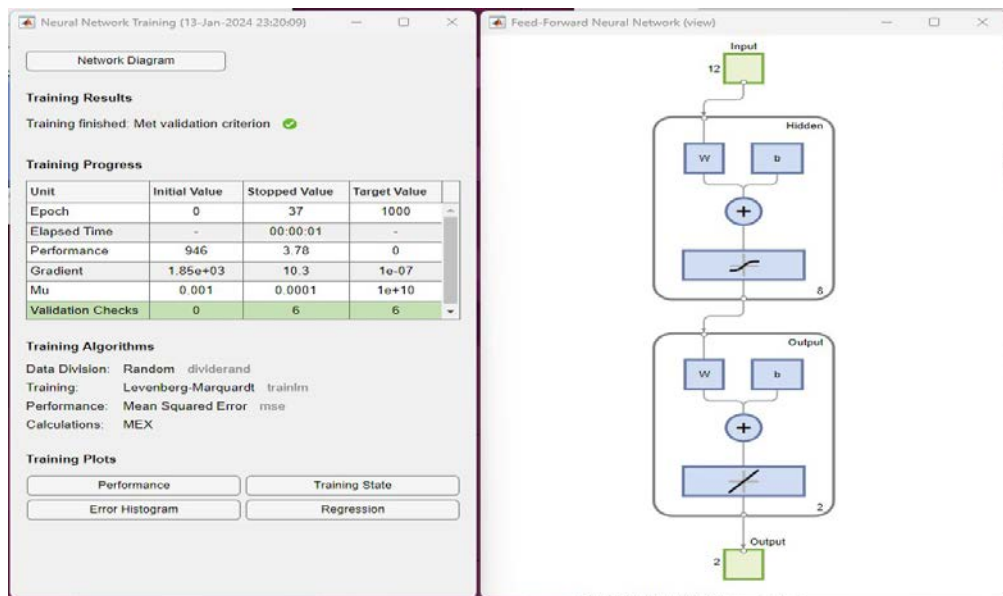
% Entrenar red neuronal con 8 neuronas en la capa oculta
net = feedforwardnet([8]);
net = configure(net,inputs,outputs);
net = train(net,inputs,outputs);

% Generar bloque de Simulink con el controlador neuronal
gensim(net)

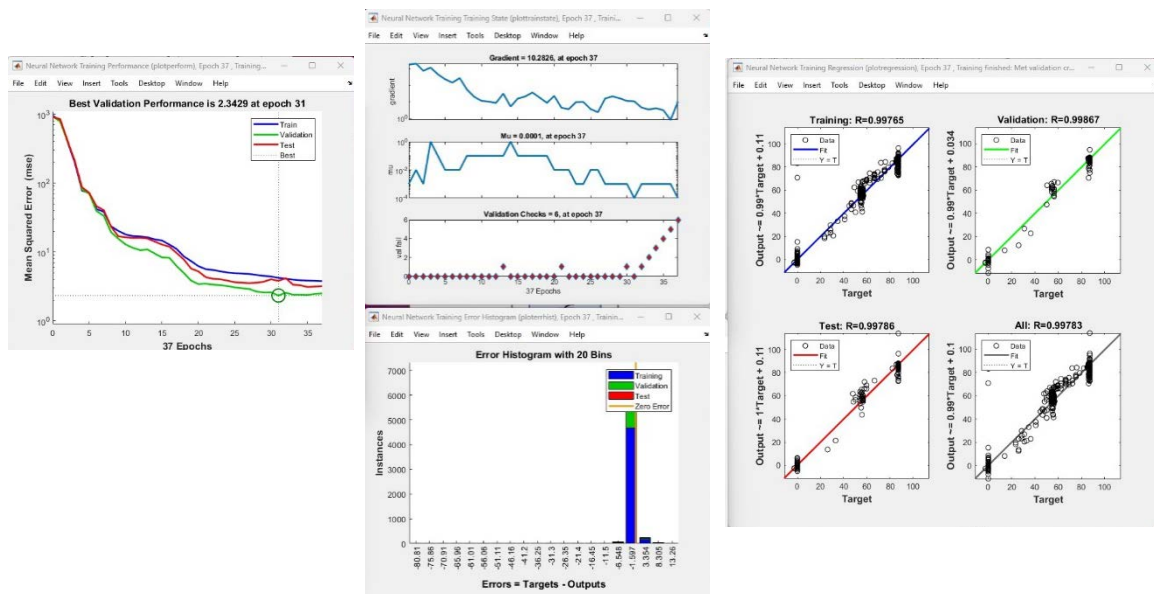
```

Este código en MATLAB se utiliza para conectar con un entorno ROS, inicializar un simulador de robot Ackerman, generar datos de entrenamiento a través de simulaciones de aparcamiento, y finalmente entrenar una red neuronal con esos datos.

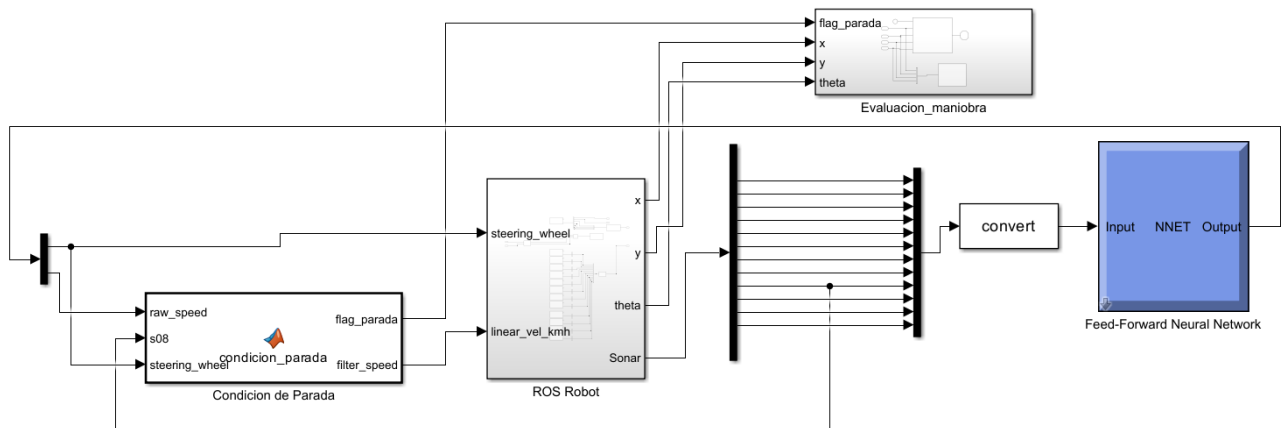
En la siguiente figura podemos ver la información relacionada con Neuronal Network y Network Diagram:



En la siguiente imagen se muestra la información de Performance, Training State, Error Histogram y Regression:



Finalmente, sustituimos nuestro controlador borroso por la red neuronal generada. Para ellos usamos como entrada todos los sensores y el bloque “Data Type Conversion” entre la salida del multiplexor de los ultrasonidos y la entrada de la red. De este modo nos queda el siguiente sistema:



## Conclusiones

Como conclusiones se puede concluir que, el controlador borroso ofrece mejores resultados ante circuitos que se conocen perfectamente las funciones de pertenencia, por lo que supera al controlador neuronal.

Por otra parte, la eficacia del controlador neuronal depende considerablemente de la calidad de los datos de entrenamiento.

Otra conclusión que hemos sacado es que a la hora de entrenar el sistema, funcionaba mejor cuando intercalábamos entre un mapa y otro, el original y el definitivo.