Universidad de Alcalá

Práctica Final

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

Sistemas de Control Inteligente

Daniel Senespleda Fernández 05953542S

Tabla de contenido

Introducción

Este trabajo se ha dividido en dos partes, ambas centradas en conseguir que un robot móvil tipo Ackerman sea capaz de aparcar de forma autónoma, como si se tratara de un coche real.

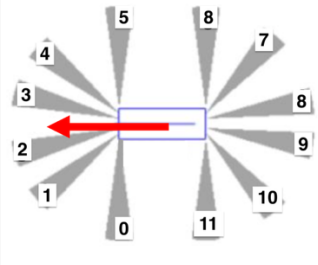
En la primera parte, se diseñó un controlador borroso (fuzzy) que tomaba decisiones basándose en la información que le daban los sensores ultrasónicos del robot. La idea era que, el controlador pudiera generar giros y velocidades en función de ciertas condiciones difusas, como "muy cerca del obstáculo" o "girando bastante". Este enfoque permitió trabajar con lógica aproximada y ajustar el comportamiento del robot para que aparcara de forma segura.

En la segunda parte, el objetivo fue imitar una maniobra de aparcamiento correcta mediante el entrenamiento de una red neuronal. Para ello, primero se diseñó y ejecutó una maniobra de aparcamiento bien hecha en el simulador, recopilando datos como distancias a obstáculos, velocidades y ángulos de dirección. Con esos datos se entrenó una red neuronal feedforward, que luego fue integrada en Simulink como bloque funcional. Así, el robot aprendía a aparcar sin necesidad de reglas explícitas, simplemente reproduciendo el comportamiento que había visto durante el entrenamiento.

Ambas partes han sido probadas en el entorno de simulación utilizando ROS, Simulink y MATLAB, lo que ha permitido ir ajustando el comportamiento del robot hasta conseguir maniobras estables y realistas.

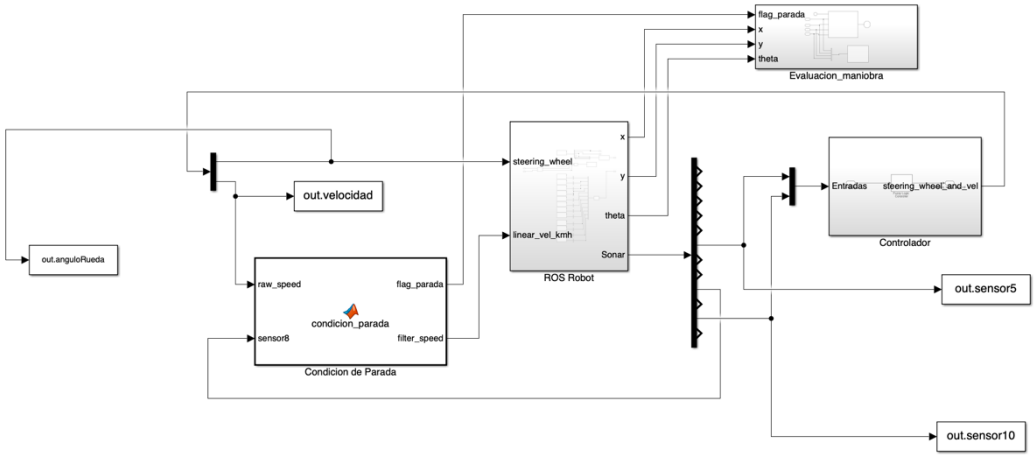
Parte 1. Diseño manual de un control borroso de tipo MAMDANI.

Tras muchos intentos y bajo el hecho de que se nos pedía usar los menores sónares posibles, se ha decidido que los únicos que necesitamos para el desarrollo de esta parte de la práctica son los sónares 5 y 10.



Sonares del robot según la posición en el entorno de la práctica

En el archivo de Simulink “CB_ackerman_ROS_controller” se han conectado las salidas del multiplexor correspondientes con el bloque del controlador.



Como se puede ver en la imagen, además se incluyeron bloques “ToWorkspace” para poder ver bien los valores que recibían los sensores, el ángulo de la rueda y de la velocidad líneal para el desarrollo de un correcto Controlador Borroso.

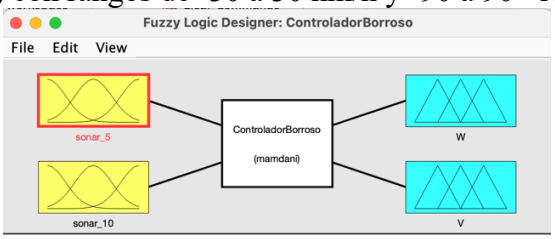
1.- Condición de Parada

Respecto al bloque de la condición de parada, vimos que el único momento en el que el sonar 8 marcaba una distancia menor a 1 era cuando el robot ya estaba bien metido en la plaza (al menos en nuestro caso y con nuestra maniobra). Por eso, decidimos usar esa condición dentro de un “if”: si se cumple, se pone la velocidad lineal a 0 y se activa una bandera de parada que hace que se termine la ejecución del archivo. Si no se cumple, el robot sigue con la velocidad actual y no se activa nada.

```
function [flag_parada, filter_speed] = condicion_parada(raw_speed, sensor8)

% Comprobar si se cumplen todas las condiciones
if (sensor8 < 1)
    % Detener el vehículo y marcar la condición de parada
    filter_speed = 0.0; % Velocidad lineal = 0
    flag_parada = 1;    % Activar bandera de parada
else
    % Continuar movimiento
    filter_speed = raw_speed; % Mantener la velocidad actual
    flag_parada = 0;          % No activar la bandera de parada
end

end
```

2.- ControladorBorroso.fis

Este archivo .fis es el propio controlador borroso en sí. Este controlador tiene dos entradas, que corresponden con los sónares mencionados anteriormente, con rangos de 0 a 5 metros. Por otro lado, las dos salidas corresponden con la velocidad lineal (V) y el ángulo de la rueda (W) con rangos de -30 a 30 km/h y -90 a 90 º respectivamente.



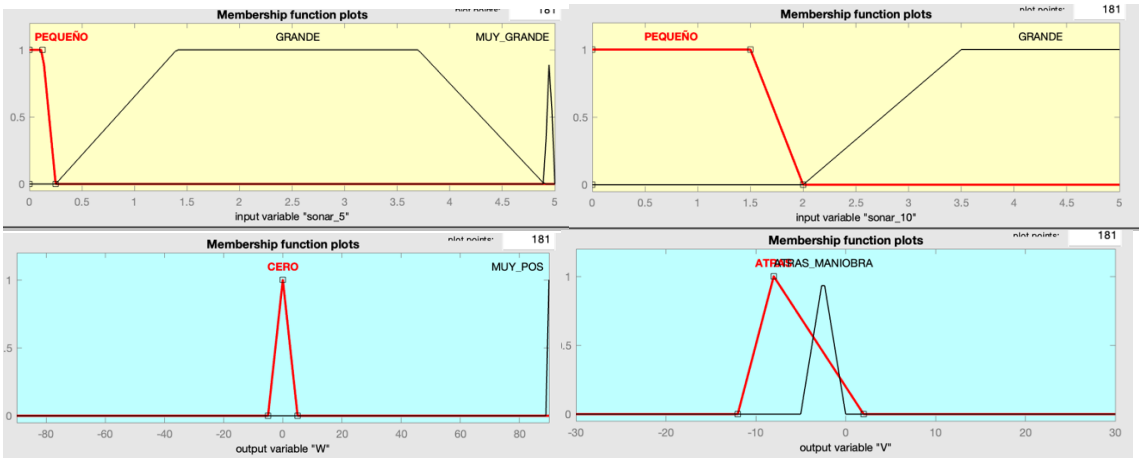
2.1.- Funciones de Pertenencia

Estas funciones están diseñadas de forma que quede claro cuándo se cumple cada una, asegurando que en ningún momento se activen dos a la vez. La única excepción es la velocidad lineal, ya que puede coincidir en distintos casos, pero eso no afecta a la lógica del sistema porque es una salida, no una condición.

Además, las funciones asociadas a las entradas, que sí determinan el comportamiento del sistema, están definidas de forma que cubren todo el rango de valores posibles. Esto asegura que el programa no se quede bloqueado en ningún momento por falta de respuesta ante una entrada concreta.

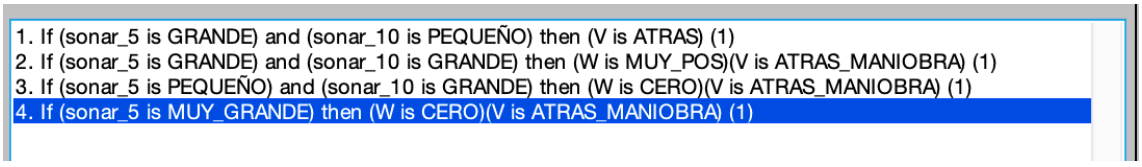
Las distancias de cada conjunto en las funciones de pertenencia se han definido según las necesidades del sistema y ajustándolas a partir de lo que se observaba en cada prueba.

A continuación, podemos ver dichas funciones de cada una de las variables de entrada y salida mencionadas.



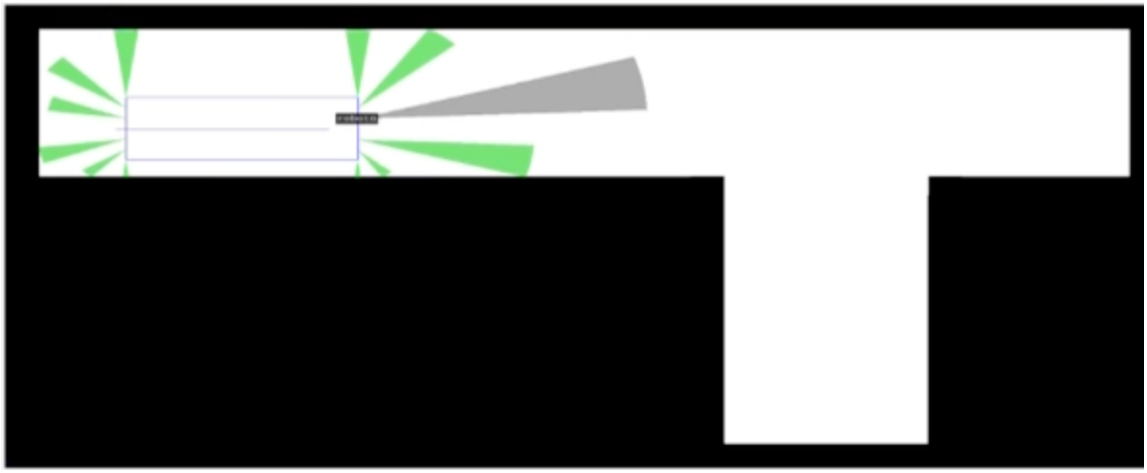
2.2.- Reglas del sistema

A continuación, se hablará de las reglas establecidas para el funcionamiento de nuestro controlador. Como se puede apreciar en la siguiente imagen, únicamente hemos necesitado de 4 reglas.



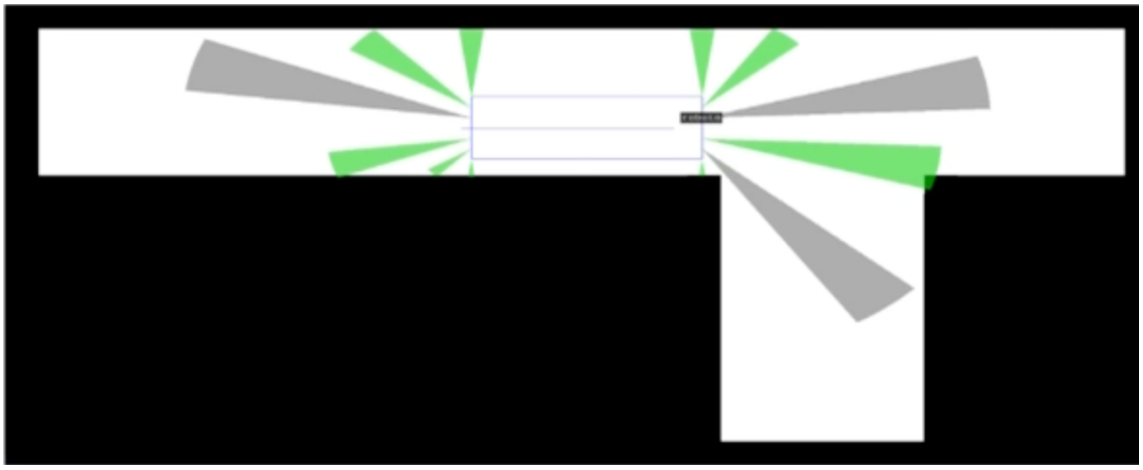
Regla 1

Sirve para el primer movimiento del robot, el cual queremos que sea simplemente marchar hacia atrás en línea recta. Por lo tanto, establecemos la velocidad líneal con un valor negativo (ATRAS) y el ángulo de la rueda no lo cambiamos (none), puesto que ya se encuentra a cero, como lo queremos.



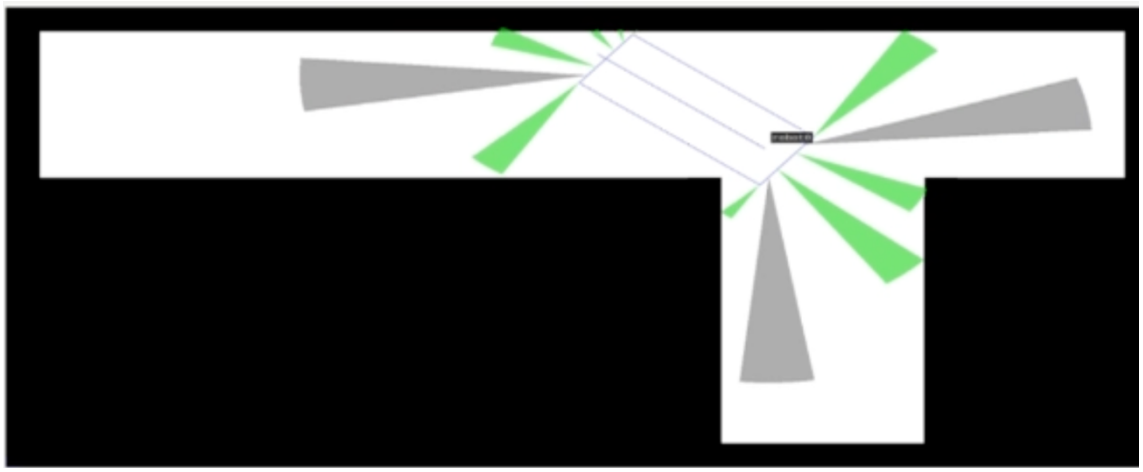
Regla 2

Esta regla se activa cuando el sonar 10 detecta un aumento en la distancia (cuando esta supera los 2 metros). En ese momento, el sistema reduce la velocidad para asegurarse de que la maniobra se realice con la mayor precisión posible. Además, se gira la rueda al máximo hacia la izquierda (90º) para facilitar el giro.

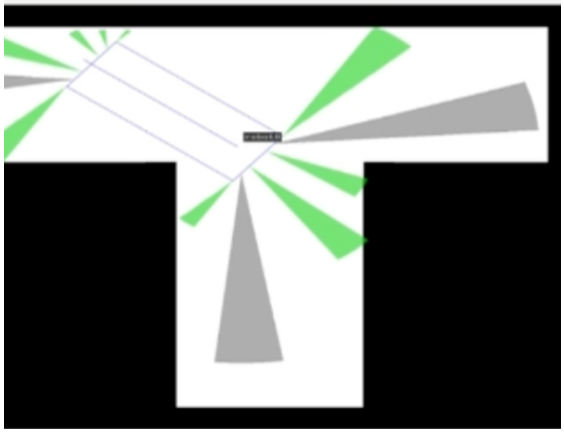


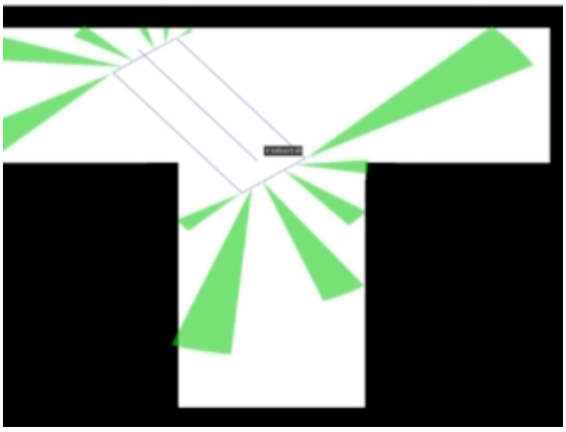
Regla 3

Al comenzar la maniobra con la regla anterior, el robot llega a un punto en el que se acerca demasiado a la pared superior. Si no existiera la regla 3, seguiría girando y acabaría chocando. Por suerte, el sonar 5 detecta la proximidad de la pared y cambia de estado GRANDE a PEQUEÑO, lo que hace que el sistema ajuste el ángulo de la rueda a CERO, haciendo que el robot avance recto para alejarse.



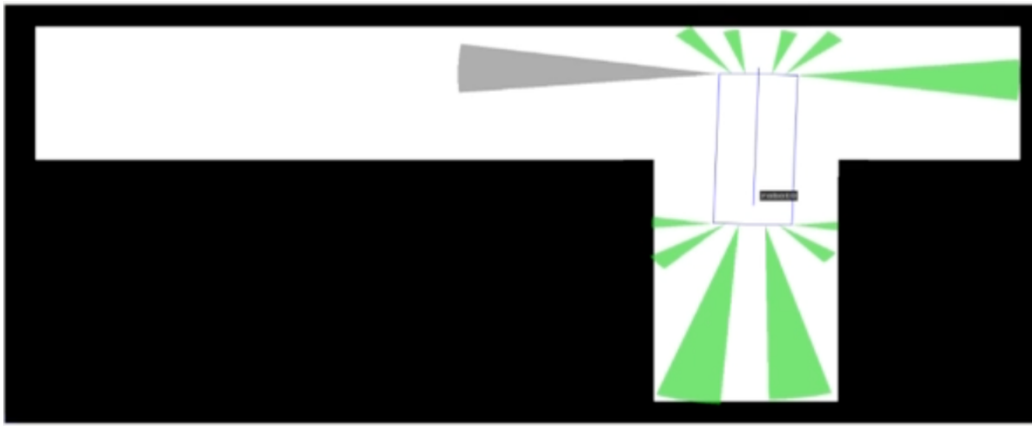
Cuando se ha alejado lo suficiente, vuelve a cumplirse la regla 2, lo que provoca que el robot se acerque otra vez a la pared y se active de nuevo la regla 3. Este bucle se repite un par de veces hasta que el robot ya está a una distancia segura y puede continuar la maniobra de giro sin problema.





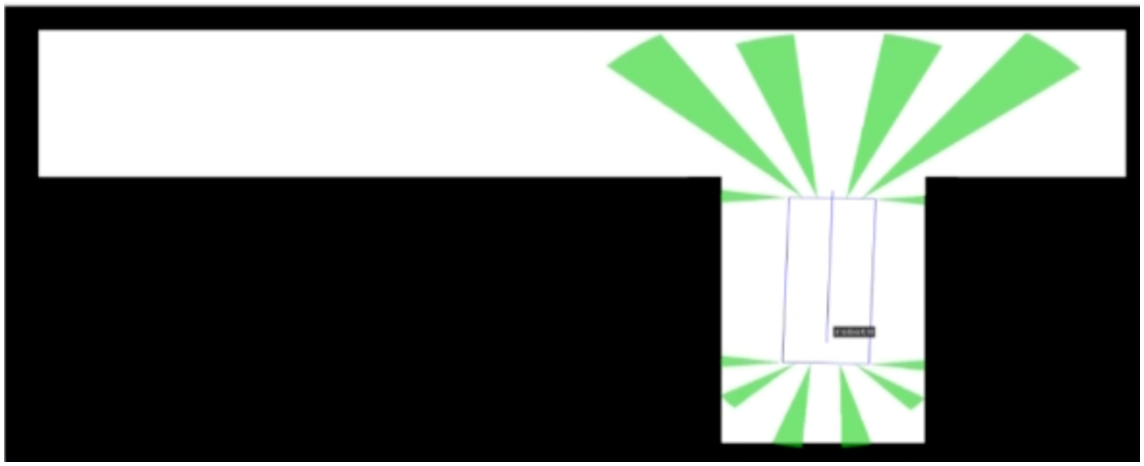
Regla 4

La última regla se activa cuando el sonar 5 detecta la pared a la máxima distancia posible (5 metros). Esto indica que el robot ya está bien colocado para empezar a aparcar, así que dejamos de girar y ponemos el ángulo de la rueda en cero para que entre recto en la plaza.

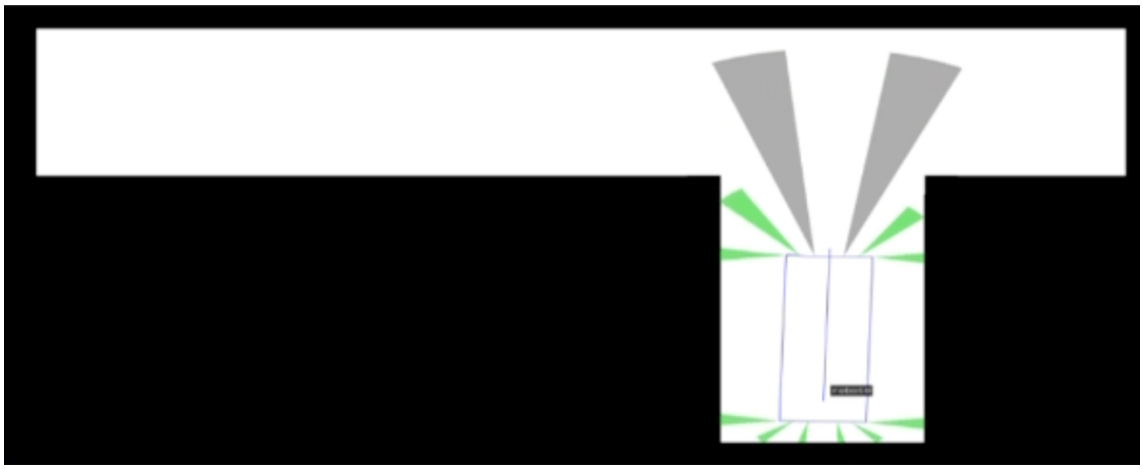


Continuación

Una vez que el robot ya está dentro de la plaza, el sonar 5 vuelve a medir distancias dentro del rango “GRANDE” y el sonar 10 dentro del rango “PEQUEÑO”, lo que hace que el sistema vuelva a la regla 1. Sin embargo, este cambio no afecta al comportamiento, ya que ambas reglas establecen los mismos valores de salida. Como mucho, puede tener un pequeño efecto si el robot ha girado de más en la regla anterior y no ha quedado del todo perpendicular a la plaza, en cuyo caso ajustará un poco la dirección para colocarse bien.

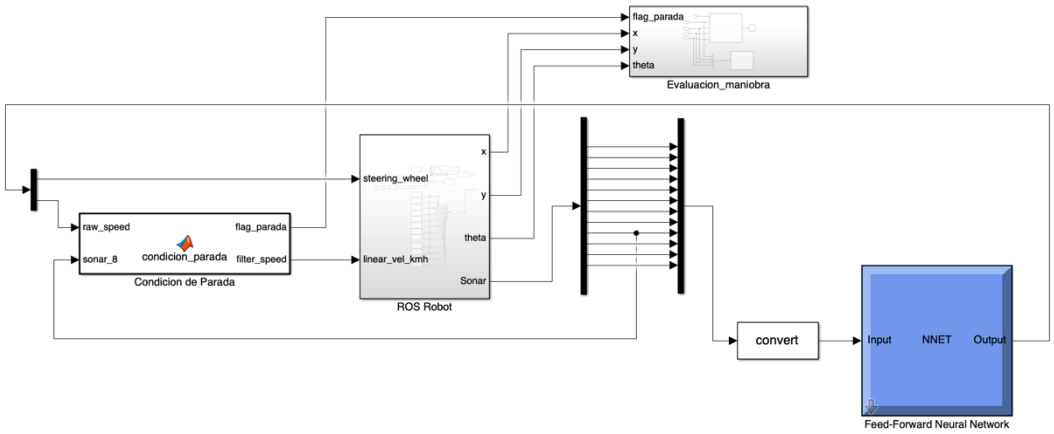


Por último, cuando el coche ya está lo suficientemente dentro de la plaza se cumple la condición de parada explicada anteriormente, por lo que el sistema termina.



Parte 2 Diseño de un Controlador Neuronal

En esta parte del trabajo se ha creado una Red Neuronal, la cual se ha sustituido por el controlador borroso dentro del archivo "CB_ackerman_ROS_controller" quedándonos de la siguiente forma:



Más adelante explicaremos porque la red neuronal tiene como entradas todos los sónares, no como el controlador borroso, que solo tenía los necesarios.

1.- Obtener los datos de entrenamiento

Para obtener los datos de entrenamiento a partir de los cuales entrenar la red neuronal, primero se optó por la opción "c)" del enunciado, obtenerlos a partir del controlador borroso creado en la parte 1. Por lo tanto, se incluyó un bloque tipo "ToFile" que guardara los datos de velocidad, ángulo de la rueda y los dos sensores que intervenían en el proceso. Sin embargo, al obtener la Red Neuronal y probarla en el entorno no se obtenían los resultados esperados. Supuse que era porque no disponía de suficientes datos como para ejecutarse correctamente, por lo que se decidió que se ejecutaría mejor la opción "b)", la cual consiste en usar un recorrido prefijado.

Se ha partido del archivo original maniobra_park_ackerman_datos_entrenamiento_alumnos.m, que venía con la práctica. Sin embargo, dicho archivo ha sido modificado. A continuación, se describen los principales cambios realizados:

1.1. Conexión explícita a ROS

En la versión modificada, se ha especificado de forma explícita la IP del *ROS Master* y del *NodeHost*:

```
rosinit('http://172.20.10.13:11311', 'Nodehost', '172.20.10.2');
```

1.2. Suscripción completa a todos los sonares

Aunque en el archivo base algunos sensores podían estar definidos o utilizados parcialmente, en esta versión se ha hecho una suscripción explícita y completa a los 12 sensores de ultrasonidos:

```
%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);
```

Esto permite recopilar datos de todos los sensores y tener una mayor flexibilidad a la hora de diseñar recorridos, extraer características, o incluso ampliar la red neuronal para que utilice más entradas si se desea.

1.3. Separación en recorridos prefijados

En lugar de depender del controlador difuso para que genere automáticamente los movimientos y guardar sus salidas, se ha optado por definir manualmente los movimientos en una serie de "recorridos prefijados", como este:

```
%% RECORRIDO 1

distancia=5.8

vel_lineal_ackerman_kmh = -5
steering_wheel_angle =  0
avanzar_ackerman

distancia= 5.5

vel_lineal_ackerman_kmh = -3.8
steering_wheel_angle =  85
avanzar_ackerman

distancia=2.9

vel_lineal_ackerman_kmh = -2.5
steering_wheel_angle =  0
avanzar_ackerman
```

Cada tramo simula una fase concreta del aparcamiento (recto, giro, corrección, etc.), lo que permite tener un control total sobre los datos que se están generando, incluyendo sensores, velocidad y ángulo.

Este cambio se hizo tras observar que el método original (opción c del enunciado, usando directamente el controlador borroso) no generaba resultados satisfactorios al entrenar la red, probablemente por falta de diversidad o cantidad de datos.

1.4. Sistema de guardado acumulativo

Otra diferencia clave es que el nuevo script incluye la opción de mantener y ampliar un conjunto de entrenamiento acumulado:

```
%% Cargar datos previos si existen
if isfile("datosEntrenamiento.mat")
    load("datosEntrenamiento.mat", "training_data");
else
    training_data = [];
end
```

...

```
%% Guardar entrenamiento acumulado
save("datosEntrenamiento.mat", "training_data");
disp('Datos añadidos correctamente al conjunto de entrenamiento.');
```

Esto permite que cada vez que se ejecuta el script se añadan nuevos ejemplos al conjunto total, facilitando así la mejora progresiva del entrenamiento sin sobrescribir los datos anteriores.

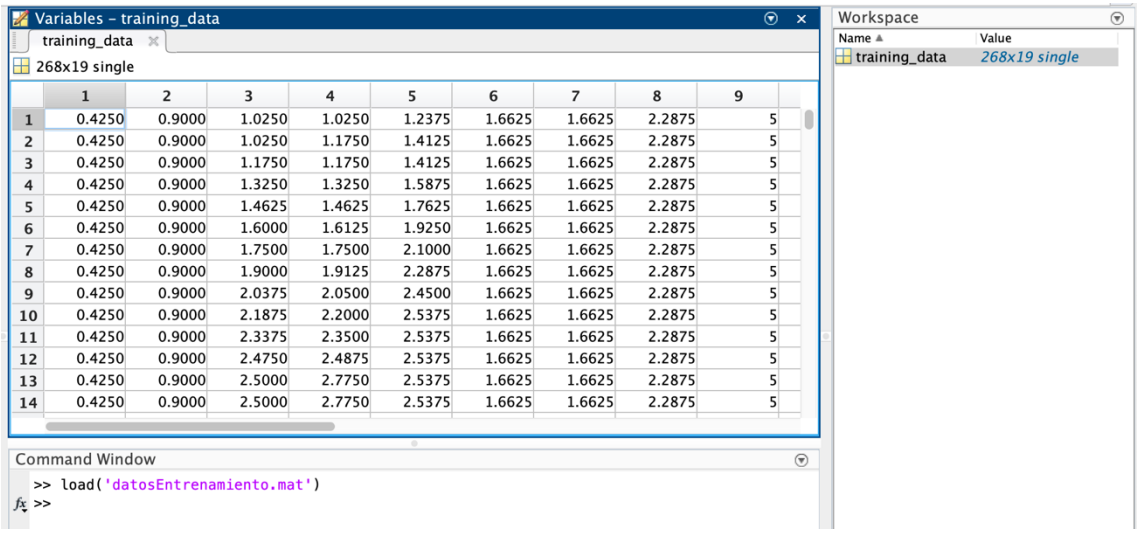
1.5. Facilidad para añadir recorridos adicionales

Se ha preparado un segundo bloque de recorrido (RECORRIDO 2), comentado por defecto, que permite generar nuevos ejemplos sin tener que duplicar o editar manualmente el archivo.

```
%% RECORRIDO 2
%Comentado por defecto, descomentar y comentar el recorrido 1 para obtener
%mas datos de entrenamiento
%{
distancia = 6.3;
vel_lineal_ackerman_kmh = -5.2;
steering_wheel_angle = 0;
avanzar_ackerman;

distancia = 1.0;
vel_lineal_ackerman_kmh = -3.7;
steering_wheel_angle = 90;
avanzar_ackerman;

distancia = 1.0;
vel_lineal_ackerman_kmh = -3.7;
steering_wheel_angle = 0;
avanzar_ackerman;

distancia = 2.0;
vel_lineal_ackerman_kmh = -3.7;
steering_wheel_angle = 90;
avanzar_ackerman;

distancia = 2.8;
vel_lineal_ackerman_kmh = -2.6;
steering_wheel_angle = 0;
avanzar_ackerman;
%}
```

Así, se puede alternar entre varios escenarios de aparcamiento sin reescribir el código.

Gracias a todos estos cambios, la obtención de los datos consistió en ejecutar este script dos veces, primero con el recorrido 2 comentado y el 1 descomentado y luego al revés. De esta forma obtuvimos esta tabla de datos:



	1	2	3	4	5	6	7	8	9
1	0.4250	0.9000	1.0250	1.0250	1.2375	1.6625	1.6625	2.2875	5
2	0.4250	0.9000	1.0250	1.1750	1.4125	1.6625	1.6625	2.2875	5
3	0.4250	0.9000	1.1750	1.1750	1.4125	1.6625	1.6625	2.2875	5
4	0.4250	0.9000	1.3250	1.3250	1.5875	1.6625	1.6625	2.2875	5
5	0.4250	0.9000	1.4625	1.4625	1.7625	1.6625	1.6625	2.2875	5
6	0.4250	0.9000	1.6000	1.6125	1.9250	1.6625	1.6625	2.2875	5
7	0.4250	0.9000	1.7500	1.7500	2.1000	1.6625	1.6625	2.2875	5
8	0.4250	0.9000	1.9000	1.9125	2.2875	1.6625	1.6625	2.2875	5
9	0.4250	0.9000	2.0375	2.0500	2.4500	1.6625	1.6625	2.2875	5
10	0.4250	0.9000	2.1875	2.2000	2.5375	1.6625	1.6625	2.2875	5
11	0.4250	0.9000	2.3375	2.3500	2.5375	1.6625	1.6625	2.2875	5
12	0.4250	0.9000	2.4750	2.4875	2.5375	1.6625	1.6625	2.2875	5
13	0.4250	0.9000	2.5000	2.7750	2.5375	1.6625	1.6625	2.2875	5
14	0.4250	0.9000	2.5000	2.7750	2.5375	1.6625	1.6625	2.2875	5

2.- Entrenar la Red Neuronal

Una vez obtenidos los datos de entrenamiento, se ha creado y entrenado la red neuronal a partir del archivo “Entrenar_Red_Neuronal.m”. Este script ha sido diseñado para utilizar los datos generados por el script anterior (“maniobra_park_ackerman_datos_entrenamiento_alumnos.m”, modificado) y construir una red neuronal *feedforward* capaz de predecir los valores de velocidad lineal y ángulo del volante a partir de las lecturas de los sensores ultrasónicos del robot.

A continuación, se detallan los pasos fundamentales del proceso:

2.1.- Carga y preprocesado de datos

```
% --- Cargar datos ---
training_data = load("datosEntrenamiento").training_data;
```

Se cargan los datos acumulados previamente, que incluyen las lecturas de los sensores (columnas 1 a 12) y los comandos de control aplicados (columnas 18 y 19). Luego se extraen y preparan las entradas y salidas para la red:

```
% --- Preparar inputs y outputs ---
inputs = training_data(:,1:12);
outputs = training_data(:,18:19);
```

Antes de continuar, se reemplazan los valores inf de los sensores por 5.0, que es el valor máximo que puede devolver un sensor ultrasónico en este contexto:

Finalmente, se convierten los datos a tipo “double”, que es el tipo que esperan las redes neuronales en Matlab

2.2.- Creación y configuración de la red neuronal

```
% --- Crear y configurar red ---
net = feedforwardnet([10,3,3]);
net = configure(net, inputs, outputs);
```

Se ha optado por una red de tipo *feedforward* (perceptrón multicapa), con una arquitectura de **tres capas ocultas**: la primera con 10 neuronas, y las dos siguientes con 3 neuronas cada una. Después, se configura la red con los datos de entrada y salida y se desactivan los procesos automáticos de normalización de datos (normalmente activos por defecto en MATLAB), para mantener los valores en el rango real en que fueron recogidos:

```
% Evitar normalización automática
net.inputs{1}.processFcns = {};
net.outputs{2}.processFcns = {};
```

Esto se hace para evitar que la red entrene sobre una versión transformada de los datos, lo cual podría afectar a su comportamiento cuando se integre posteriormente en Simulink, especialmente en entornos embebidos o de control físico real.

2.3. Entrenamiento de la red

```
% --- Entrenar la red ---
[net, tr] = train(net, inputs, outputs);
```

Se entrena la red neuronal con los datos proporcionados. El objeto tr recoge información sobre el proceso de entrenamiento, incluyendo los índices de los datos utilizados para entrenamiento, validación y prueba.

2.4. Generación del bloque Simulink

```
% --- Generar bloque Simulink ---
gensim(net);
```

Una vez entrenada la red, se genera automáticamente un bloque Simulink que implementa dicha red. Este bloque lo integramos directamente en el archivo .slx para observar cómo se comporta.

2.5. Evaluación gráfica del rendimiento (fase de validación)

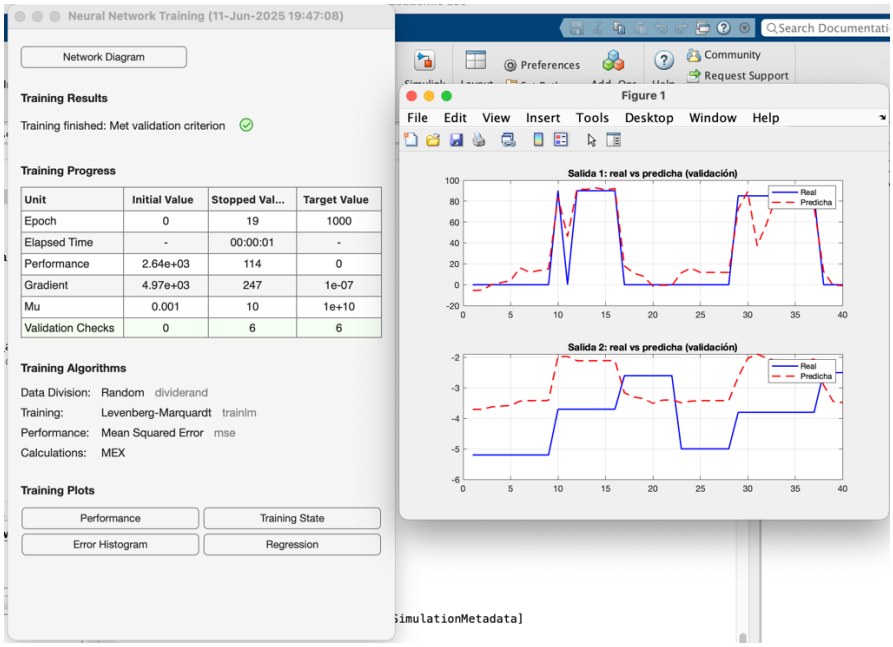
Si durante el entrenamiento se han asignado correctamente datos para la fase de validación, se generan gráficas comparando las salidas reales con las predichas por la red:

```
% --- Graficar resultados de validación si existen datos ---
if isempty(tr.valInd)
    disp('No hay datos para validación, no se muestran gráficas.');
else
    val_inputs = inputs(:, tr.valInd);
    val_outputs = outputs(:, tr.valInd);
    pred_outputs = net(val_inputs);

    figure;
    subplot(2,1,1);
    plot(val_outputs(1,:), 'b-', 'LineWidth', 1.5); hold on;
    plot(pred_outputs(1,:), 'r--', 'LineWidth', 1.5);
    title('Salida 1: real vs predicha (validación)');
    legend('Real','Predicha');
    grid on;

    subplot(2,1,2);
    plot(val_outputs(2,:), 'b-', 'LineWidth', 1.5); hold on;
    plot(pred_outputs(2,:), 'r--', 'LineWidth', 1.5);
    title('Salida 2: real vs predicha (validación)');
    legend('Real','Predicha');
    grid on;
end
```

Esto permite visualizar hasta qué punto la red ha logrado capturar el comportamiento deseado. Si no hay datos de validación (isempty(tr.valInd)), se avisa al usuario, aunque el entrenamiento se ha realizado igualmente.

Gracias a esto último, fui haciendo cambios en el código visualizando los resultados obtenidos en la gráfica, y una vez obtuviera una gráfica que me convenciera, entonces lo probaba en el archivo .slx.

La última red Neuronal creada (la que usamos) devolvió los siguientes datos:



En cuanto a la condición de parada, es la misma que en la parte 1, cuando la distancia detectada por el sonar 8 es menor que 1.

A continuación, se muestra un ejemplo de cómo aparcaría el robot en el escenario definitivo: