Sistemas de Control Inteligente

Aparcamiento Inteligente

Grado en Ingeniería Informática
Universidad de Alcalá

Pablo García García
Valeria Villamares Félix

14 de enero de 2024

Índice general

Introducción

El problema que se plantea y resuelve a lo largo de esta práctica de laboratorio es el diseño de dos controladores basados en técnicas populares de Inteligencia Artificial como son la Lógica Borrosa y las Redes Neuronales, que sean capaces de realizar el estacionamiento en batería de un vehículo.

Problema objetivo

Se cuenta con un escenario en el simulador STDR de ROS que simula el mapa en el que se deberá mover y aparcar el vehículo. Este cuenta con un carril de $24 \times 4{,}3$ metros, y ligeramente hacia la mitad de este, en perpendicular, se encuentra una plaza de aparcamiento de $7{,}8 \times 4{,}3$ metros. El frontal del vehículo se encuentra al principio de este carril, por lo que la maniobra deberá comenzar marcha atrás.

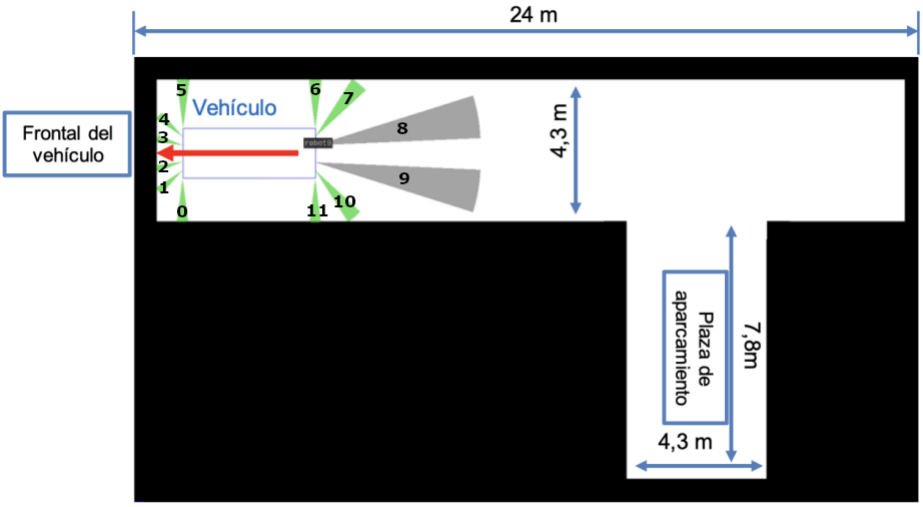

Figura 1: Escenario del problema

El vehículo a controlar tiene unas dimensiones de $4{,}5 \times 1{,}5$ metros, puede alcanzar velocidades de ± 30 km/h, y el volante puede girar $\pm 90^{\text{o}}$. Como es evidente, no es capaz de girar si no se está moviendo. Además, cuanta con doce sensores de ultrasonidos numerados del 0 al 11, capaces de medir distancias de hasta 5 metros. Finalmente, para el desarrollo de la práctica se empleará ROS, MATLAB, Simulink, y las toolboxes de *Fuzzy Logic*, *Deep Learning*, y *ROS*.

Condición de parada

Como el diseño de la condición de parada del simulador será común a los diferentes modelos y situaciones, será explicada en esta sección introductoria. Esta tarea consiste en que cuando el coche se encuentra en una posición similar a la mostrada en la Figura 2, fijando una distancia a la pared trasera, el vehículo deje de moverse y se pare la ejecución de Simulink.

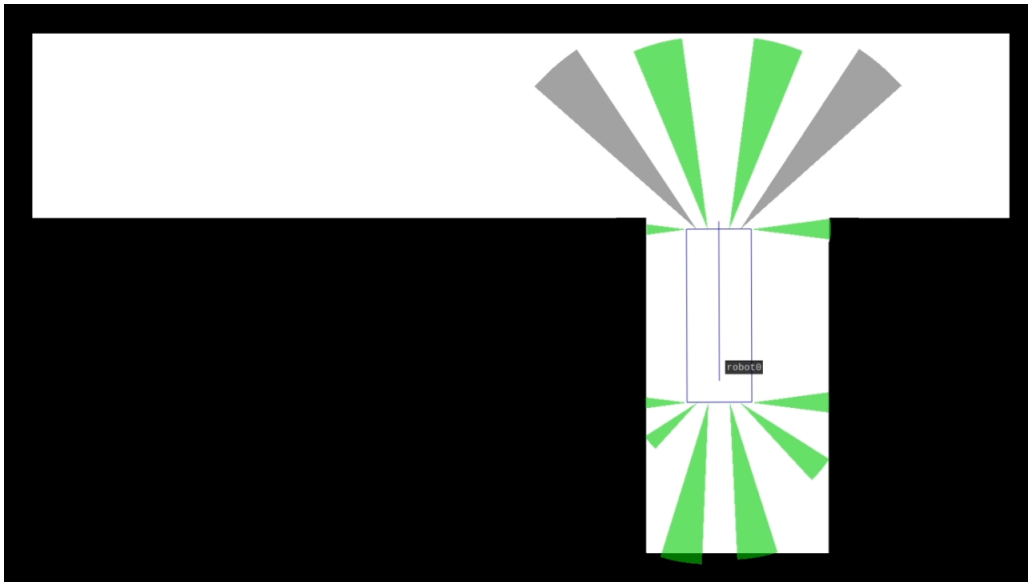

Figura 2: Posición de parada

Para ello, se introduce un bloque en Simulink que representa la ejecución de una función MATLAB, llamado "Condición de parada". Para fijar un umbral de distancia, se utilizarán los sensores 8 y 9, que son aquellos que se ubican en la parte trasera del coche. Como salida se tiene la velocidad modificada del coche y un *flag* con valor binario para avisar a Simulink de si la ejecución debe continuar o no.

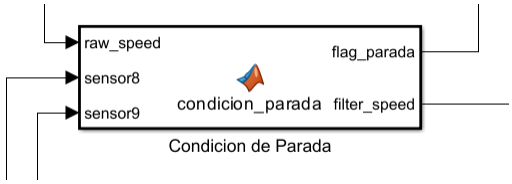

Figura 3: Bloque de condición de parada

El trabajo de este bloque es verificar si ambos sensores reciben una distancia menor a, por ejemplo 0,6 metros (este es el umbral que se fija), y en dicho caso activa el *flag* de parada y pone la velocidad a 0, y en caso contrario devuelve la velocidad que recibe del controlador. Se intenta dejar cerca de la pared, aprovechando el espacio de la plaza, pero sin que llegue a chocarse. El código que se encuentra dentro del bloque es el que se muestra a continuación.

```
function [flag_parada, filter_speed] = condicion_parada(raw_speed,
    sensor8, sensor9)

condicionS8 = sensor8 < 0.6;
condicionS9 = sensor9 < 0.6;

if condicionS8 && condicionS9
 filter_speed = 0.0;
 flag_parada = 1;
else
 filter_speed = raw_speed;
 flag_parada = 0;
end
```

Código 1: Código del bloque de parada

Parte 1

Controlador borroso

Durante la primera parte de esta práctica se abordará el diseño de un controlador borroso en MATLAB y que permita al vehículo realizar la maniobra de estacionamiento en batería.

1.1. Variables

En primer lugar, para llevar a cabo el diseño del controlador, se deben decidir cuáles son las variables que intervienen. Evidentemente, las salidas serán la velocidad del vehículo, que llamaremos V o `v`, y el ángulo de giro del volante, que llamaremos θ o `theta`. Ahora bien, se deben determinar cuáles serán las variables de entrada, es decir, si se elige un conjunto de variables $\mathbf{x}$ y un conjunto de reglas $\mathbf{r}$, de alguna manera, podrían entenderse las salidas, como $\theta(\mathbf{x}, \mathbf{r})$ y $V(\mathbf{x}, \mathbf{r})$.

La primera, e ingenua de las opciones, sería considerar como entrada los doce sensores, sin embargo esto no es útil, pues teniendo conocimiento experto del dominio del problema, y que este no variará en gran cantidad (siempre será ir marcha atrás y aparcar a la izquierda) podemos fijarnos en los instantes relevantes, qué sensores marcan distancias relevantes. Más adelante se explicará el por qué de la elección, pero se toman como entradas $S_0, S_8, S_9, S_{10}, S_{11}$, y $S_0 - S_{11}$. De esta manera, el controlador tiene el siguiente aspecto.

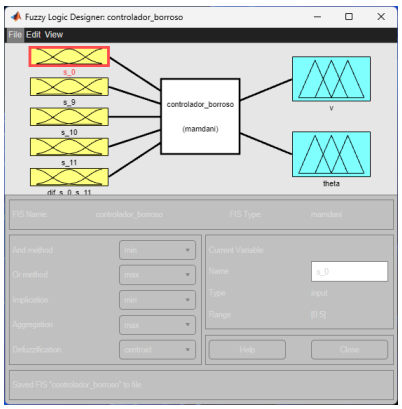

Figura 1.1: Entradas y salidas del controlador

Para todas las variables se mantienen las operaciones por defecto de los conectivos $\wedge, \vee, \rightarrow$, y de la desborrosificación. A continuación se explicarán las funciones de pertenencia de cada una de las variables que intervienen en el controlador. Respecto a las de entrada, se hará una explicación para S_i y $S_0 - S_{11}$, pues las de los S_i son todas iguales.

- `s_0`: como los sensores están limitados a una distancia máxima de cinco metros, se asume que el rango de la variable será $[0, 5]$. Por otro lado, por comodidad a la hora de razonar las reglas, se definen tres conjuntos borrosos, `CERCA`, `CENTRO`, y `LEJOS`. Sus funciones de pertenencia son las siguientes.

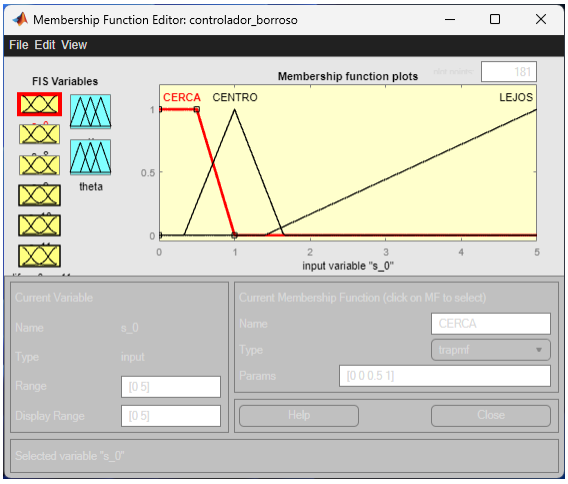

Figura 1.2: Funciones de pertenencia de S_i

Estas funciones de pertenencia se diseñaron pensando en que si la anchura del hueco en el que se encuentra el vehículo es de 4,3 metros, entre 0 y 1 metro, son valores que concuerdan con esta definición. De la misma manera se razonó con el conjunto de `CENTRO` y de `LEJOS`, pero modificando los valores donde más altas son las curvas en base a la experiencia y las pruebas.

- `dif_s_0_s_11`: teniendo en cuenta que $0 \leq S_i \leq 5,\ 0 \leq 9$ es fácil ver que en la expresión $S_0 - S_{11}$, el caso menor es cuando $S_0 = 0$ y $S_{11} = 5$, en cuyo caso la expresión vale -5, y el caso contrario, vale 5, siendo el rango de esta variable $[-5, 5]$. De acuerdo a esto se definen tres conjuntos borrosos, `CERO`, `POSITIVO`, y `NEGATIVO`. Sus funciones de pertenencia son las siguientes.

Figura 1.3: Funciones de pertenencia de $S_0 - S_{11}$

Al igual que en el caso anterior, estas funciones de pertenencia se diseñaron utilizando el sentido común (`POSITIVO` abarca el tramo $[0, 5]$), pero modificando los valores donde más altas son las curvas en base a la experiencia y las pruebas.

- `v`: en las especificaciones del vehículo se nombró que $-6 \leq V \leq 6$, por lo que se decidió tomar el rango $[-6, 6]$ para esta variable. De la misma manera que las funciones de pertenencia anteriores, se diseñaron aplicando el sentido común y realizando cambios en los puntos que son máximas en base a la experiencia. Se eligió definir cinco conjuntos borrosos, que representen velocidades cercanas a cero, y rápidas o lentas, tanto de frente como marcha atrás, siendo estos `CERO`, `POS`, `MPOS`, `NEG`, y `MNEG`.

Figura 1.4: Funciones de pertenencia de V

- `theta`: en las especificaciones del vehículo se nombró que $-90 \leq \theta \leq 90$, sin embargo en este caso el rango no será $[-90, 90]$ si no $[-110, 110]$, de forma que se deje algo de holgura al límite, y en ciertos casos al desborrosificar ciertas reglas en ciertos casos, sea

más fácil obtener valores más cercanos a 90. Obtener en la salida valores superiores a 90 no será ningún problema, pues en dicho caso el coche girará su máximo, 90 grados (de igual forma para el caso de –110). Como en el caso de V se definen cinco conjuntos borrosos según si tenemos el volante recto, o se desea mover poco o mucho el volante a izquierda o derecha, estos son `CERO`, `POS`, `MPOS`, `NEG`, y `MNEG`.

Figura 1.5: Funciones de pertenencia de θ

1.2. Reglas

Una vez se tienen definidas todas las variables, sus conjuntos borrosos, y funciones de pertenencia asociados, se está en condiciones de definir una serie de reglas de forma que cuando se reciban una serie de valores de entrada, el controlador aplique el proceso de inferencia sobre dichas reglas y se desborrosifique la conclusión obtenida. Para ello, aunque el control borroso debe pensarse como condiciones que se cumplen en ciertos estados, es de ayuda pensar primero en la maniobra de estacionamiento de manera algorítmica:

1. Acelerar marcha atrás con volante recto hasta llegar al hueco
2. Girar todo el volante a la izquierda dando marcha atrás hasta tener el coche recto
3. Girar el volante a la derecha hasta dejar las ruedas rectas y seguir dando marcha atrás
4. Parar al llegar cerca de la pared

Esta fue la primera aproximación en la que se pensó, pero no fue difícil ver que al girar desde la posición horizontal original de partida era un tanto arriesgado, dejando muy poco espacio al girar, por lo que pareció razonable seguir el siguiente proceso.

1. Acelerar marcha atrás girando hacia la izquierda ligeramente hasta estar cerca de la pared

2. Al estar cerca de la pared, girar el volante a la derecha hasta tener el coche recto, y entonces dejar el volante recto, todo mientras se da marcha atrás

3. Seguir marcha atrás hasta llegar al hueco

4. Girar todo el volante a la izquierda dando marcha atrás hasta tener el coche recto

5. Girar el volante a la derecha hasta dejar las ruedas rectas y seguir dando marcha atrás

6. Parar al llegar cerca de la pared

En base al paso **1** se decidió que se necesitaban los sensores 0 y 11 para determinar la cercanía del coche a la pared, y su diferencia, pues si $S_0 < S_{11}$ entonces el frontal del coche estaba más cerca de la pared que la parte trasera, y al contrario. Por otro lado, se decidió usar S_{10} para detectar cuándo el coche estaba a punto de llegar al hueco de aparcamiento. Finalmente, S_8 y S_9 fueron utilizados para determinar cuando parar para no chocar con la pared, y además S_9 también intervino en el giro del coche como ahora se observará.

En base al último proceso algorítmico mostrado, se pudieron deducir las reglas empleadas en el controlador, que a grandes rasgos son las siguientes:

$$
\begin{array}{lll}
R_1: & \neg\texttt{CERCA}(S_0) \wedge \neg\texttt{CERCA}(S_{11}) & \longrightarrow \ \texttt{MNEG}(V) \wedge \texttt{POS}(\theta) \\
R_2: & \texttt{POS}(S_0 - S_{11}) \wedge \texttt{CERCA}(S_{11}) & \longrightarrow \ \texttt{NEG}(V) \wedge \texttt{MNEG}(\theta) \\
R_3: & \texttt{CERO}(S_0 - S_{11}) \wedge \texttt{CERCA}(S_{10}) & \longrightarrow \ \texttt{MNEG}(V) \wedge \texttt{CERO}(\theta) \\
R_4: & \texttt{LEJOS}(S_9) \wedge \texttt{LEJOS}(S_{10}) & \longrightarrow \ \texttt{MNEG}(V) \wedge \texttt{MPOS}(\theta)
\end{array}
$$

Véase el razonamiento de cada una. R_1 dice que si el coche no está pegado a la pared, debe dar marcha atrás y girar a la derecha para pegar la parte trasera del coche a la pared. R_2 dice que cuando la parte trasera del coche está pegada a la pared y la delantera no, rápidamente corrija la trayectoria en sentido contrario para el coche paralelo a la pared, colocándose para el aparcamiento. R_3 indica que si el coche está paralelo a la pared, y cerca de esta, simplemente debe seguir marcha atrás con el volante recto. Finalmente R_4 indica que cuando S_9 y S_{10} marcan valores muy altos, es el momento de hacer el giro a la izquierda para meter el coche en la plaza de aparcamiento. Una vez finaliza el giro, la situación es exactamente la misma que cuando comenzaba la ejecución.

Sin embargo, este sería un razonamiento simplemente en base al conocimiento del problema, sin pensar si en otros momentos podría ser que se verificasen reglas cuando no era la intención, o si hay variables que "fastidien" la regla. Por ello, una vez llevadas al editor de MATLAB, con ayuda de la herramienta "*View rules*" se pueden ir introduciendo valores del simulador y ver cuál es la salida del controlador, y a qué reglas y conjuntos se debe dicha salida.

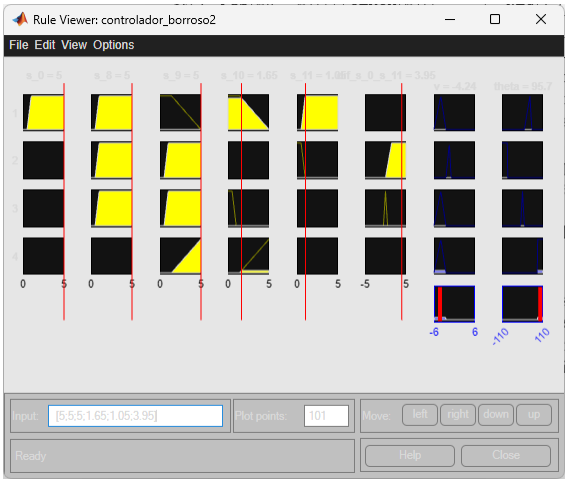

Figura 1.6: Herramienta *View rules*

Si se analizan las reglas introduciendo valores obtenidos durante la ejecución en el simulador, se llega a la conclusión de que no se cumple la ejecución esperada pues algunas reglas actúan "por accidente" cuando no tocan. Esto se debe a que se ha pensado el aparcamiento de forma algorítmica en vez de en estados, pero se puede solucionar modificando cada regla, de forma que represeten estados disjuntos, añadiéndoles los elementos negados del par que se activan. Es decir, si R_i es de la forma $(p \wedge q) \rightarrow t$, y R_j de la forma $(u \wedge v) \rightarrow r$, y cuando se pretendía que se activase sólo R_j, también se activa R_i, esta se puede reescribir de la forma $(p \wedge q \wedge \neg u \wedge \neg v) \rightarrow t$. De esta forma siempre que se verifican todas las condiciones de R_j no lo hacen todas las de R_i. Detectando esta serie de "momentos no disjuntos" entre las cuatro reglas explicadas previamente con ayuda de *View Rules* (se puede detectar esto cuando se quiere que una determinada salida sea la única en tener un "trapecio azul" pero hay más de uno en dicha columna), se reescriben y se obtienen las siguientes:

$$
\begin{array}{lll}
R_1: & (\neg\texttt{CERCA}(S_0) \wedge \neg\texttt{LEJOS}(S_9) \wedge \neg\texttt{LEJOS}(S_{10}) \wedge \neg\texttt{CERCA}(S_{11})) & \longrightarrow \ \texttt{MNEG}(V) \wedge \texttt{POS}(\theta) \\
R_2: & (\texttt{POS}(S_0 - S_{11}) \wedge \texttt{CERCA}(S_9) \wedge \texttt{CERCA}(S_{11})) & \longrightarrow \ \texttt{NEG}(V) \wedge \texttt{MNEG}(\theta) \\
R_3: & (\texttt{CERO}(S_0 - S_{11}) \wedge \neg\texttt{CERCA}(S_9) \wedge \texttt{CERCA}(S_{10})) & \longrightarrow \ \texttt{MNEG}(V) \wedge \texttt{CERO}(\theta) \\
R_4: & (\texttt{LEJOS}(S_9) \wedge \texttt{LEJOS}(S_{10})) & \longrightarrow \ \texttt{MNEG}(V) \wedge \texttt{MPOS}(\theta)
\end{array}
$$

De esta manera en la Figura 1.7 vemos estas reglas transcritas al editor de MATLAB, y una vez hecho todo esto, hacemos las conexiones pertinentes en Simulink y el resultado es el observado en la Figura 1.8. Además, en la Figura 1.9 se muestra la serie de fases que sigue el vehículo al realizar la maniobra de estacionamiento.

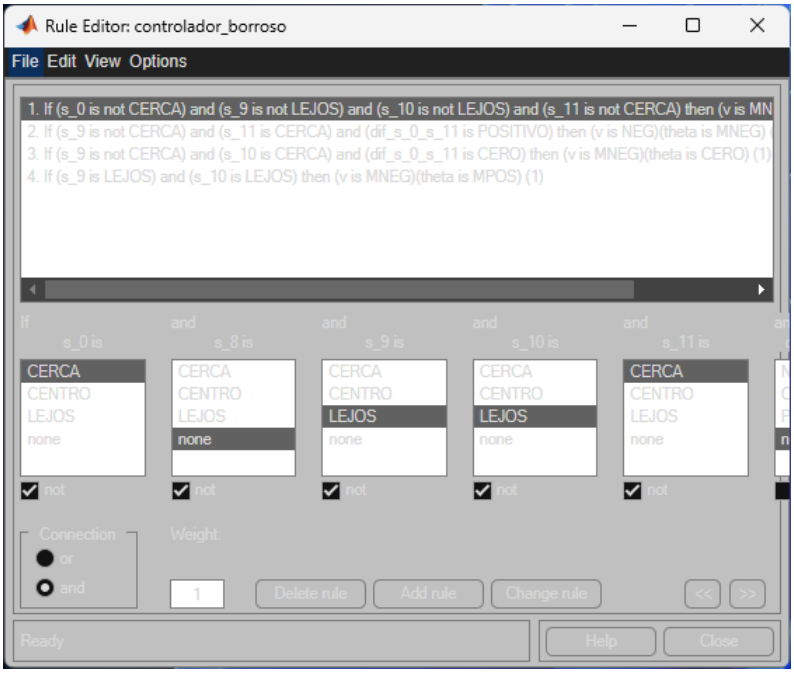

Figura 1.7: Reglas en MATLAB

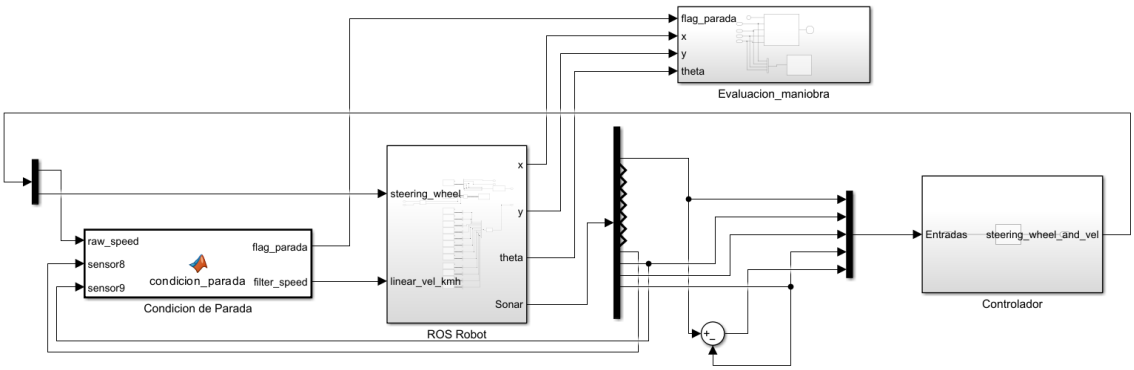

Figura 1.8: Diagrama de Simulink

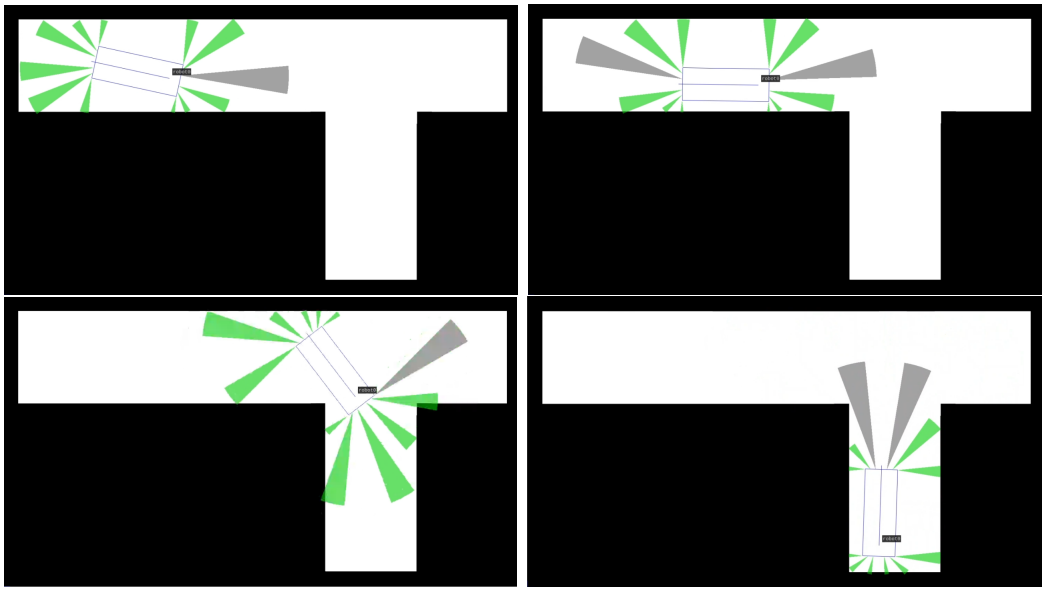

Figura 1.9: Aparcamiento con controlador borroso

Parte 2

Controlador neuronal

En la continuación de esta práctica, el objetivo será el mismo que en la parte anterior pero haciendo uso de redes neuronales, para lo que es necesario generar datos para el entrenamiento de la red. Estos datos se pueden extraer de diferentes formas, como el telecontrol, la ejecución de un recorrido predefinido o el uso de un controlador borroso. En este contexto, se ha optado por emplear el controlador borroso desarrollado en la parte 1 de la práctica.

2.1. Obtención de datos de entrenamiento

Al tener implementado el controlador borroso, se procede a realizar una simulación utilizando el modelo proporcionado en el archivo `ackerman_ROS_neural_controller_capture.slx`. Durante la simulación, se registran los valores de los sensores sonar y las velocidades lineal y angular. Se extraen los valores de los sensores sonar y las velocidades lineal y angular de las señales generadas durante la simulación. De manera similar a la anterior parte, se trabaja con los sensores S_0, S_8, S_9, S_{10} y S_{11}. Se forman las matrices de entrada (inputs) y salida (outputs) necesarias para el entrenamiento de la red neuronal. Los datos generados se guardan en un archivo `.mat` para su posterior uso. Se repite este proceso varias veces para tener un buen conjunto de datos de entrenamiento. Esto se refleja en el archivo `capturar_datos.m`.

```
clear all;
clc;

% Simulación
sim("ackerman_ROS_neural_controller_capture.slx");

% Recoger los datos
sonar_0_values = sonar_0.signals.values;
sonar_8_values = sonar_8.signals.values;
sonar_9_values = sonar_9.signals.values;
sonar_10_values = sonar_10.signals.values;
sonar_11_values = sonar_11.signals.values;
velocidad_lineal_values = velocidad.signals.values;
velocidad_angular_values = theta.signals.values;

% Matrices para la red
```

```
inputs = [sonar_0_values, sonar_8_values, sonar_9_values, sonar_10_values,
    sonar_11_values];
outputs = [velocidad_lineal_values, velocidad_angular_values];

% Guardar datos
save("datos_15.mat", "inputs", "outputs");
```

Código 2.1: `capturar_datos.m`

Una vez que se tiene dicho conjunto de datos, se deberán juntar los datos de todas las ejecuciones en un único *dataset*, que contiene datos de aparcamientos desde la posición original y la definitiva. Primero, se crean matrices vacías para almacenar los datos consolidados de entrada (`X`) y salida (`Y`). Se carga cada conjunto de datos generado por el script anterior (por ejemplo: `datos_1.mat` a `datos_15.mat`) y se juntan en las matrices `X` e `Y`. Destacamos que, las matrices se convierten al tipo de dato `double` para asegurar la compatibilidad con la red neuronal. Y por último, los datos concatenados se guardan en un archivo llamado `datos.mat` para su uso en el entrenamiento de la red neuronal.

```
X = [];
Y = [];

for i = 1:15
    load(strcat("datos_", num2str(i), ".mat"));
    X = [X; inputs];
    Y = [Y; outputs];
end

X = double(X);
Y = double(Y);

save("datos.mat", "X", "Y");
```

Código 2.2: `juntar_datos.m`

Para realizar esta preparación de los datos de entrenamiento, fue necesario realizar modificaciones en el archivo `ackerman_ROS_neural_controller_capture.slx` mediante la adición de bloques `ToWorkspace` para cada sensor (0, 8, 9, 10, 11) y para la velocidad lineal y angular. Ya que, al incorporar bloques `To Workspace` en Simulink y se ajusta para exportar una señal específica, se tiene que los datos generados durante la simulación se almacenen en las variables correspondientes (`sonar_0`, `sonar_8`, `sonar_9`, `sonar_10`, `sonar_11`, `theta`, `velocidad`) en el entorno de MATLAB. Este enfoque resulta valioso al querer acceder y analizar dichos datos tras la simulación, ofreciendo así una mayor flexibilidad en el procesamiento y análisis de los resultados obtenidos. El modelo queda de la siguiente manera.

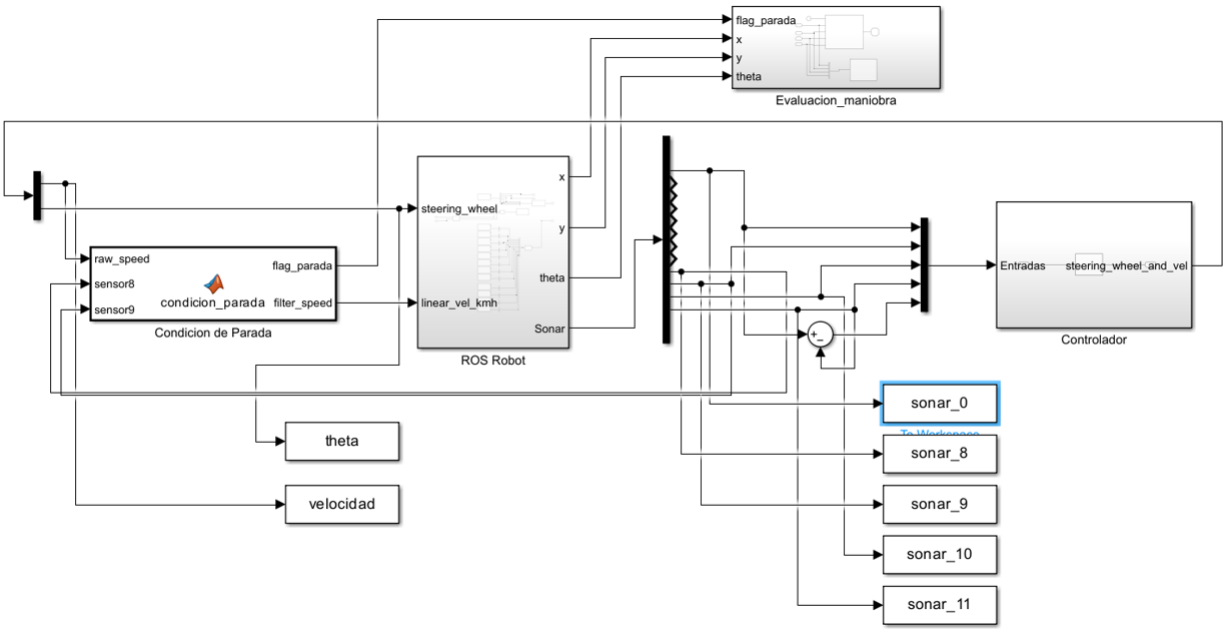

Figura 2.1: Modelo para la captura de datos

Posteriormente, se accedió a la configuración del modelo en `MODELING` → `modeling settings` → `data Import/Export` → `single simulation output` para asegurar la obtención de los valores provenientes de los bloques `ToWorkspace`.

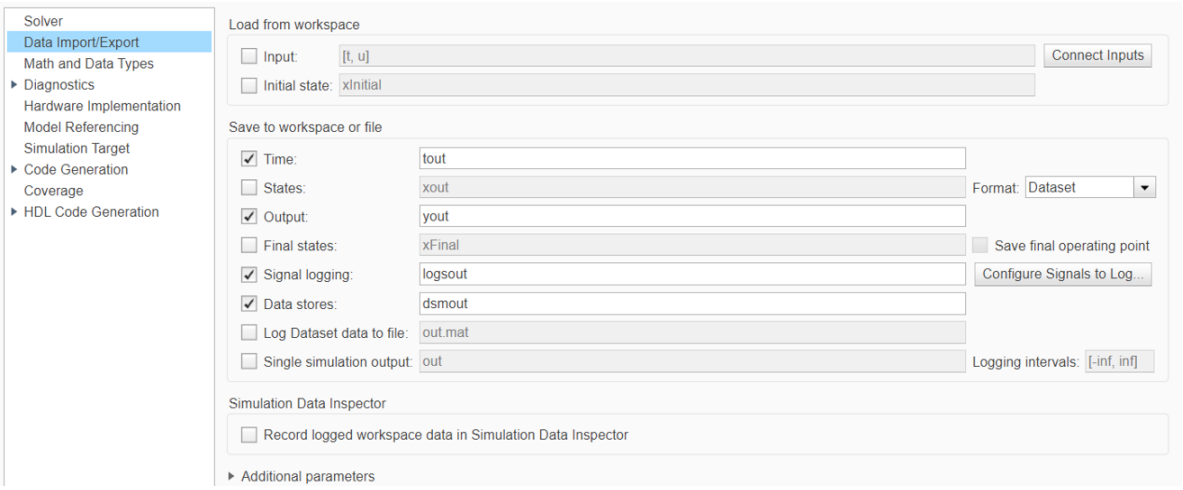

Figura 2.2: Configuración de Simulink

2.2. Creación y entrenamiento de la red

En primer lugar se cargan los datos de entrenamiento previamente almacenados en el archivo `datos.mat`. Estos datos incluyen las matrices de entrada (`X`) y salida (`Y`) que se han

preparado durante el proceso de captura y consolidación de datos. Para crear la estructura de la red, se tienen que elegir el número de neuronas. De acuerdo a la regla que dice,

> "*El número de neuronas de la capa oculta no debe ser mayor al 15 % del número de vectores de entrenamiento.* "

mediante prueba y error, y observando las diferentes gráficas de rendimiento, se optó de acuerdo a los resultados observados, tomar el número de neuronas igual al 1 % del número de vectores de entrenamiento. En general la fórmula seguida es

$$n = \lfloor \texttt{size}(X, 1) \cdot 0{,}01 \rfloor .$$

Posteriormente, se crea una red neuronal alimentada hacia adelante con la cantidad de neuronas calculada en el paso anterior. La red se configura con las matrices de entrada (`X`) y salida (`Y`), y comienza el entrenamiento con el algoritmo de Levenberg–Marquardt. Se guarda la red neuronal entrenada y la información de entrenamiento en un archivo llamado `red.mat`. Esto permite recuperar la red para uso posterior sin necesidad de volver a entrenarla, así como los datos y gráficas del entrenamiento. Por último, se genera el bloque de Simulink con la red entrenada para probar su funcionamiento en el simulador.

```
% Carga de datos de entrenamiento
load("datos.mat");

% Elección del número de neuronas de la capa oculta
neuronas = floor(size(X, 1) * 0.01);

% Creación de la red
net = feedforwardnet(neuronas);
net = configure(net, X', Y');

% Entrenamiento
[net, tr] = train(net, X', Y');

% Guardar en local y Simulink
save("red.mat", "net", "tr");
gensim(net);
```

Código 2.3: `entrenar_red.m`

El modelo de Simulink es bastante similar al primero mostrado durante esta práctica, pero eliminando la diferencia entre S_0 y S_{11}, y cambiando el controlador neuronal por un subsistema que representa el controlador neuronal, que contiene la red neuronal junto con un conversor de tipos.

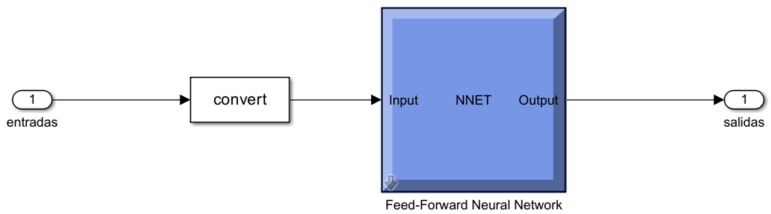

Figura 2.3: Controlador neuronal

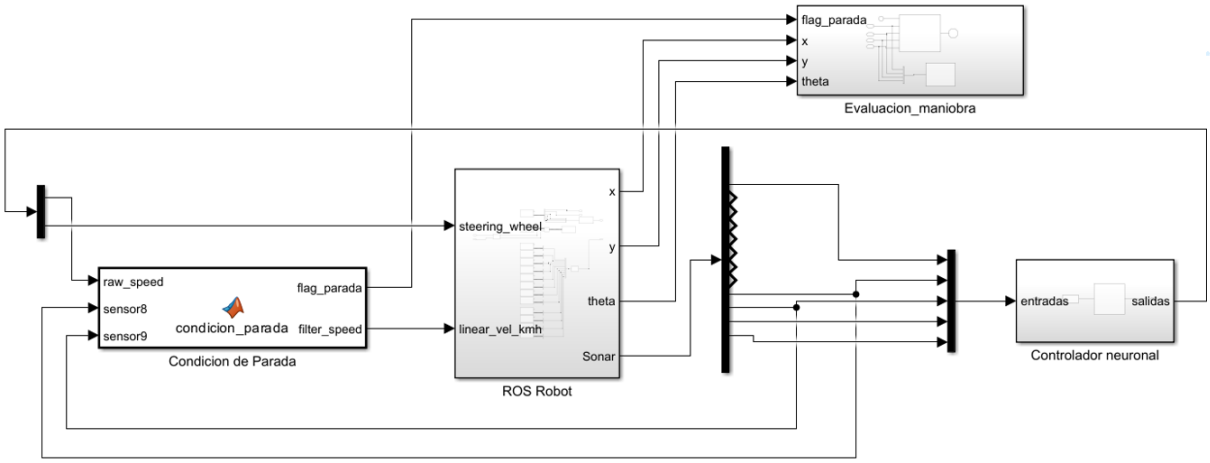

Figura 2.4: Modelo con controlador neuronal

2.3. Ánalisis y resultados

A continuación, se procede a analizar el entrenamiento de la red neuronal, evaluando los resultados obtenidos durante el proceso. Cabe destacar que la red neuronal y los resultados de entrenamiento están almacenados en el archivo `red.mat`, de manera que usando `load('red.mat')` se puede recuperar el análisis de entrenamiento y las gráficas correspondientes.

Si se visualiza la arquitectura de la red, se tiene una capa de entrada con 5 neuronas, una capa oculta con 26 neuronas y una capa de salida con 2 neuronas. También, con `disp(tr)` se puede mostrar la estructura del entrenamiento. En este caso, el entrenamiento se ha realizado con la función `trainlm` y se ha completado ya que se cumplió el criterio de validación. Se realizaron un número de épocas igual a 90.

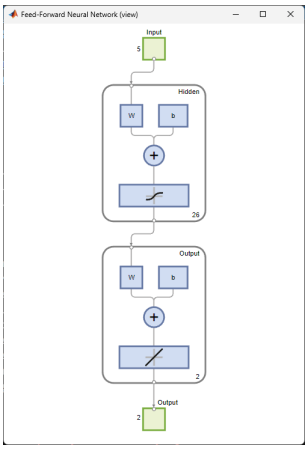

Figura 2.5: Estructura de la red neuronal

Se puede observar, en la figura siguiente, cómo la red neuronal entrena y reduce el error

conforme el tiempo avanza, ajustándose a la salida de los datos de entrenamiento. Se puede concluir que existe cierto error, por lo que el aparcamiento no será perfecto, pero tampoco existen problemas de *overfitting*, pues no existen grandes diferencias entre los errores calculados sobre los diferentes conjuntos.

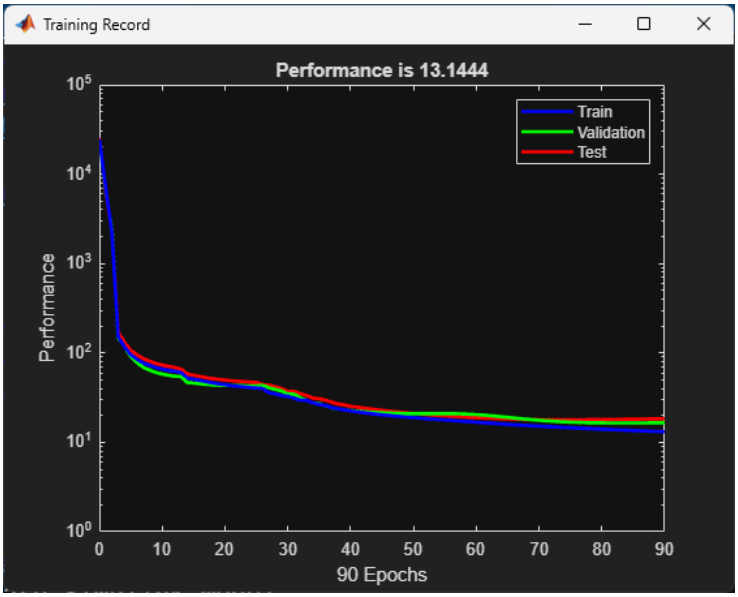

Figura 2.6: Error durante el entrenamiento

Se observa que el modelo se ajusta bastante bien a los datos. Si hay una discrepancia significativa entre las predicciones y los valores reales, se verán claramente en en la siguiente gráfica, pero no es el caso, pues el valor de R se acerca a $0.\bar{9}$.

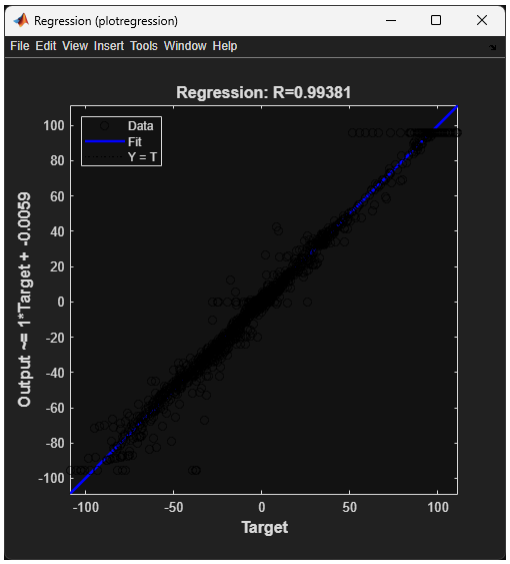

Figura 2.7: Similitud entre y e $\hat{y}$

Como se puede observar en las siguientes figuras, el vehículo es capaz de realizar con

éxito la maniobra de estacionamiento partiendo desde la posición original y la definitiva con el controlador neuronal diseñado.

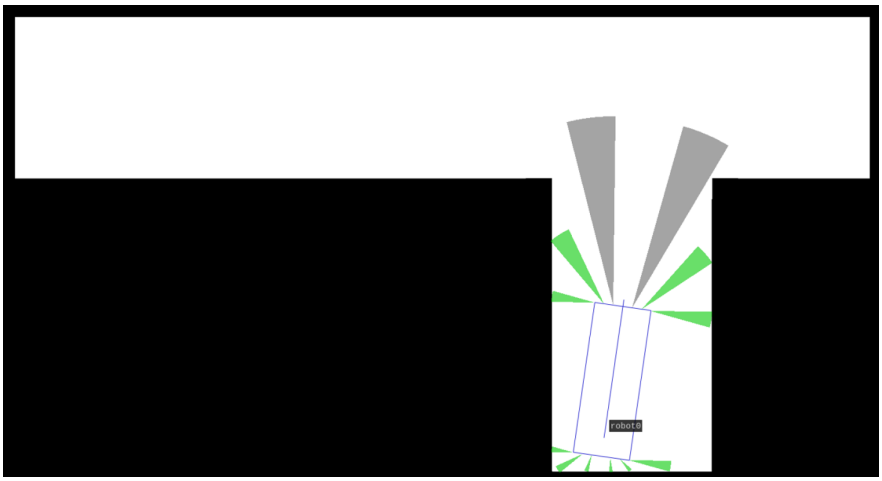

Figura 2.8: Aparcamiento desde la posición original

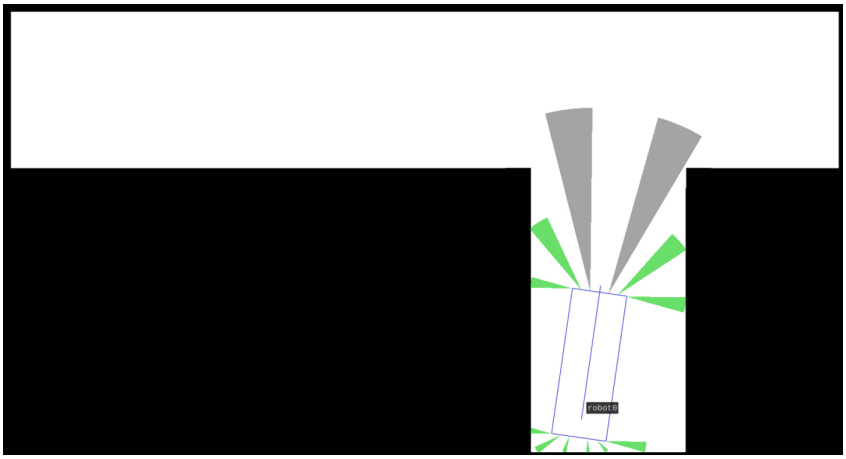

Figura 2.9: Aparcamiento desde la posición definitiva