Escola Tècnica Superior d'Enginyeria ETSE-UV

GRADO EN INGENIERÍA ELECTRÒNICA INDUSTRIAL

TRABAJO DE FIN DE GRADO

TELEOPERACIÓN ASISTIDA DE UN ROBOT MÓVIL, CON CAPACIDAD DE DETECCIÓN Y EVITACIÓN DE OBJETOS MEDIANTE VISIÓN ARTIFICIAL E INTELIGENCIA ARTIFICIAL

AUTOR/A:
EVELINE ALEXANDRA MIHUT

TUTORÍA:
VALERO LAPARRA PÉREZ-MUELAS
VICENT GIRBÉS JUAN

JUNIO, 2020

VNIVERSITAT DE VALÈNCIA | Escola Tècnica Superior d'Enginyeria ETSE-UV

GRADO EN INGENIERÍA ELECTRÒNICA INDUSTRIAL

TRABAJO DE FIN DE GRADO

TELEOPERACIÓN ASISTIDA DE UN ROBOT MÓVIL, CON CAPACIDAD DE DETECCIÓN Y EVITACIÓN DE OBJETOS MEDIANTE VISIÓN ARTIFICIAL E INTELIGENCIA ARTIFICIAL

AUTOR/A:
EVELINE ALEXANDRA MIHUT

TUTORÍA:
VALERO LAPARRA PÉREZ-MUELAS
VICENT GIRBÉS JUAN

TRIBUNAL:

PRESIDENTE/PRESIDENTA:

VOCAL 1:

VOCAL 2:

FECHA DE DEFENSA:

CALIFICACIÓN:

RESUMEN

Este proyecto pone en práctica los conocimientos adquiridos durante estos años en los ámbitos de visión artificial y robótica.

Se trata de un sistema de teleoperación con asistencia a la conducción mediante visión artificial. El sistema es capaz de detectar posibles colisiones con un anticipo mayor que el ofrecido por sensores de distancia o posición. Actúa de forma que si es posible evitar un objeto lo evita y si el objeto es inevitable, frena.

Esto se consigue gracias al uso de ROS como sistema de comunicación entre los distintos componentes que conforman el sistema y gracias a YOLO una red neuronal entrenada para la detección de objetos. El movimiento del robot será controlado con un Joystick que permite su teleoperación.

RESUM

Aquest projecte posa en pràctica els coneixements adquirits durant aquests anys en els àmbits de visió artificial i robòtica.

Es tracta d'un sistema d'assistència a la conducció mitjançant visió artificial, capaç de detectar possibles col·lisions amb una bestreta major que l'ofertada per sensors de distància o posició. Actua de manera que si és possible evitar un objecte l'evita o bé si no és possible evitar-lo, frena.

Això és possible gràcies a l'ús de ROS com a sistema de comunicació entre els diferents components que conformen el sistema i gràcies a YOLO una xarxa neuronal entrenada per a la detecció d'objectes. El moviment del robot es governa amb un Joystick que permet la seva teleoperació .

INDICE

ANEXOS

INDICE DE FIGURAS

1. INTRODUCCIÓN

Hoy en día la tecnología tiene una parte muy importante en la vida cotidiana. Cada vez más, se intenta optimizar y facilitar la labor humana con el fin de ofrecer protección en los trabajos y actividades más peligrosas y al mismo tiempo ofrecer productos y servicios de calidad.

Uno de los sectores en el que actualmente la tecnología está en continuo desarrollo es el sector de la robótica, que trata de ofrecer productos de calidad y seguros.

En base a esto, el objetivo principal de este proyecto es la realización de un sistema de teleoperación con asistencia a la conducción que mediante visión artificial sea capaz de procesar el entorno de un robot móvil y ofrecer una respuesta rápida a cualquier acción/inconveniente que pueda provocar una situación peligrosa para éste.

Se han elegido dos métodos para evitar las posibles colisiones. El primero es evitar el obstáculo actuando sobre la dirección del robot y el segundo es frenar en caso de que el obstáculo no se pueda evitar, ya sea porque hay más de un obstáculo en la trayectoria del robot o porque el obstáculo se encuentra en medio del camino y no se puede decidir qué dirección tomar.

El prototipo que se va a presentar está simulado sobre un robot móvil, pero el algoritmo es aplicable a automóviles, como se mostrará en algunos ejemplos. La elección del realizado en simulación del TFG ha sido necesaria a causa de la pandemia del Covid-19.

1.1 OBJETIVOS ESPECÍFICOS

El proyecto está dividido en 3 partes y cada una de ellas se encarga de un proceso distinto.

El funcionamiento sigue el esquema de la *Figura 1*. Primero se adquieren los datos de la cámara, volante y pedales, los cuales se envían a ser procesados por la red neuronal. La red toma las imágenes y detecta los diferentes objetos que hay en el camino. Sobre estas detecciones decide si se actúa sobre la dirección o velocidad del robot (módulo de procesado) dependiendo de la posibilidad de evitar el objeto detectado y dependiendo de la dirección y velocidad actual del robot. Si el objeto no interfiere en la trayectoria del robot se enviarán los datos leídos directamente al robot. Por otro lado, si el objeto interfiere, se creará un efecto de fuerza sobre el volante girándolo en dirección contraria al objeto siempre y cuando el objeto sea evitable. En caso de que el objeto esté en el camino y no se pueda evitar ya sea porque está en medio o porque hay más de un objeto (objeto tanto en la trayectoria actual del robot como en la del giro), se frenará. Estos datos procesados se envían al controlador y éste se encarga de reproducir en el simulador el comportamiento del robot móvil.

En conclusión, el robot es teledirigido por el usuario siempre y cuando no haya ningún peligro, en el momento que se detecta alguna situación peligrosa será el sistema el que actúe sobre el robot, pero por seguridad ante posibles fallos en la detección, si el usuario interfiere en la actuación del sistema, el control se vuelve a ceder al usuario.

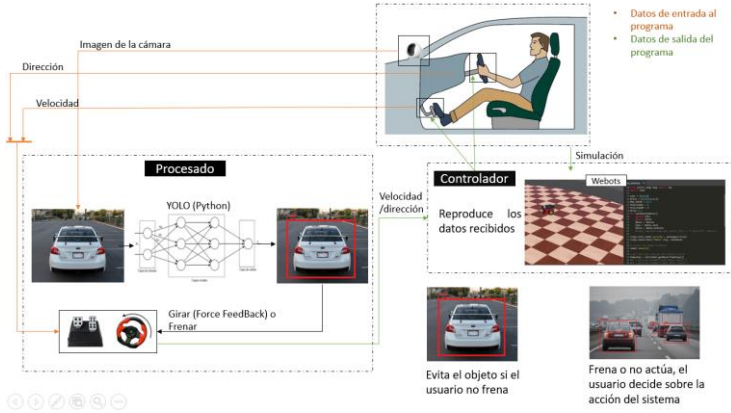

Figura 1, Esquema del funcionamiento: Se reciben los datos del vehículo y se envían al procesado donde la imagen percibida se pasa por la red para detectar la posición del objeto y junto con la velocidad del vehículo se determina la necesidad de frenar o girar. Finalmente se envían al controlador y simulador para reproducir el movimiento procesado.

1.2 ESTRUCTURA LA MEMORIA

En primer lugar, se explica de manera general los conocimientos necesarios para llevar a cabo el proyecto. Se hace una introducción a las redes neuronales, a ROS (Robot Operating System) y a los robots móviles. Al final de la sección se introducen los diferentes entornos y componentes utilizados para el desarrollo del proyecto.

En segundo lugar, se explican los procesos para instalar y configurar los principales elementos utilizados: la plataforma ROS, el simulador Webots y el Joystick.

Finalmente, se explica el código de cada uno de los bloques que componen el proyecto y se exponen los resultados obtenidos.

2. MARCO TEÓRICO

2.1 ESPECIFICACIONES TÉCNICAS

Para la comunicación entre las partes que componen el proyecto, se ha optado por usar ROS, un framework para el desarrollo de software para robots que permite una comunicación sencilla entre diferentes nodos o dispositivos. En este caso se usa la versión Melodic, por ser la única compatible con el sistema operativo Windows.

El procesado de imágenes se hace mediante inteligencia artificial, usando YOLO la versión Tiny, que a pesar de ser una versión que reconoce objetos con más dificultad nos ofrece una gran velocidad de procesado para la máquina en la que se está usando.

Tras el análisis entre distintos robots y simuladores, se ha decidido usar el robot Pioneer 3TX simulado en Webots, para el cual se ha tenido que desarrollar un controlador compatible con ROS que ejecutara las acciones del volante y del procesado de imágenes.

Y finalmente se controlará el movimiento mediante el volante Wingman Formula Force GP, compuesto por dos pedales y un volante con 6 botones.

El sistema está desarrollado en Python por la gran cantidad de librerías y recursos que ofrece.

2.2 INTELIGENCIA ARTIFICIAL Y MACHINE LEARNING

La inteligencia artificial (AI) es una tecnología encargada de "dotar" a las máquinas con capacidad de tomar decisiones a partir del procesado de datos, es decir, producir comportamientos considerados como inteligentes. A partir de un gran conjunto de datos, las máquinas dotadas de AI son capaces de formular predicciones automáticas identificando tendencias y patrones. Por otro lado, también se pueden entrenar para reproducir el comportamiento inteligente a partir de un aprendizaje previo.

Una de las ramas de la inteligencia artificial es el Machine Learning (ML). Hace referencia a un sistema capaz de aprender por sí solo. Es decir, un algoritmo se encarga de revisar millones de datos y a partir de estas revisiones predice comportamientos futuros. El ML puede ser supervisado o no-supervisado. En el aprendizaje supervisado el usuario se encarga de definir las características y etiquetas para cada dato, es decir es el usuario el que entrena el algoritmo. El aprendizaje no-supervisado define las características principales y el algoritmo se encarga de agrupar en función de esas características por similitud todos los datos.

Por otro lado, dentro de la rama de ML está el Deep Leraning, que se encarga de hacer un aprendizaje profundo, es decir, se usa un número de capas más elevado (4, 5 o más). El algoritmo utilizado funciona

por capas, cada capa se encarga de analizar unas características específicas obteniendo como resultado un valor de probabilidad de que los datos de entrada equivalgan al resultado propuesto por el usuario. Para que se obtengan las predicciones finales se necesita un entrenamiento previo del algoritmo.

Con entrenamiento se hace referencia a detectar la combinación de características/parámetros que más se ajuste al valor de salida especificado por el usuario. Por ejemplo, al pasar una gran cantidad de imágenes de sillas por el algoritmo y el usuario haber especificado que se trata de una silla, el algoritmo extraerá todas las características de las imágenes, pero solo se centra en las que más se repitan. A partir de ahí siempre se centrará en esas para que cuando se vuelva a pasar la imagen de una silla, éste compruebe que se cumplen dichas características “aprendidas” en el entrenamiento y pueda deducir que se trata de una silla. Esto sería en el caso del aprendizaje supervisado.

2.3 VISIÓN ARTIFICIAL

La visión artificial es una rama de la inteligencia artificial. Hace referencia a la capacidad de las máquinas de reproducir el "sentido de la visión humana". De esta manera permite crear aplicaciones capaces de procesar imágenes extrayendo todo el tipo de información que se desee.

"Visión es un proceso que produce, a partir de imágenes del mundo exterior, una descripción útil para el observador y no tiene información irrelevante." D.Marr

Para desarrollar una aplicación de visión artificial se suele seguir el esquema de González y Woods según el cual hay tres tipos de niveles de información: bajo, medio y alto; mientras que el conocimiento se ubica en el centro, ya que las imágenes tienen un análisis que necesita de un conocimiento previo por su gran complejidad.

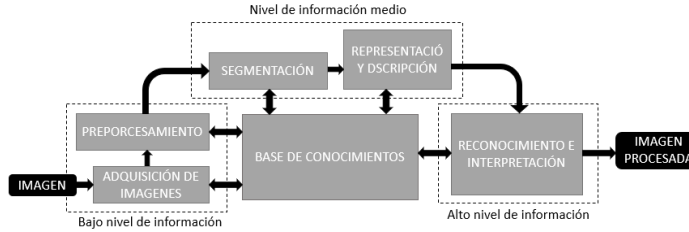

Figura 2, Esquema visión artificial.

En primer lugar, se coge una serie de imágenes "muestra" las cuales se dividen en 2 grupos, un 80% de estas imágenes se usarán para el entrenamiento y un

20% para la validación, que permite comprobar si la red se comporta de manera correcta. Cada imagen del conjunto debe ser clara, con buena iluminación y contraste. Una vez la imagen cumple con los requisitos iniciales, se pasa al pre-procesado, donde se tiene que "editar", es decir, aplicar una serie de procesos para mejorar las características como iluminación, contrastes, filtros… y así facilitar el trabajo de los bloques siguientes, ya que se obtendrán los detalles de los objetos más relevantes.

En el nivel de información medio, se hace la segmentación y extracción de características. En la segmentación se divide la imagen en regiones con significado para posteriormente extraer las características según su textura, contornos, color…
Finalmente, en el nivel alto se reconocen los segmentos asignándoles una etiqueta de alto nivel. Estas etapas se verán posteriormente en la explicación de las redes para este tipo de aplicaciones y YOLO.

Una vez desarrollada la aplicación, las imágenes que se inserten en ella deberán tener una calidad parecida a las de la "muestra" usada en su creación. Todo este proceso se necesita para el aprendizaje de la red neuronal. En el caso de este trabajo, como se explica en el apartado "2.2 Redes neuronales", la red ya venía entrenada, por tanto, no se han tenido que preparar las imágenes para su entrenamiento. Pero sí se han tenido que adaptar antes de pasarlas por el reconocimiento, cambiándoles el tamaño.

2.4 REDES NEURONALES

Una de las estructuras usadas en el Machine Learning son las redes neuronales y dentro de éstas está el Deep Learning. Las redes neuronales son estructuras compuestas por unidades denominadas "neuronas" que están interconectadas entre sí y que trabajan en conjunto para ofrecer a la salida (y) información procesada de un input (x) siempre y cuando haya sido entrenada para ello. El proceso de entrenamiento en modo supervisado, como se ha explicado en el apartado 2.1, consta en modificar los parámetros de la red para que a partir de la información (x) extraiga las características necesarias para asociarla a un valor especificado por el usuario (y).

A continuación, se explica el proceso interno de la red para conseguir el aprendizaje. Las conexiones entre una neurona y otra tiene asociado un peso (w), que representa la intensidad de interacción entre ellas. Estos pesos se asignan de manera aleatoria cuando la red esta desentrenada y se ajustan posteriormente (entrenamiento) en función de la importancia que tiene el elemento de entrada para el procesamiento de la neurona que recibe el input. Tras recibir los pesos e inputs cada neurona realiza una suma ponderada de la información además de sumar un valor *b* denominado "bias" que se encarga de decidir si una neurona necesitara un valor de entrada más alto o más bajo (parecido a un threshold) para generar a la salida un valor normalizado y a esta suma se le aplica una función de activación.

$$f(\sum_{i=0}^{N-1}(\text{xi} \cdot \text{wi}) + b)$$

Las funciones de activación (f(·)) dependen del tipo de red que se esté manejando y tiene como función clasificar los outputs de la neurona para mantenerlos en ciertos rangos, es decir, se encargan de eliminar la linealidad en los casos más complejos de clasificación distorsionando el plano generado por la neurona permitiendo de esta manera el enlace de varias neuronas evitando que a la salida se obtenga una respuesta lineal. Por ejemplo, uniendo varias neuronas que tienen como función de activación una Sigmoide (Figura 3*)* se podría clasificar el conjunto de la Figura 4.

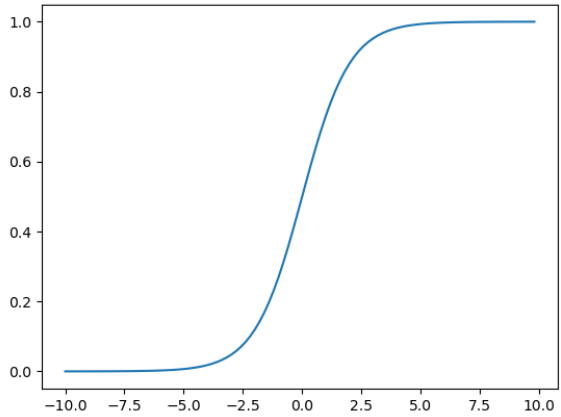

Figura 3, Sigmoide.

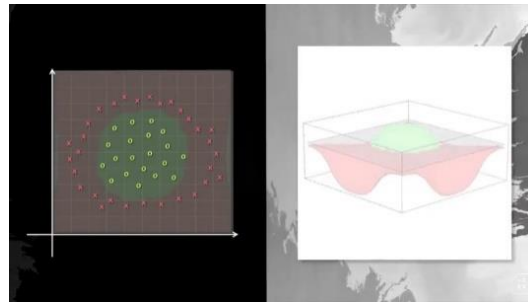

Figura 4, Izquierda: Problema de clasificación no lineal. Derecha: Solución obtenida mediante la combinación de Sigmoides.

Las neuronas se agrupan por capas y la concatenación se hace entre neuronas de distintas capas. La organización de las redes consta de una capa de entrada, las capas ocultas y una capa de salida (Figura 5). Cuantas más capas añadimos más complejo es el conocimiento que elaboramos, aunque más complejo el entrenamiento de la red.

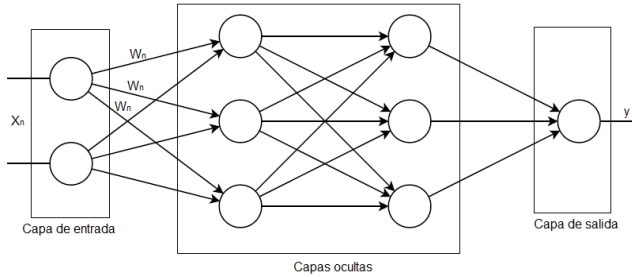

Figura 5, Distribución redes neuronales

2.4.1 ALGORITMO BACKPROPAGATION

El algoritmo se encarga de analizar la responsabilidad de cada neurona en el resultado final cuando éste ha sido erróneo, lo hace de forma recursiva capa tras capa, es decir, de N_i a N_0 tratándose de una red de *i* capas. La aplicación del algoritmo da como resultado una derivada parcial en función del error de salida para cada neurona (la implicación de cada una en el error final). Una vez se tiene estas implicaciones se modifica el peso de las neuronas de manera proporcional a su implicación para reducir la función objetivo.

En conclusión, según la característica analizada por cada neurona y la implicación de cada una, se obtiene un valor final que representa la probabilidad de cada output. Si dicho valor es incorrecto, se modificará más los pesos de las neuronas más implicadas.

2.5 YOLO Y REDES CONVOLUCIOALES

2.5.1 DEFINICIÓN Y CARACERÍSTICAS

Yolo Tiny es una red neuronal entrenada en darknet (es un framework personalizado) y diseñada para reconocer objetos dentro de una imagen en tiempo real. Es un modelo de deep learning compuesto por 16 capas contando la capa de entrada. Al tratarse de una red para procesado de imagen se usan capas convolucionales (9 capas) y capas MaxPooling (6 capas). Por tanto, la red se define como una red neuronal convolucional.

Las redes neuronales convolucionales imitan el comportamiento del sistema visual humano. Trabajan de forma que las primeras capas detectan formas/características simples y a medida que se avanza a través de capas aumenta la complejidad. Están compuestas por:

- Capas convolucionales que cogen conjuntos de pixeles a los cuales se les aplica unos "filtros" convolucionales que son una matriz llamada Kernel (los pesos w). Esto da como resultado imágenes que contienen ciertas características de la imagen original definidas por los filtros. A estas "nuevas imágenes", se les aplica la función de activación.
- Las capas MaxPooling se encargan de reducir el tamaño de las nuevas imágenes. Para ello también se recorre con un Kernel la imagen (por grupo de pixeles) y de los grupos de pixeles se almacena el valor más alto en una nueva imagen.

Yolo utiliza Kernels de 3x3 para las capas convolucionales y 2x2 para las Max-Pooling, es decir cada capa reduce las imágenes a la mitad de su tamaño. La función de activación usada es Leaky Rectified Linear Unit (Leaky ReLU). Se representa mediante la *Ecuación 2*, permite clasificar los valoresde manera no lineal permitiendo generalizar los valores de salida. Si el valor de la suma ponderada realizada por la neurona es mayor a cero (o un valor específico) la función se comporta como una función lineal y como una función lineal con tendencia a cero cuando es menor que cero (o un valor especifico) (Figura 6).

$$f(x) = \begin{cases} 0.01x & para\ x \leq 0 \\ x & para\ x > 0 \end{cases}$$

Ecuación 2

Entre las ventajas que presenta esta función tenemos su velocidad y la falta de errores cuando se aplica Backpropagation.

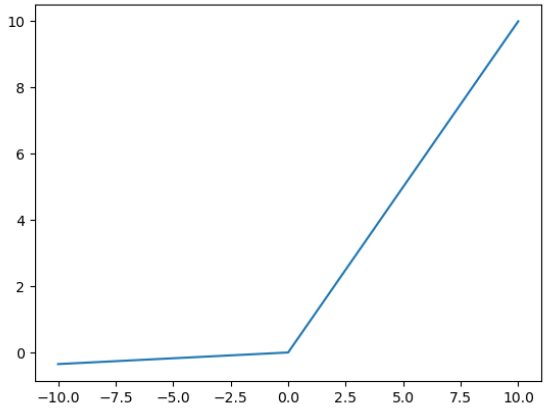

Figura 6, Leaky ReLU

2.5.2 FUNCIONAMIENTO YOLO

Primero se divide la imagen en una cuadrícula de 13x13. Para cada uno de los 169 cuadrados se va a predecir hasta 5 cajas, en función de la probabilidad de que contengan un objeto (cualquiera) y en función de ese valor el margen de las cajas tendrá un grosor u otro. Las cajas de un cuadrado que tengan más probabilidad se juntan con las cajas de su cuadrado adyacente para dar

lugar a una nueva caja de mayor área en la cual la posibilidad de tener un objeto será mayor y por tanto ésta tendrá márgenes más gruesos. Éste proceso se repite con los cuadrados adyacentes hasta dibujar la caja del objeto total. Todas esas cajas finales tienen asociado un porcentaje que define la precisión total de la caja, es decir la seguridad de que la caja contenga algún objeto, unas coordenadas "x", "y" relativas a los bordes de la caja (celda) y unos valores "w", "h" que se predicen en función del tamaño total de la imagen. Una vez se tiene la caja del objeto total, se predice el objeto que contiene. De esta manera se consigue procesar la imagen en una sola pasada a través de la red, ya que como se ve en la Figura 7, con una sola pasada se obtienen todas las detecciones posibles con sus respectivas etiquetas.

Para limitar el número de cajas que hay en la imagen, es decir, dejar sólo las de los objetos completos y no todas las cajas posibles como en la imagen superior de la Figura 7, hay que definir un umbral como mínimo al 30%, es decir, si la caja devuelve una probabilidad inferior a ese valor que no se dibuje, así conseguimos dejar solo los objetos mejor detectados, tal y como se observa en la imagen derecha de la Figura 7. Cada una de las cajas pasa por la red como "una imagen independiente" las cuales atravesarán cada capa de neuronas donde los kernels se encargarán de analizarlas y extraer las características principales y antes de pasar a la siguiente capa se les aplicará la función de activación, esto en las capas convolucionales. Si se trata de las capas MaxPooling se les aplicará otro tipo de kernel que se encarga solo de reducir el tamaño de la "imagen". Y como ya se ha mencionado, a la salida de

la red obtendremos las coordenadas de las cajas, un valor que representa la probabilidad de que el objeto detectado sea correcto y la etiqueta de dicho objeto.

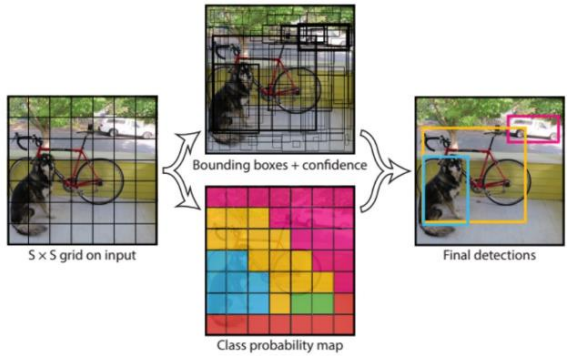

Figura 7, Funcionamiento YOLO

2.6 ROBOT OPERATING SYSTEM

Robot Operating System (ROS) es un framework que permite la escritura de programas para robots. Se usa como sistema operativo básico en comunicación entre procesos, abstracción de hardware, administración de paquetes y control de dispositivos de bajo nivel en todo tipo de robots. Ofrece una serie de herramientas que aumenta las posibilidades de los sistemas ROS, dando lugar a un entorno completo y sencillo para el desarrollo dentro de la robótica. Entre estas herramientas se encuentran:

- rviz: utilizada para visualización 3D

- rosbag: útil para capturar mensajes y grabarlos en archivos ".bag" a través del terminal ROS.
- catkin: es la herramienta usada para la compilación de proyectos ROS (basada en CMake).
- roslaunch: usada para ejecutar desde un solo comando múltiples nodos en máquinas específicas, automatizar procesos de arranque... Está escrito en XML.

Por otro lado, ROS permite la utilización de distintos lenguajes como Python, C++ y Java. Además, permite la integración con varios simuladores y comunicación con otro software como Matlab, Unity, etc.

La estructura básica de organización de ROS son los paquetes (pkg), los cuales contienen los nodos, mensajes, archivos de configuración, las dependencias, es decir, los componentes para la programación y comunicación. Las dependencias son todos los elementos que se van a usar posteriormente en el código, como las librerías, los mensajes o servicios de distintos paquetes, etc.

Algunos componentes que podemos encontrar en ROS son:

- Master: Actúa como un registro, los nodos se registran en el "master" para poder comunicarse posteriormente entre ellos, es decir, permite a los nodos encontrarse unos con otros. También registra los topics y Servicios para cada nodo.
- Nodos: Son el código de control, los encargados de enviar y recibir mensajes/servicios entre ellos

con el fin de llevar a cabo la acción para la cual han sido escritos. Se pueden comportar como "Publisher/s" y "Subscriber/s", "Servicios", hacer procesamientos, comunicación Hardware, gestión de parámetros...

  - Los "Publisher/s" se encargan de enviar mensajes con un topic (nombre o etiqueta) específico asignado por el usuario.
  - Los "Subscriber/s" son los que reciben mensajes de topic asignado.
  - Los "Servicios" se componen de 2 mensajes, uno que hace la petición de información y otro que devuelve una respuesta, de esta manera se consigue optimizar la comunicación entre muchos nodos ya que un nodo actúa como servidor ofreciendo datos mientras que otro nodo actúa como cliente enviando y recibiendo datos mediante los mensajes mencionados y así se consigue una comunicación síncrona.

- Mensajes (msg): Son estructuras de datos que se envían entre nodos, pueden ser mensajes de sensores, actuadores, planificación, controles, etc.
- Topic: Es una etiqueta o nombre único que se asigna a un mensaje determinado, de modo que los topics no se pueden repetir, pero puede haber varios diferentes que definen el mismo tipo de dato.

La comunicación entre Publisher y Subscribers se hace siguiendo el esquema de la Figura 8.

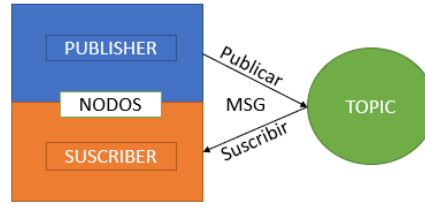

Figura 8, Esquema comunicación ROS caso Publisher/Subscriber.

El Publisher se encarga de publicar en un topic específico la información almacenada en un tipo de mensaje determinado. Mientras que el Subscriber se suscribe a dicho topic para recibir la información contenida en el mensaje. Al tratarse de una arquitectura punto a punto permite que los nodos comuniquen entre sí tanto de manera síncrona como asíncrona.

El protocolo usado en la comunicación se conoce como TCPROS. Éste se basa en el protocolo TCP/IP mediante sockets.

Los servicios tienen datos adicionales, pero al no usarse en este trabajo no se van a explicar.

Para ofrecer una funcionalidad universal, ROS tiene una serie de paquetes ya definidos. Esto le permite trabajar de manera modular, es decir no se necesita hacer todas las partes del proyecto, sino usar el mismo tipo de datos. De esta manera se pueden integrar posteriormente unas partes con otras. Por ejemplo, si tenemos un robot que es compatible con ROS, éste podría estar diseñado para ser controlado con mensajes estándar definidos en ROS

como trajectory_msgs/JointTrajectory.msg (paquete que contiene mensajes para las trayectorias de los robots). Por tanto, por parte del usuario sólo se tendría que usar dicho tipo de mensaje para enviar la información y que el robot la interpretara. Es decir, el robot sería un Subscriber mientras que el usuario tendría que desarrollar solamente un Publisher.

En este proyecto los mensajes usados pertenecen al paquete "sensor_msgs" concretamente usamos "Joy.msg" y "Image.msg" que se componen de los siguientes datos:

Joy.msg: header
float32[] axes: Contiene los valores de todos los ejes que componen un joystick, en este caso un volante y dos pedales)
int32[] buttons: Contiene los valores que toma cada botón del joystick

Image.msg: header
uint32 height: Altura de la imagen
uint32 width: Ancho de la imagen
string encoding: tipo de codificación de la imagen para que el mensaje sea entendido en el receptor
uint8 is_bigendian: determina si el formato en el que se almacenará será con el dato más significativo en primera posición.
uint32 step: longitud en bytes de una fila
uint8[] data: datos de la foto.

Tanto la ejecución de los nodos como la verificación de su funcionamiento, se ha controlado mediante el terminal ROS. Los comandos básicos usados en este proyecto han sido:

- *Roscore* activa el Master.
- *Rosrun <nombre paquete> <nombre nodo>* activa un nodo de un paquete específico.
- *Rostopic list y Rosnode list*, muestra una lista con los topics y nodos (respectivamente) activos.
- *Rostopic echo <topic>*: saca por consola de ROS el mensaje que se escribe en un topic específico.
- *Rosnode ping <nombre nodo>* para comprobar la conectividad del nodo.

2.7 ROBOT MÓVIL

Un robot móvil es un dispositivo programable con capacidad de desplazarse y realizar acciones en distintos ambientes de trabajo. Tienen distintas funciones y usos, pueden ser de uso médico, robots industriales o bien robots móviles.

Según el medio donde el robot trabaje puede ser: terrestre, aéreo, acuático, subterráneo, etc. En este caso se va a usar un robot móvil terrestre o AGV (Autonomous Ground Vehicle). Otros tipos que se pueden encontrar son: UAV (Unmanned Aerial Vehicles) y AUV (Autonomous Underwater Vehicles). Según el tipo de locomoción empleada, en los robots

móviles podemos encontrar: ruedas, patas (robots antropomorfos), híbridos y orugas.

Entre los robots con ruedas o cintas hay algunos modelos de tracción como:

- Diferencial: Mediante dos ruedas controladas de manera independiente y una o varias ruedas auxiliares que funcionan como punto de apoyo.
- Triciclos: Mediante una rueda fija independiente, una orientable independiente y una fija dependiente.
- Ackermann: Mediante dos ruedas tractoras traseras y dos ruedas delanteras para la dirección.
- Omnidireccional: Mediante varias ruedas que pueden moverse en más de una dirección de manera independiente.

Según el ambiente donde opere el robot, independiente de si es exterior o interior, se puede distinguir entre entorno estructurado y entorno no estructurado.

- Los entornos no estructurados son los que contienen objetos dinámicos y/o con movimientos imprevisibles.
- Los entornos estructurados tienen los objetos fijos con características particulares, es decir, se asocian a formas simples y fáciles de reconocer.

En este caso, es un entorno no estructurado y se trata de un robot móvil terrestre con ruedas diferenciales como tipo de locomoción.

2.7.1 SENSORIZACIÓN Y CONTROL:

Para la comunicación e interacción con el entorno los robots utilizan sensores. Los sensores se pueden definir como elementos sensibles a magnitudes físicas como temperatura, velocidad, luz y distancia, entre otras. Por tanto, un robot podrá localizarse, detectar obstáculos, hacer mapas, etc. Entre los sensores más utilizados en robótica destacan los siguientes: codificadores angulares para la odometría, LIDAR o láser de telemetría, infrarrojos, ultrasonidos, cámaras 2D/3D, unidad inercial de medida (IMU) que incorpora acelerómetro, giróscopo y magnetómetro, GPS, etc.

Los controladores son los encargados de interpretar las señales y datos recibidos desde los sensores y utilizarlos en la trayectoria del robot de la manera que se desee o bien enviarlos a terceros. El controlador se puede resumir como un sistema de control y programación para el movimiento del robot.

2.7.2 ELECCIÓN DEL ROBOT

Par la elección del robot usado en este proyecto, se ha hecho una comparativa basada en tamaños, velocidad

máxima, compatibilidad con ROS y posibilidad de incluir cámaras de 6 de los robots móviles ofrecidos de manera estándar por Webots, el simulador utilizado en este proyecto que se explicará en la siguiente sección (2.8 Simulador).

Candidatos y sus características principales:

Tabla 1, Comparación robots móviles

Robot	Tamaño	V.Max	Cámara	ROS
E-Puck	Φ 77mm x 55mm	15 cm/s	Si	Si
Pioneer 3-DX	45.5cm x 38.1cm x23.7cm	1.2 m/s	Si	Si
Pioner 3-AT	50.8cm x 27.7cm x 49.7cm	0.7 m/s	Si	Si
Turtle-bot 2 burger	13.8cm x 17.8cm x 19.2cm	0.22m/s	No	Si
Khepera IV	Φ 140mm x 58mm	1m/s	Si	N/S
Altino	98mm x 180mm x 63mm	50cm/s	N/S	N/S

*N/S: no se ha encontrado información.

Finalmente se ha decidido usar Pioneer 3-DX, ya que es el que mejor cumple con los requisitos impuestos.

2.8 SIMULADOR

Un simulador es un software empleado para reproducir comportamientos específicos de manera virtual imitando las condiciones reales. Los simuladores usados en robótica son muy útiles ya que no se necesita el robot físico para comprobar el comportamiento en distintas situaciones. Esto permite evitar situaciones peligrosas tanto para los usuarios como para el robot (roturas).

Entre los simuladores disponibles en el mercado se han elegido 5 que permiten simular robots móviles.

	Lenguajes suportados	**ROS**	**Licencie**	**Windows**
Gazebo	C++	Si	Open source	Si
Webots	C++, Python, Java, Matlab, C	Si	Open source	Si
RoboDk	Python, C++, C, Matlab	No	Closed source	Si
V-Rep	Urbi, Python, Java, C++, C, Matlab	Si	Closed source	Si
OpenHRP	C++, Python, Java	No	Open source	Si
4DV-Sim	FMI/FMU, Matlab	Si	Closed source	No

Tabla 2, Comparación simuladores

De los cinco mencionados sólo dos cumplen todas las necesidades del proyecto, Webots (ver Figura 9) y Gazebo (ver Figura 10). Pero por interfaz gráfica y comodidad de uso se ha elegido Webots.

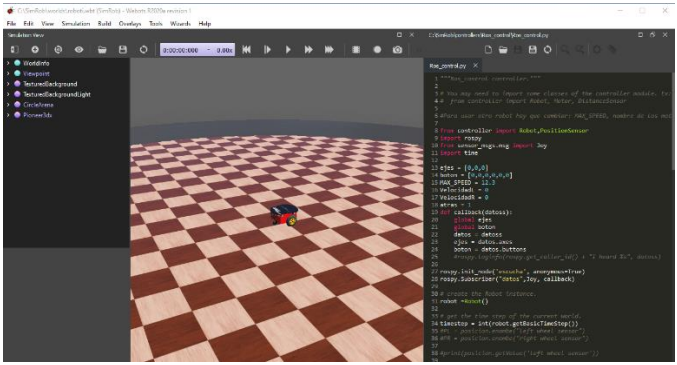

Figura 9, Webots.

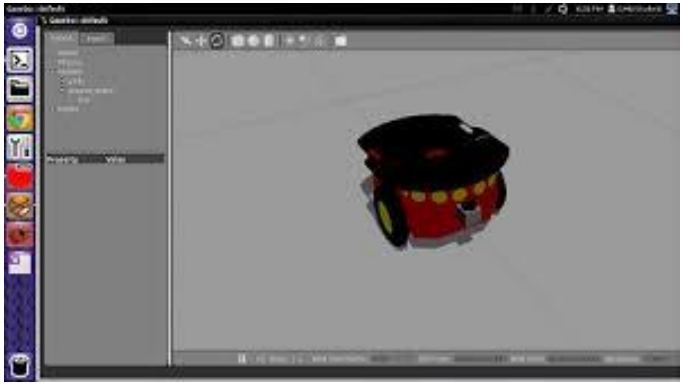

Figura 10, Gazebo.

Webots es un programa de código abierto para la simulación de robots móviles. Está soportado por sistemas operativos como Linux y Windows, entre otros, y permite la integración con ROS.

Ofrece un entorno de desarrollo de prototipos que permite al usuario crear mundos con distintas propiedades físicas, así como agregar objetos pasivos y

activos. Entre los objetos activos tenemos una gran variedad de robots móviles, los cuales tienen integrados desde sensores hasta cámaras y motores. De esta manera el usuario puede crear entornos semejantes a los reales. Además, incluye también una serie de interfaces de robots reales de manera que permite trasmitir el comportamiento simulado a los robots reales.

En cuanto al control de los robots, tiene definidos una serie de controladores estándar para sistemas que trabajan en Linux, pero también se permite la creación de controladores personalizados en distintos lenguajes de programación como Python, C++, C, Matlab…

Por último, otras de las funciones que ofrece es la de grabar y sacar imágenes de la simulación en formatos estándar.

2.9 PROGRAMACIÓN MULTIHILO

En la mayoría de los casos una aplicación necesita tener activadas varias tareas a la vez, esto se conoce como multitarea. Este término se define como el procesamiento en el que un sistema operativo se encarga de administrar distintas tareas/procesos que están en ejecución en el mismo tiempo, las cuales comparten recursos del sistema y compiten entre sí. Para poder realizar la multitarea en programación se usan hilos de ejecución “threads”. En este proyecto se han usado “threads” para poder hacer una adquisición y procesado continuo de la imagen y al mismo tiempo poder actuar sobre la dirección del volante en tiempo real.

2.10 DRIVERS

Los drivers son programas que permiten que los dispositivos periféricos conectados a otros dispositivos funcionen correctamente, es decir otorgan una capacidad de reconocer el tipo y comportamiento del dispositivo conectado. Para este proyecto se ha necesitado la instalación de un driver para el control del volante y pedales, ya que sin éste los valores que se obtenían no estaban calibrados y cambiaban cada vez que se conectaba el periférico. Normalmente los drivers son proporcionados por el fabricante o bien estándar a varios tipos de dispositivos.

3. MARCO PRÁCTICO

En este aparado se explican los procedimientos seguidos para la instalación de algunos requisitos del proyecto como ROS y Webots. También, se explica la configuración necesaria para obtener datos y crear efectos en el Joystick.

3.1 INSTALACIÓN DE ROS

Para la instalación de ROS en Windows en primer lugar se necesita instalar Visual Studio 2019 que será el encargado de compilar/ejecutar las distintas partes del código y también necesitamos Microsoft SDKs. Para la instalación de los diferentes componentes que se usan en los proyectos ROS, se necesitará un administrador de paquetes en éste se recomienda usar Chocolatey. Para instalar Chocolatey se ejecuta como administrador "x64 Native Tools Command Prompt for VS 2019", es decir la consola de Visual Studio 2019 (VS) y se pega el siguiente comando:

```
@"%SystemRoot%\System32\WindowsPowerShell\v1.0\powershell.exe" -NoProfile -InputFormat None -ExecutionPolicy Bypass -Command "iex ((New-Object System.Net.WebClient).DownloadString('https://chocolatey.org/install.ps1'))" && SET "PATH=%PATH%;%ALLUSERSPROFILE%\chocolatey\bin"
```

Esto instalará Chocolatey en C:/opt que es la ruta donde se instalarán todos los componentes de ROS, así como todos los recursos necesarios para hacerlo funcionar.

En otra consola de VS se ha de instalar Git (controlador de versiones):

```
choco upgrade git -y
```

Una vez todos los componentes están instalados, se puede instalar ROS. Para hacerlo se abre otra consola VS (ejecutándola como administrador) y se ejecutará el siguiente comando:

```
mkdir c:\opt\chocolatey

set ChocolateyInstall=c:\opt\chocolatey

choco source add -n=ros-win -s="https://roswin.azurewebsites.net/api/v2" --priority=1

choco upgrade ros-melodic-desktop_full -y --execution-timeout=0
```

Una vez finalizado ya se podrá usar ROS.

Para este proyecto se han tenido que instalar algunas librerías adicionales a las que venían por defecto, como numpy y sdl2. Para ello se ha usado:

```
pip install numpy1.16.6

pip install PySDL2

pip install PySDL2-dll
```

Se tiene que especificar la versión de numpy ya que ROS usa una versión de Python antigua, la 2.7 y la versión de numpy más nueva no está soportada por esa versión de Python.

3.1.1 PROBLEMAS DURANTE LA INSTALACIÓN

Se tenía instalada una versión superior de Python, lo cual daba problemas a la hora de ejecutar los scripts. Para solucionarlo se han tenido que configurar las variables de entorno y añadir a "path" la carpeta donde esta python27amd64, que en este caso es c:/opt, ya que Python se instala junto a la instalación de ROS. Por otro lado, también durante la primera instalación se habían perdido archivos de algunos paquetes por tanto se ha tenido que forzar una actualización y conseguir que se descargara todo. Para forzar la actualización se ha ejecutado el siguiente comando en la consola de VS:

choco upgrade ros-melodic-desktop_full -y --force

ERROR: the following packages/stacks could not have their rosdep keys resolved

to system dependencies:

ros_core: No definition of [rosbag_migration_rule] for OS [windows]

rospack: No definition of [python-catkin-pkg-modules] for OS [windows]

diagnostic_common_diagnostics: No definition of [hddtemp] for OS [windows]

trajectory_msgs: No definition of [rosbag_migration_rule] for OS [windows]

desktop: No definition of [common_tutorials] for OS [windows]

simulators: No definition of [stage_ros] for OS [windows]

joystick_drivers: No definition of [wiimote] for OS [windows]

Continuing to install resolvable dependencies...

executing command [vcpkg install sdl2 --triplet x64-windows]

Computing installation plan...

ERROR: the following rosdeps failed to install

vcpkg: command [vcpkg install sdl2 --triplet x64-windows] failed

vcpkg: Failed to detect successful installation of [sdl2]

Para solucionar estos errores que salieron después de reinstalar ROS, se ha tenido que ejecutar el siguiente comando:

```
rosdep install - -from-paths src - -ignore-src - - rosdistro melodic-y –os=windows:10
```

Además de instalar el pgk inglés de VS y ejecutar el siguiente comando:

```
vcpkg install sdl2 --triplet x64-windows
```

3.2 CREACIÓN DEL WORKSPACE Y PAQUETES ROS

Todos los comandos ejecutan desde la consola de Visual Studio 2019.

Comandos usados para crear un entorno de trabajo (ws) Catkin para ROS en Windows:

C:\Windows\System32>roscd

C:\opt\ros\melodic\x64>cd..

C:\opt\ros\melodic>mkdir catin_ws\src

C:\opt\ros\melodic>cd catin_ws

C:\opt\ros\melodic\catin_ws>catkin_make

Comandos para crear un paquete ROS en Windows:

C:\Windows\System32>roscd

C:\opt\ros\melodic\x64>cd..

C:\opt\ros\melodic>cd catin_ws/src

C:\opt\ros\melodic\catin_ws\src>catkin_create_pkg paquete std_msgs rospy roscpp sensor_msgs

C:\opt\ros\melodic\catin_ws\src>cd..

C:\opt\ros\melodic\catin_ws>catkin_make

C:\opt\ros\melodic\catin_ws>cd devel

C:\opt\ros\melodic\catin_ws\devel>setup.bat

Para comprobar que se ha creado el paquete correctamente se ejecuta el siguiente comando que devolverá las dependencias con las que se ha creado el paquete si éste es correcto:

```
C:\opt\ros\melodic\catin_ws\devel> rospack depends1 paquete
```

Si apareciera el siguiente fallo:

```
CMake Error at paquete/CMakeLists.txt:110:

  Parse error.  Expected a command name, got right paren with text ")".
```

Se debe revisar el CMakeLists.txt y descontentar:

catkin_package(...)

Finalmente se añade la ruta del nuevo paquete a las variables de entrono de Windows ejecutando el siguiente comando:

```
C:\> setx path "ROS_PACKAGE_PATH" C:\opt\ros\melodic\x64\share;C:\opt\ros\melodic\catkin_ws\src
```

3.3 INSTALACIÓN DE WEBOTS E INTEGRACIÓN CON ROS

Para la instalación de Webots en Windows simplemente se tiene que descargar desde la página oficial. Para poder usarlo con ROS se tiene que instalar en la ruta "C:\opt\ros\melodic\x64\share" el paquete webots_ros, para ello se ejecuta en la consola de Windows el siguiente comando:

```
cd C:\opt\ros\melodic\x64\share
```

```
git clone -b melodic
https://github.com/cyberbotics/webots_ros.git
```

Finalmente añadimos la siguiente ruta a las variables de entorno del sistema y así acceder a Webots desde la consola de Windows. Para ello ejecutamos el siguiente comando:

```
Set path=C:\Program Files\Webots\msys64\ mingw64
\bin
```

Como se ha mencionado anteriormente, esto nos perite ejecutar desde la consola el comando "webots" que tiene entre otras la función "--mode=<mode>" que permite abrir Webots en pausa, tiempo real, ejecutando la simulación o modo rápido y la función "--fullscrean" que ejecuta a pantalla completa la simulación.

3.3.1 PROBLEMAS/LIMITACIONES

Uno de los problemas que se han encontrado a la hora de usar el simulador es que se bloqueaba muchas veces cuando se iniciaba o bien aparecía la pantalla en blanco, para solucionarlo se ha reiniciado el controlador gráfico. Otro problema es que Webots en Windows tiene ciertas limitaciones, como por ejemplo los controladores. Para trabajar con Webots y ROS en Windows no hay controladores estándar, sino que se tienen que hacer uno personalizado con su propio archivo de configuración.

3.4 CONFIGURACIÓN DEL JOYSTICK

Para el uso del joystick Wingman GT force se ha tenido que instalar el driver adecuado, ya que con los drivers

estándares de Windows, cada vez que se conectaba el dispositivo aparecían máximos y mínimos distintos para el volante y los pedales. Una vez instalado el driver, se ha podido calibrar y configurar el dispositivo de forma que permite que los valores obtenidos fueran los mismos a pesar de conectar y desconectar el periférico, los máximos y mínimos oscilan entre [-32770, 32770] tanto para el volante como los pedales. En la Figura 12 se puede ver la pantalla usada para la calibración.

Por último, ha permitido la configuración de force feedback (Figura 11), ya que por defecto cuando conectabas el dispositivo, la opción de "centrado" estaba activada lo que impedía realizar la realimentación de fuerza sobre el volante haciendo que tras cualquier movimiento el volante volviera al centro.

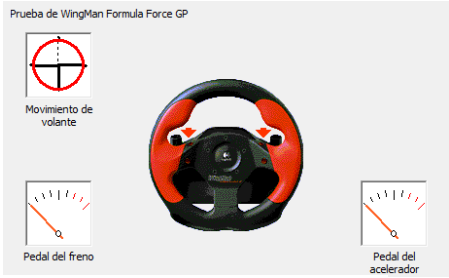

Figura 11, Driver Joystick

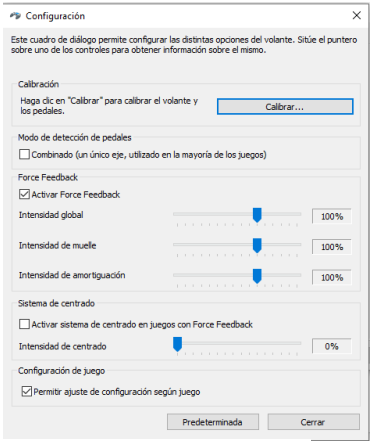

Figura 12, Driver Joystick

4. DESARROLLO

4.1 INTRODUCIÓN

A continuación, se describe el comportamiento del robot móvil en las diferentes situaciones peligrosas para las cuales ha sido diseñado.

Si se detecta un objeto a la derecha o izquierda de la trayectoria del robot, éste intentará evitarlo siempre y cuando 2s después de la detección, el usuario no esté frenando y el objeto no haya desaparecido de la imagen. Este tiempo (2s), se puede disminuir hasta un 60%, pero se ha dejado en dos segundos a causa de problemas de velocidad del ordenador a la hora de detectar objetos y simular. Por lo contrario, si no se está frenando, pero el objeto sigue en la trayectoria del robot, se aplicará una realimentación de fuerza sobre el volante para girar al lado contrario al objeto (el usuario podrá controlar el volante manualmente si aplica fuerza a pesar de la realimentación).

Para evitar posibles colisiones al hacer el giro por realimentación de fuerza, se ha añadido el siguiente comportamiento. Si en la trayectoria hay más de un objeto, el robot frena, ya que no es capaz de calcular la distancia mínima entre los objetos para determinar si colisiona o no. El usuario puede desactivar este comportamiento desde un botón integrado en el volante y tomar el control total del móvil para pasar entre los objetos si considera que es posible. El paro por objeto

inevitable se volverá a activar cuando el peligro haya pasado y el usuario vuelva a apretar el botón.

4.2 EXPLICACIÓN DEL CÓDIGO

El proyecto está dividido en 3 partes, la parte de procesamiento, controlador y datos cámara. Sigue el siguiente esquema:

Figura 13, Esquema del código.

Las librerías usadas a lo largo del proyecto son:

- **Rospy**: librería utilizada para funcionalidades de ROS en Python.
- **Cv2(OpenCv)**: usada para la manipulación de imágenes.
- **Cv_bridge**: permite convertir imágenes de OpenCv a una imagen compatible con los mensajes ROS.
- **Threading**: contienen funciones para el uso de hilos.
- **Sdl2**: librería elegida para el manejo de las funciones del joystick.
- **Time**: funciones de tiempo.
- **Sys**: funciones para el control de la ventana de símbolos de sistema.

- **Numpy**: librería que aumenta la capacidad de uso de vectores y matrices.

También se ha optado por utilizar variables globales, ya que son variables que se necesitan en varias funciones a lo largo del proyecto y de esta manera su manipulación es más simple y efectiva. Se han declarado e inicializado al principio del código de cada bloque.

4.2.1 DATOS CÁMARA

El primer bloque, "Datos Cámara" (Anexo I) es el encargado de recoger las imágenes de la webcam (línea 7) y enviarlas mediante ROS a través de mensajes tipo "Image" al nodo de "Procesamiento". Este bloque equivale a un nodo Publisher llamado "camera" que publica en el topic "camera" un mensaje de tipo "Image" (líneas 8-9) perteneciente al paquete "sensor_msgs". Hay que recordar que los datos de este tipo de mensajes se han explicado en el apartado "2.6".

Está compuesto por una función "camera()" en la que en primer lugar tenemos la declaración de variables y configuraciones que se van a usar. Se declara el nodo (línea 9), una variable "pub" de tipo *rospy.Publisher* encargada de publicar en el topic "camera", una variable "cam" que contiene los datos de la webcam (en este caso la 0 ya que solo teneos una webcam) y finalmente *rospy.rate(15)* que se encarga de marcar la

frecuencia a la que se ejecutará el bucle (en este caso 15Hz).

En segundo lugar, tenemos un bucle "while()" que se mantiene activo mientras el nodo esté disponible, una vez lanzado dejará de estar disponible si el "master" se apaga o bien si se fuerza la salida con ctrl+C. Dentro del bucle se va a leer la imagen de la cámara y se va a transformar a un formato compatible con el mensaje ROS, es decir a una variable de tipo Image() y con codificación bgr8 gracias a Cv_bridge.

Los mensajes ROS tienen una cabecera la cual se tiene que rellenar con el tiempo en el cual se ha enviado el mensaje y la secuencia del mensaje (línea 17-20). Finalmente se publicará la variable "*imamsg*". También hay una sentencia "*rospy.loginfo(imamsg)"* comentada. Ésta se usa para ver por consola el mensaje que se está publicando, también podría usarse el comando "*rostopic echo camera"* pero de la primera forma se escribe también en un fichero.

4.2.2 PROCESAMIENTO

La parte de procesamiento está compuesta principalmente por dos nodos ROS y la red neuronal (Anexo II).

La función "*configuración()*" (línea 26) se encarga de inicializar los diferentes componentes que se van a utilizar:

- El joystick y todas sus dependencias como los efectos Haptic que permiten el force-feedback (línea 36-44) y los eventos que permitirán la recepción de datos en tiempo real. También se declaran variables específicas que contienen datos necesarios para el uso del joystick.
- Para el uso de YOLO, se carga una constante con todas las etiquetas (clases) de los objetos detectables, y una constante "*net*" con la configuración y pesos del modelo pre-entrenado. Además, se asigna un color aleatorio a cada clase, este color es el que se usará al dibujar el cuadro alrededor del objeto (líneas 50-58).

La función *get_output_layers(net)* (línea 60). Se le pasa una variable "net" que contiene los datos de la red neuronal y se encarga de obtener los valores de salida de la red.

La función *draw_prediction()* (línea 71), se encarga de dibujar el rectángulo que enmarca la figura detectada (línea 94) la cual tendrá un color específico según la asignación hecha en la función configuración y también de insertar la etiqueta identificativa del objeto (líneas 96-97). Además, detecta si el objeto está a la izquierda o a la derecha del robot. Para determinar la posición del rectángulo se ha dividido la imagen por la mitad, como tenemos una imagen de 620x480, 480/2=240 las dimensiones de la imagen y un rectángulo predictivo serían las siguientes:

- Si el punto final del rectángulo “x_plus_w” es menor de la mitad, el objeto está a la izquierda (líneas 79-80).
- Si el punto de inicio “x” es mayor que la mitad el rectángulo, el objeto está en el lado derecho (líneas 82-83).
- Si hay más tamaño en la parte derecha “240-x” que en la parte izquierda “x_plus_w-240”, el objeto está a la derecha y viceversa (líneas 85-86 y 88-89).
- Si el objeto esta exactamente en el centro, se girará hacia la izquierda para evitarlo (líneas 91-92).

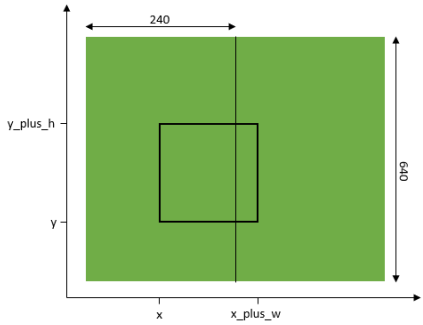

Figura 14, Dimensiones imagen

La función *descubrir()* (línea 100), determina si la dirección del volante es la misma que el objeto

detectado para decidir si se debe actuar o no sobre el volante.

Posiciones del volante:

Figura 15, Posiciones volante.

La variable "posact" guarda la posición (izquierda/derecha) del volante y la variable "fre" el valor del freno. En función de los valores de dichas variables se pueden dar las siguientes condiciones:

- Si el valor de "posact" es menor que cero, es decir el volante está girando hacia la izquierda y el valor de la variable "ya" es izquierda (líneas 109-116), el volante apunta a la dirección del objeto y en consecuencia el móvil se dirige hacia el objeto. Antes de crear el efecto de fuerza, el proceso se espera 2s para comprobar si el usuario no está frenando (se ha dado cuenta del peligro y va a evitarlo) si no es así, la función devuelve los datos "[-1, posact]", la dirección hacia donde se debe girar (derecha) y posición actual del volante.
- Si la posición actual del volante es mayor que cero significa que su dirección es la derecha

(líneas 118-125), por tanto, al igual que en el caso anterior, si el objeto está a la derecha y en 2s el usuario no frena o bien el objeto no deja de interferir en el camino, se creará un efecto de fuerza hacia la derecha "return (1,posact)".

En resumen, si la dirección del volante es la misma dirección que la del objeto de la imagen, primero se espera 2s y si el objeto sigue en la trayectoria del robot, se actúa sobre la dirección para evitarlo. Los 2s son para asegurar que el objeto no se está moviendo.

La función *callback()* (línea 129), es la encargada de recibir la imagen del nodo camera y decodificarla para tener una imagen en el formato utilizado en la red. La variable "si" se usa para determinar si se ha recibido alguna imagen. Se usará posteriormente para activar los hilos, ya que dependen unos de otros y no se puede procesar sin que el nodo cámara esté funcionando.

Las funciones explicadas anteriormente se ejecutan bajo llamadas específicas que se hacen dentro de los tres hilos que contiene "Procesamiento". En primer lugar, se ejecutará el hilo principal (líneas 295-359), es el programa *"main"* que se encarga de lanzar el resto de hilos y publicar los datos del nodo. En segundo lugar, se lanzará *"hilo_camara"* (hilo1, línea 314) siempre y cuando se haya recibido la primera imagen, es decir, la variable "si" sea "1". Finalmente, 4 segundos después (aproximadamente el tiempo que le toma al hilo1 en procesar una imagen y determinar la posición del objeto en la imagen, además de un tiempo margen de 1,5s) de lanzarse *"hilo_camara"*, se lanzará el *"hilo_volante"* (hilo2, línea 316).Se puede reducir el tiempo de 4

segundos a 2s si el usuario al simular lo ve viable. Este orden se debe a la dependencia de datos que hay entre unos con otros. No se puede procesar si no hay imagen y no se puede actuar sobre el volante si no hay imagen o bien no se sabe dónde están los objetos. Una vez ejecutados los hilos en el orden mencionado, estos seguirán ejecutándose de manera concurrente.

El *"hilo_camara"* (línea 136), es el encargado del procesado de la imagen, determinar la dirección en la cual se encuentra el objeto y hacia donde debe girar el robot si tuviera que hacerlo. En primer lugar, se recogen los datos sobre las dimensiones de la imagen (líneas 149-152), estos datos se usarán posteriormente para dibujar los rectángulos predictivos. Para pasar la imagen por la red se necesita crear un "blob" que es la imagen pre-procesada. La función *"cv2.dnn.blobFromImage(imagen, escala, tamaño, mean, intercambiar RB, recorte del centro)"* se encarga de este trabajo. Al pasar la imagen por la función se crear un blob de cuatro dimensiones con un tamaño 416x416. El parámetro "*mean*" se usa para normalizar la imagen por valores de los canales, en este caso no se usa (al no usarse no se ha investigado más). El parámetro *"intercambio"* se usa para evitar discrepancias ya que OpenCv trabaja con el orden de canales BGR, pero el parámetro *"mean"* usa el orden RGB, pero al no usarse está a "False". El parámetro "*recorte del centro"* está a *"True"* esto significa que se cambia el tamaño de la imagen de modo que un lado es igual al parámetro "escala" y el otro era igual o mayor. Se ha comprobado de manera experimental la mejor opción obteniendo los resultados de las Figuras 16 y 17.

A continuación, se procede a pasar la imagen por la red (línea 155) y obtener los valores de probabilidad, dimensiones del rectángulo y etiqueta identificativa correspondiente, es decir, los outputs de YOLO. Para ello primero obtenemos la probabilidad de las 80 clases dentó de la imagen y la etiqueta que le corresponde a cada una. Después extraemos las clases con más probabilidad de aparecer en la imagen y la comparamos con el "threshold" definido por el usuario. Si se cumple, se configuran las dimensiones de los rectángulos, la etiqueta del objeto y su probabilidad. Finalmente se dibuja sobre la imagen la detección procesada, es decir, un rectángulo que enmarque el objeto y su etiqueta identificativa.

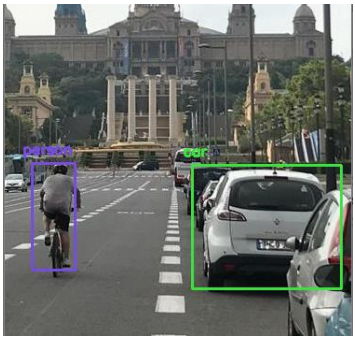

Figura 16, recorte del centro = *True*

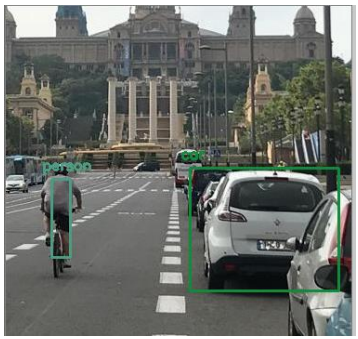

Figura 17, recorte del centro = *False*

Para añadir robustez al procesado, se comprueba que en "*índices*" hay algún valor, ya que si no hay nada significa que no hay objetos en el camino y la variable "*ya*" deberá tener un valor 0 (líneas 198-199). De esta

manera aseguramos que no se actuará sobre el volante de manera innecesaria borrando cualquier valor que pudiera haber en "*ya*".

Por otro lado, también tenemos la sentencia encargada de determinar si debe parar por haber más de un objeto en el camino, ya que el proceso no detecta si es posible pasar entre distintos objetos (líneas 201-202). Para terminar el *"hilo_camara"* se pulsa *"s"* (líneas 212-213).

El *"hilo_volante"* (línea 215), es el encargado de crear el efecto force-feedback sobre el volante. Para ello, primero se comprueba que no hay ningún efecto creado, y se crea con los parámetros de (líneas 230-245). Para que el efecto se aplique al volante, la función *descubrir()* tiene que devolver un valor m[0] (valor de la dirección si hay objeto) distinto de 0 (línea 252), ya que 0 hace referencia a que no hay ningún motivo para girar, es decir, no se actúa sobre la dirección. Si la dirección cambia (línea 269), se cargan los nuevos valores en el efecto (líneas 256-267) y se lanza (líneas 279-280). Mientas el parámetro m[0] no cambie el efecto estará en marcha, ya que significa que todavía está el objeto en el camino (líneas 282-285). Una vez ha cambiado, se para el efecto (línea 273) y se espera a la siguiente instrucción de "*descubrir()*". Al tratarse de un hilo que depende de otro hilo, no hay problema de que el valor m[0] cambie ya que se reciben datos en tiempo real. Para terminar el hilo se pulsa "a".

El hilo principal (línea 295) se encarga de la gestión de todo el programa. Es donde se declara el nodo

“proceso” encargado de publicar en el controlador y suscribirse a las imágenes del nodo cámara. También se encarga de ejecutar los hilos según la variable “*si*”, que como se ha mencionado anteriormente, se encarga de confirmar si se ha recibido la primera imagen del nodo cámara. Después de lanzar los hilos se sale del bucle, sólo se usa para asegurar el orden de ejecución de los hilos. Finalmente, en el “Main” hay un bucle que además de publicar los datos necesarios para controlar el robot (la dirección y velocidad), se encarga de activar/desactivar el “freno por más de un objeto” y frenar si hay más de un objeto o hay un objeto en el centro (líneas 335-337). Para desactivar el paro por objetos inevitables, se ha habilitado el botón 2 (líneas 330-331). Una vez desactivado se volverá a activar cuando el peligro haya desaparecido y el usuario haya accionado el botón por segunda vez (líneas 332-334). Esta opción es útil para tomar el control del robot en caso de que se pueda evitar los obstáculos de manera manual.

Para normalizar los datos recibidos del joystick, se han hecho las siguientes operaciones. En el volante para obtener valores entre -1 y 1, se ha tenido que normalizar dividiendo entre el valor máximo que registra el volante en giro (devolverá un valor aproximado a 0.95, líneas 325-326). Los valores del acelerador y freno tienen como valores máximos aproximados 32770 (acelerador/freno suelto) y -32770 (acelerando/frenando al máximo) que se han normalizado a [0,1]. Para ello se ha restado al valor recibido del pedal el valor máximo y esta resta se ha dividido entre el máximo valor que pueda devolver dicha resta. Cuando se está frenando/acelerando

totalmente se obtiene un 1 y un valor 0 cuando el pedal está suelto (líneas 322-324 y 327-32).

4.2.3 CONTROLADOR WEBOTS:

A continuación, tenemos el código encargado de controlar la dirección y velocidad del robot (Anexo III). Para controlar la dirección, al estar asociado al volante, se ha decidido manipular la velocidad de cada rueda de manera independiente, con la lógica explicada a continuación. Para los giros, las dos ruedas giran hacia el mismo sentido, pero a velocidades distintas. Por ejemplo, si el giro es hacia la derecha, la rueda derecha irá a una velocidad inferior que la izquierda específicamente a factor 1-eje menor y viceversa. En el caso de girar por ejemplo hacia la izquierda de manera abrupta, la rueda izquierda tendría velocidad 0 y la derecha la marcada por el pedal. Este factor de diferencia entre las ruedas, se ha elegido en función de la intensidad del giro, de esta manera si el volante está girado al máximo en una dirección, la rueda que esté en la misma dirección que el giro tenía que tener velocidad nula (se mueve por inercia) y la rueda contraria la velocidad marcada por el acelerador. Para conseguir esta velocidad nula se necesita que el factor tuviera un valor 0 cuando el volante tiene un valor 1. Por otra parte, si el volante está recto, las dos ruedas necesitan girar en el mismo sentido y a la misma velocidad, esto se consigue con un factor 1 y la posición actual de los ejes sería 0. Si esto lo expresamos en una gráfica obtendríamos el siguiente resultado:

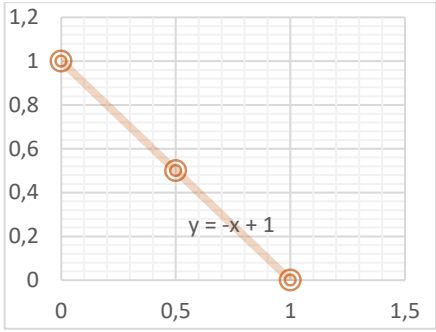

Figura 16, Grafica controlador

Con esta lógica se podría conseguir el control de dirección del robot, pero al no tener nunca el 1/0 absoluto, se ha necesitado añadir algunas sentencias que garanticen que el robot irá recto si el volante está recto (líneas 77-87). Donde se ha elegido un valor de 0.05 en lugar del cero absoluto porque es complicado poner el volante exactamente en esa posición. Por tanto, se permite un margen de error (zona muerta) para considerar que el volante indica una dirección 0.

Para controlar el sentido, se ha habilitado un botón que permite activar la marcha atrás (líneas 56-60).

Figura 17, ilustración de los botones usados para activar y desactivar la marcha atrás.

Finalmente, para controlar la velocidad (líneas 64-73), se necesita guardar el último valor de velocidad registrado para poder compararlo con la velocidad actual y determinar si se está desacelerando o bien se ha frenado. El último valor de la velocidad depende de la dirección, ya que dependiendo del sentido de giro del volante una rueda tenía más velocidad que otra (líneas 64-67). Por tanto, antes de guardarlo se comprueba que se cogerá el valor máximo.

Figura 18, freno y acelerador para el control de velocidad por parte del usuario

Una vez tenemos este valor pueden darse tres casos. El primero es que el robot ya esté parado (líneas 68-69), entonces se pondrá la velocidad a 0. El segundo es que la velocidad esté disminuyendo, pero todavía no esté parado, entonces se disminuirá la velocidad poco a poco (por inercia, líneas 70-71). Finalmente, el tercer caso es que se esté frenando, entonces la velocidad se disminuirá un valor equivalente a la quinta parte recibida del freno (1/5 obtenido de manera experimental sobe el simulador), ya que si se frenara al valor completo la frenada sería demasiado brusca (este valor actúa como ABS, líneas 72-73).

**Para usar este controlador en otro robot hay que cambiar: MAX_SPEED, y el nombre de los motores.*

4.2.3.1 ARCHIVO .INI DEL CONTROL:

Hace falta un archivo de configuración para usar el controlador con ROS, para ello se tienen que especificar las siguientes rutas (en este caso los valores son los siguientes):

PYTHONPATH = *C:\opt\python27amd64\Lib\site-packages:C:\opt\ros\melodic\x64\lib\site-packages*

ROS_LOG_DIR = *C:\tmp\logsim*

ROS_PACKAGE_PATH = *C:\opt\ros\melodic\catkin_ws\src;C:\opt\ros\melodic\x64\share*

ROS_MASTER_URI = *http://localhost:11311*

5. RESULTADOS EXPERIMNTALES Y LIMITACIÓNES

Se puede activar el sisema en dos modos:

- Modo simulación: Se puede simular en el terminal donde se ejecta el nodo procesamiento o bien en otro terminal. Si se va a usar un terminal diferente se necesita ediar en el archivo ".ini" la IP y puerto del master (se debe abrir el puerto que se vaya a usar en el firewall).
  Para usar el modo simulación se ejecutan 3 consolas VS/ROS en el terminal donde se ha conectado el joystick, una para cada módulo:

  1. Master: Ejecutar *roscore* (Figura 21).
  2. Nodo cámara: Ejecutar *rosrun paquete publishercam.py* (Figura 22).
  3. Nodo proceso: Ejecutar *rosrun paquete publishernode.py.* Para desactivar el nodo cámara pulsar *"S"*. Para desactivar el nodo volante pulsar *"A"*. Esto permite desactivar los efectos sobre el volante pero el usuario puede seguir contorlándolo (Figura 23).

  El nodo controlador se ejecuará en el terminal que se desee simular, o bien ejecutando Webots desde su acceso directo o bien en una consola Windows escribir *"webots"* si se han configurado todas las rutas en los pasos mencionados en la instalación (Figura 24).

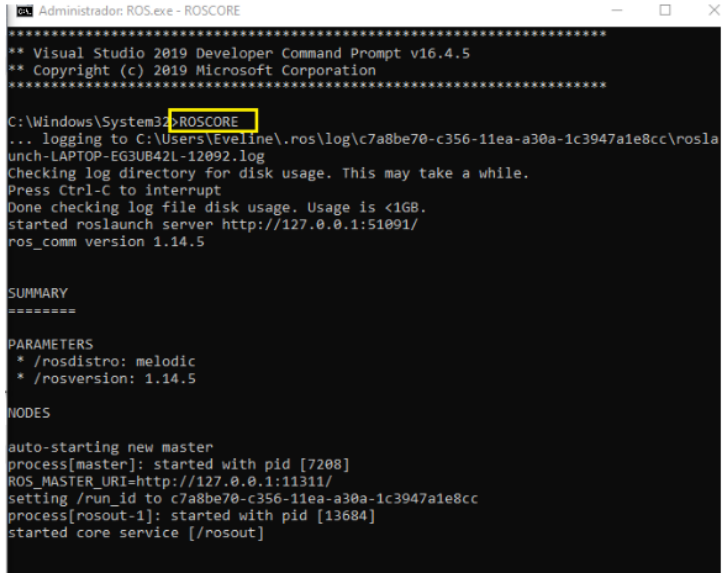

Figura 19, Terminal ROS ejecutando roscore

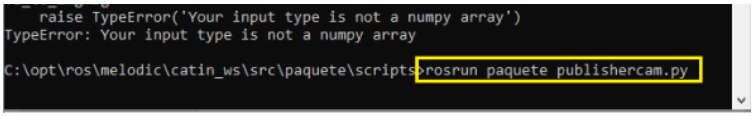

Figura 20, Terminales ROS ejecutado nodo cámara.

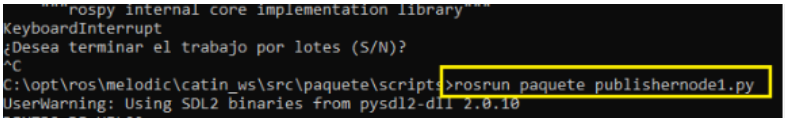

Figura 21, Terminal ROS ejecutando nodo Publisher.

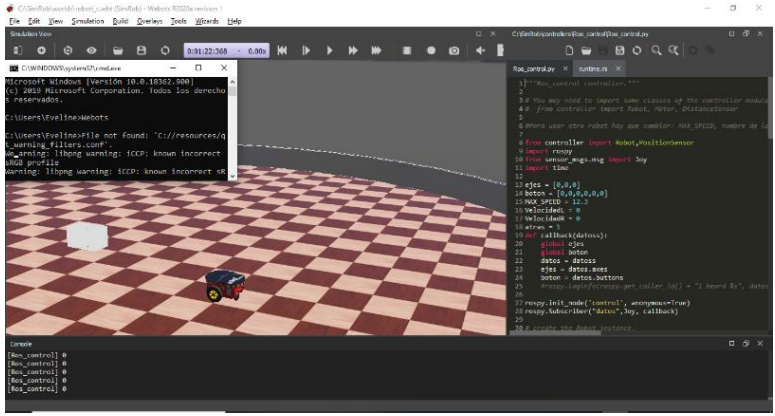

Figura 22, Simulador Webots con el robot Pioneer 3DX en un entorno virtual vacío.

Una vez activados todos los módulos, se verá la siguiente pantalla:

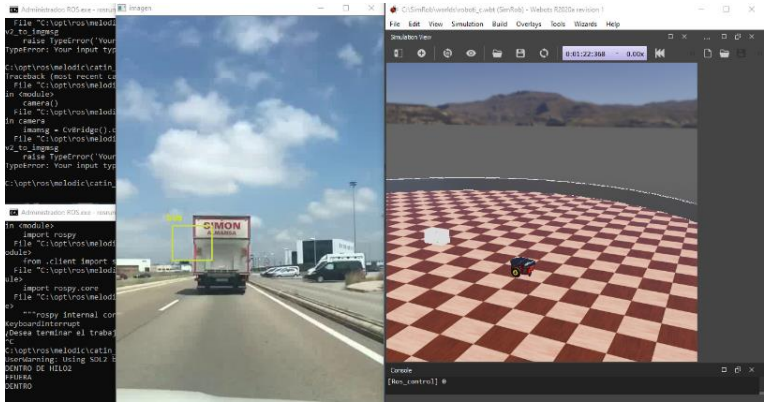

Figura 23, Visualización de webcam y simulador.

Por un lado, las imágenes del nodo cámara y por otro lado el comportamiento del robot respecto

al entorno captado por la cámara. En este caso se está usando un solo terminal (Figura 25).

- Modo no simulación: Si se desea usar un robot físico, se necesita ejecutar "roscore" y el nodo "Procesamiento" en el terminal donde se ha conectado el joystick. También es necesario asegurarse que el robot publica y se suscribe a los nodos que hay en "Procesamiento", es decir, las imágenes se publican en el topic "camera" y se suscribe a un topic llamado "datos" para recibir los datos del joystick. En cuanto al controlador, si se quiere reproducir el funcionamiento explicado anteriormente, se debería adaptar y cargar al robot el código del Anexo III. Por otro lado, si se usara el controlador del robot, la función de marcha atrás no funcionaría con el botón de la Figura 19.

5.1 FUNCIONAMIENTO

Al activar el nodo Cámara (Anexo I), éste comienza a enviar las imágenes que lee de la webcam (línea 7) a través de ROS (línea 20).

Figura 24, imagen leída de la webcam

Las imágenes se envían al nodo Procesamiento (Anexo II) donde se recibe una imagen como en la Figura 26 a través de la función “callback” (línea 129) y la guarda en la variable “images”. Una vez se ha recibido “images”, se empieza a ejecutar todos los hilos de la

manera mencionada en el apartado anterior. Primero se ejecuta “hilo_camara()” (línea 136) donde “images” se redimensiona a 640x480 y se guardan sus nuevas dimensiones (línea 148-151), Width :640, Height: 480. Se crea el blob a partir de “images” y se pasa por la red (líneas 153-156) donde se obtienen los valores de salida “outs”, en este caso, ['yolo_16', 'yolo_23']. A partir de los “outs” obtenidos, se extrae:

- Un índice “class_id” que hace referencia a la posición del objeto detectado en la lista “yolov3.txt” (Figura 27) que contiene una lista de todos los objetos para los cuales ha sido entrenada la red (línea 170) en este caso se obtendría class_id = 2.
- La posibilidad “confidence” de que el objeto detectado concuerde con la etiqueta asignada (línea 171). Para la Figura 26 confidence = 0.70927.
- Las dimensiones (en función del tamaño de la imagen) de la caja que contiene el objeto (líneas 172-177), center_x = 312. center_y = 151, w = 79, h = 58, x = 272.5, y = 122.0.

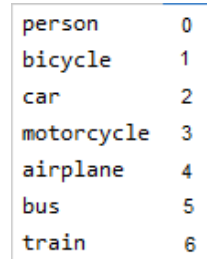

Figura 25, Primeros 7 objetos detectables por YOLOv3.

Con todos los datos obtenidos se calcula las dimensiones finales de la caja (del objeto) que se dibujará sobre la imagen. Como se ha mencionado en la explicación de YOLO, se dibujan múltiples cuadrados sobre la imagen, pero con la función de la "línea 182" solamente se guardarán las cajas que tengan relación de dimensiones con el centro del objeto por encima del umbral definido (conf_threshold = 0.5, nms_threshold = 0.5). A continuación, la función "draw_prediction()" obtiene la etiqueta (label) equivalente para el class_id obtenido a la salida de la red, en nuestro caso, en la posición 2 del archivo .txt está "car" (Figura 27). Finalmente, se dibuja la caja alrededor del objeto y se guarda su posición (centro, izquierda, derecha) en la variable "ya" por pantalla se muestra la Figura 28. Como se deduce de las Figuras 26 y 28, la variable "ya" toma el valor "derecha". Se vuelve al "hilo_camara ()" donde se revisan varias condiciones, la primera determina si hay objeto en la imagen (por seguridad) y la segunda condición busca más de un objeto en la imagen, en este caso sólo se detecta un objeto. Por lo

tanto, no se cumple ninguna de las condiciones mencionadas. Se pasa a obtener el valor de m (si se debe girar el volante o no) a partir de la función “descubrir()” (línea 100). Supongamos que el volante está girando hacia la derecha, “posact” > 0, se cumpliría la condición de la “línea 118”, además no se está frenando y el objeto sigue a la derecha. Por tanto, se cumple la condición de la “línea 120” también, lo que significa que se va a crear un efecto de fuerza sobre el volante obligando al usuario a girar para no chocar con el objeto.

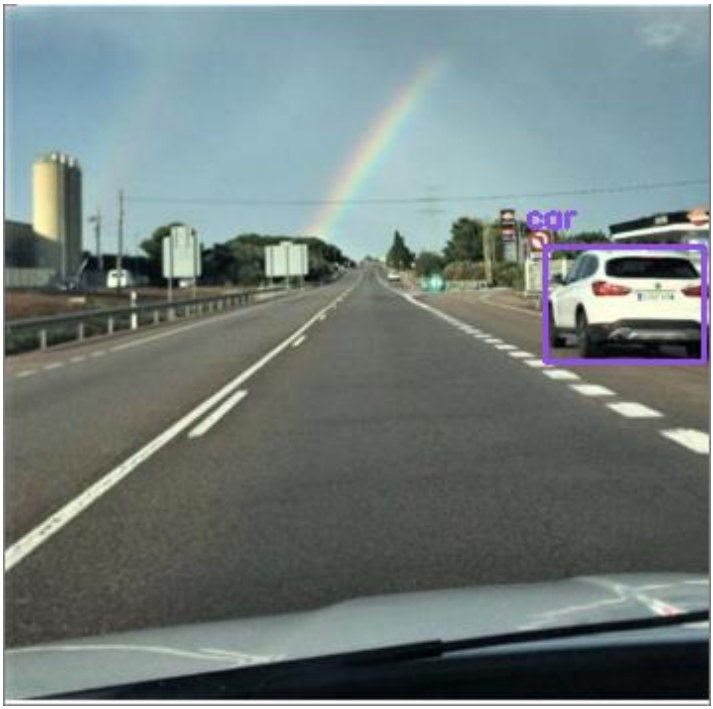

Figura 26, Imagen tras pasar por la red y una vez se ha dibujado la caja que enmarca el objeto.

El efecto de fuerza se crea en el “hilo_volante()” que ha empezado a ejecutarse 4s después del “hilo_camara()” (línea 216). Se crea el efeto de fuerza con la dirección m[0] = 1 (línea 237) y hasta que no se reciba del “hilo_camara()” que el peligro ha pasado, “m[0] == 0” o bien que hay un nuevo peligro “m[0] ¡= ant” el volante seguirá girando en la misma dirección. En el caso de la Figura 26, dejaría de girar cuando el volante ya no tuviera la misma dirección que el coche. Si por lo contrario al girar se detecta un objeto a la izquierda, se crearía un nuevo efecto sobre el volate evitando el nuevo peligro (línea 280).

Una situación que podría darse en las Figuras 26 y 28 es que el usuario considere que el peligro no provoca una situación adversa o bien que evite el peligro manualmente, en esta situación el usuario puede tomar el control del volante aun cando se ha creado el efecto de fuerza, el usuario siempre tiene la posibilidad de tomar el control ya que cuando se aplica fuerza contraria al efecto, éste cesa.

Figura 27, ilustración del botón para apagado del freno por objeto inevitable.

Otras posibles situaciones independientes de las Figuras 26 y 28 son:

- Los objetos son inevitables, como en la Figura 30. En este caso no se crearía el efecto de fuerza ya que en la "línea 202" se detectaría más de un objeto en la imagen y se asignaría a la variable "ya" (contiene la posición de los objetos en la imagen o bien indica si para por varios objetos) el valor "para". Aquí es el usuario el que debe decidir sobre la acción, ya que el sistema hará que el móvil pare. El usuario puede desactivar esa parada accionando el botón "freno por objeto inevitable" (Figura 29).
- Objeto en el centro, como en la Figura 31. En este caso, si el usuario no está frenando, se activa un freno de emergencia para movilizar el vehículo (línea 336).

Figura 28, Detección de dos objetos que son imposibles de evitar.

Figura 29, Detección de objeto en el centro.

Como se ha mencionado anteriormente, el nodo procesamiento envía los datos al nodo controlador (Anexo III) en tiempo real. Por tanto, los datos del que recibe el controlador son las posiciones del volante y pedales ya procesados (función callback líneas 20-22). En el caso de la imagen se recibiría primero la posición del volante en dirección al objeto (posición > 0,5) y después la posición creada por el efecto de fuerza (posición: 0,4 0,3 ... 0 ... 0,1 ... dependiendo de la duración del efecto).

El comportamiento descrito se puede ver en el siguiente video:
https://mmedia.uv.es/buildhtml/64293

Contraseña: TFG2020Eveline

5.2 LIMITACIONES

Algunas de las limitaciones que presenta el sistema de asistencia a la conducción expuesto son:

- La incapacidad de determinar si se puede pasar entre varios objetos, es decir, a no ser que se desactive la opción (desactivación explicada en apartados anteriores), siempre que haya más de un objeto en el camino, el sistema hará parar el móvil.
- No es capaz de determinar al 100% si el objeto interfiere en el camino. Por tanto, siempre que se detecte un objeto en algunas de las situaciones mencionadas, intentará evitarlo, aun si el objeto está a una distancia no peligrosa.

Por otro lado, también hay limitaciones en cuanto a la ejecución del código en un solo terminal, ya que se necesita un terminal potente y si no se tiene, la simulación no es fluida, es decir, hay varios segundos de retraso entre los datos enviados desde "Procesamiento" al controlador. Por tanto, no se puede apreciar el funcionamiento del sistema al 100%.

6. CONCLUSIONES Y PROYECTOS FUTUROS

Los resultados obtenidos a lo largo del proyecto han sido satisfactorios. La realización de este trabajo me ha permitido adquirir nuevos conocimientos en áreas como la robótica e informática, además de aumentar mi capacidad de resolución de problemas y autonomía. Entre los conocimientos adquiridos destacaría en primer lugar el lenguaje usado para llevar acabo todo el proyecto, Python, un lenguaje que no había usado hasta ahora y diferente a los estudiados a lo largo del curso. Por otro lado, también están los hilos de ejecución para la programación concurrente y la gran importancia de drivers para uso de periféricos. En segundo lugar, están los conocimientos sobre redes neuronales como: qué son y su funcionamiento básico, cómo están formadas, cómo se comportan, cómo se entrenan, etc. En último lugar, conocimientos sobre ROS como: qué es, sus componentes, cómo crear nodos capaces de comunicar distintos elementos Hardware y la importancia de tener un protocolo estandarizado. Gracias a la cantidad de recursos que ofrece ROS se ha conseguido simular el comportamiento programado, pero además al tener paquetes de mensajes estándar, se podría probar el comportamiento descrito en robots reales (compatibles con ROS) simplemente enviando los datos a su controlador, sin necesidad de cambiar los tipos de datos.

A pesar del estudio y análisis de cada uno de los componentes usados (mencionados en el párrafo

anterior), a lo largo del proyecto han surgido varios problemas. Algunos se han mencionado en los apartados 3.3.1 y 3.1.1, pero considero que el factor común a todos ha sido el uso de ROS en Windows. Usar esta nueva versión publicada hace poco ha limitado considerablemente la posibilidad de integrar esta herramienta con otras como simuladores. Cosa que ha aumentado la carga de trabajo. Por ejemplo, Webots para ROS en Windows no tiene controlador estándar, lo que ha llevado a crear uno propio. Por un lado, gracias a esto el controlador es específico para este proyecto lo que considero que es una mejora, pero por otro lado ha sido un imprevisto que ha costado varios días de trabajo. Otro aspecto que considero un problema es la cantidad de componentes que necesita para su funcionamiento, además versiones antiguas de estos componentes. Por ejemplo, al inicio del proyecto se usó PyCharm para escribir y analizar el código de la red. PyCharm es un entorno de desarrollo para trabajar con Python que usa la versión 3.6 o superior, mientras que ROS (Windows) usa la 2.7. A la hora de pasar el código a un nodo ROS había librerías que no eran compatibles con Python 2.7 y se tuvo que reinstalar otras versiones de estas librerías. Tener dos versiones de Python y además distintas versiones de las mismas librerías en distintas rutas y todas ellas configuradas en las "variables de entorno" de Windows hizo que se tuviera que desinstalar ROS, Python y borrar de todas las librerías dobles para luego volver a instalar sólo una versión de cada una. Para terminar, a todo esto se le puede sumar la falta de información sobre problemas de

ROS en Windows, ya que la mayoría de usuarios usan ROS en Linux y por tanto toda la información disponible es para Linux.

Por otro lado, como se ha mencionado al principio del proyecto, la situación vivida por la pandemia del covid-19 ha tenido también influencia en cómo se ha llevado a cabo el proyecto. La idea inicial era probar el funcionamiento en un robot móvil real, en un entorno de oficina. Pero al no poder acceder al robot se cambió a un entrono simulado cuando ya había parte del proyecto realizado. Éste es también el motivo por el cual, a la hora de simular, el entorno de simulación es tan simple (solo el robot y algún objeto simple). Esta situación es causa de otra de las limitaciones principales del trabajo, la coordinación de todos los nodos a la hora de simular. Este problema se debe a que el ordenador usado no es lo suficientemente potente para ejecutar todos los nodos de ROS y además el simulador al mismo tiempo, causando tiempos de retraso entre el comportamiento descrito en los nodos y lo que aparece en el simulador. Pero a pesar de esos descuadres de tiempo, se ha podido probar el funcionamiento completo con descuadres y también por partes.

En conclusión, se puede decir que los resultados obtenidos son satisfactorios ya que se ha conseguido cumplir con el objetivo propuesto. Además, se han implementado funciones adicionales como poder controlar el sentido de la dirección y frenar en caso de no poder evitar los objetos.

Como trabajos futuros se podría estudiar la viabilidad de aplicar el algoritmo a los automóviles. Como se mencionaba al principio del proyecto, se ha probado con este tipo de entornos, pero no se ha tenido en cuenta todos los posibles casos que pudieran aparecer a la hora de la conducción. Por otro lado, se debería entrenar una red con datos específicos de este tipo de entornos. Adicionalmente, se podría integrar con la cámara funciones de medición de distancias, lo cual eliminaría algunas limitaciones como no poder determinar si el móvil es capaz de pasar entre varios objetos y mejoras como controlar la velocidad en función de la distancia al objeto. Otra opción para mejorar el sistema sería el uso de sensores de posición y distancia, como el LIDAR, que aportarían precisión al sistema en cuanto a distancias cortas. En conclusión, se puede decir que como trabajo futuro se puede estudiar la posibilidad de desarrollar un sistema de conducción automática más completo y avanzado.

7. BIBLIOGAFÍA

[1] Carlos García Moreno. Deep-Learning
https://www.indracompany.com/es/blogneo/deep-learning-sirve

[2] DotCSV. Redes neuronales.
https://www.youtube.com/watch?v=MRIv2IwFTPg

[3] DotCSV. Redes neuronales.
https://www.youtube.com/watch?v=uwbHOpp9xkc

[4] DotCSV. Redes neuronales.
https://www.youtube.com/watch?v=M5QHwkkHgAA

[5] DotCSV. Redes neuronales.
https://www.youtube.com/watch?v=eNIqz_noix8

[6] Damián Jorge Matich. Redes neuronales.
https://www.frro.utn.edu.ar/repositorio/catedras/quimica/5_anio/orientadora1/monograias/matich-redesneuronales.pdf

[7] Blog: Juan Ignacio Bagnato. Redes Convolucionales.
https://www.aprendemachinelearning.com/como-funcionan-las-convolutional-neural-networks-vision-por-ordenador/

[8] Jaime Durán. Técnicas de Regularización Básicas para Redes Neuronales. https://medium.com/metadatos/t%C3%A9cnicas-de-regularizaci%C3%B3n-b%C3%A1sicas-para-redes-neuronales-b48f396924d4

[9] Blog: Matthijs Hollemans. Funcionamiento de YOLO. https://machinethink.net/blog/object-detection-with-yolo/

[10] Manish Chablani. YOLO Funcionamiento. https://towardsdatascience.com/yolo-you-only-look-once-real-time-object-detection-explained-492dc9230006

[11] Enrique A. YOLO. https://medium.com/@enriqueav/detecci%C3%B3n-de-objetos-con-yolo-implementaciones-y-como-usarlas-c73ca2489246

[12] ROS. http://wiki.ros.org/ROS/

[13] César Vargas. Que es ROS. http://mariachirobot.com/que-es-ros/

[14] Autrancen. ROS sistema operativo. http://www.autracen.com/ros/

[15] ROS. CvBrige http://wiki.ros.org/cv_bridge/Tutorials/ConvertingBetweenROSImagesAndOpenCVImagesPython

[16] Simuladores.
https://www.coursehero.com/file/56983062/TRABAJO-Simuladorespdf/

[17] Simuladores.
https://en.wikipedia.org/wiki/Robotics_simulator

[18] Instalación Webots.
https://cyberbotics.com/doc/guide/installing-webots

[19] Robots Webots y especificaciones.
https://cyberbotics.com/doc/guide/robots

[20] Controladores en Webots
https://cyberbotics.com/doc/guide/tutorial-4-more-about-controllers?tab-language=python#program-a-controller

[21] Pioneer 3-DX. Especificaciones.
https://www.generationrobots.com/media/Pioneer3DX-P3DX-RevA.pdf

[22] Pioneer 3-AT. Especificaciones.
https://www.generationrobots.com/media/Pioneer3AT-P3AT-RevA-datasheet.pdf

[23] Librería SDL2
https://wiki.libsdl.org/CategoryJoystick

[24] Inteligencia artificial
https://www.salesforce.com/mx/blog/2017/6/Que-es-la-inteligencia-artificial.html

[25] Inteligencia artificial
https://www.iberdrola.com/innovacion/que-es-inteligencia-artificial

[24] Machine Learning y Deep Leaning
https://blogthinkbig.com/diferencias-entre-machine-learning-y-deep-learning

[25] Machine Learning
https://cleverdata.io/que-es-machine-learning-big-data/

[26] Visión Artificial
http://www.etitudela.com/celula/downloads/visionartificial.pdf

8. BIBLIOGRAFÍA DE IMÁGENES

[1] Figura 1. (Figura superior derecha)
https://es.wikihow.com/aprender-a-conducir

[2] Figura 4. Autor DotCSV.
https://www.youtube.com/watch?v=uwbHOpp9xkc

[3]Figura 7. Manish Chablani.
https://towardsdatascience.com/yolo-you-only-look-once-real-time-object-detection-explained-492dc9230006

[4] Figura 10. Lofarolabs.
http://wiki.lofarolabs.com/index.php/Moving_The_Pioneer_3-DX_In_Gazebo

ANEXO I: CÓDIGO DATOS CÁMARA

```
1   import cv2
2   import rospy
3   from sensor_msgs.msg import Image
4   from cv_bridge import CvBridge
5
6   def camera ():
7       cam = cv2.VideoCapture(0)
8       pub =   rospy.Publisher('camera',Image,queue_size=10)
9       rospy.init_node('camera', anonymous=True)
10      rate = rospy.Rate(15)
11      foto = Image()
12      cont = 0
13      while not rospy.is_shutdown():
14          _,image = cam.read()
15          imamsg =
16          CvBridge().cv2_to_imgmsg(image,'bgr8')
17          imamsg.header.stamp = rospy.Time.now()
18          cont =cont + 1
19          imamsg.header.seq = cont
20          pub.publish(imamsg)
21  #       rospy.loginfo(imamsg)
22
23  if __name__ == '__main__':
24
25  try:
26  camera ()
27  except rospy.ROSInterruptException:
28  pass
```

ANEXO II: CÓDIGO PROCESAMIENTO

```
1   #Librerías:
2   import cv2
3   import numpy as np
4   from time import time
5   import threading
6   from sdl2 import *
7   import sys
8   import rospy
9   from sensor_msgs.msg import Joy, Image
10  from cv_bridge import CvBridge
11
12  #Variables Globales:
13  ya = int
14  classes = str
15  posact = int
16  ima = None
17  event = None
18  net = None
19  COLORS = None
20  joy = int
21  ff = float
22  efect = None
23  images = None
24  m = (0,0)
25
26  def configuracion ():
27      global classes
```

```
    global event
    global ima
    global net
    global COLORS
    global joy
    global ff
   global efect

#  Para Volante:
    SDL_Init(SDL_INIT_HAPTIC)
    SDL_Init(SDL_INIT_JOYSTICK)
    SDL_JoystickEventState(1)

    joy = SDL_JoystickOpen(0)
    ff = SDL_HapticOpenFromJoystick(joy)
    event = SDL_Event()
    efect = SDL_HapticEffect()

#  Para Yolo:
#  ima = cv2.VideoCapture(0)  # habilito cámara
para pruebas

    with open("yolov3.txt", 'r') as f:

        classes = [line.strip() for line in f.readlines()]

    net = cv2.dnn.readNet("yolov3-
    tiny.weights","yolov3-
    tiny.cfg")
    COLORS = np.random.uniform(0, 255,
    size=(len(classes),3))
```

```
59
60 def get_output_layers(net):
61
62     layer_names = net.getLayerNames()  # pone en
63    la variable todas las capas de la net
64
65     output_layers = [layer_names[i[0] - 1] for i in
66     net.getUnconnectedOutLayers #Devuelve el
67     valor de las neuronas de la última capa.
68
69     return output_layers
70
71 def draw_prediction(img, class_id, confidence, x,
72  y, x_plus_w,  y_plus_h, COLORS, classes):
73
74     global ya
75
76     label = str(classes[class_id])
77     color = COLORS[class_id]
78
79     if (x_plus_w < 240):
80         ya = "izquierda"
81
82     elif (x > 240):
83         ya = "derecha"
84
85     elif (240 - x < x_plus_w - 240):
86         ya = "derecha"
87
88     elif (240 - x > x_plus_w - 240):
89         ya = "izquierda"
```

```
90
91      elif (240 - x == x_plus_w-240):
92          ya = "centro"
93
94      cv2.rectangle(img, (x, y), (x_plus_w, y_plus_h),
95      color, 2)
96      cv2.putText(img, label, (x - 10, y - 10),
97      cv2.FONT_HERSHEY_SIMPLEX, 0.5, color, 2)
98
99
100def descubrir():
101
102     global event
103     global joy
104     global ya
105     SDL_PumpEvents(event)
106     posact = SDL_JoystickGetAxis(joy, 0)
107
108     if (posact < 0 and ya == 'izquierda'):
109         SDL_Delay(2000)
110         if (ya == 'izquierda' and fre > 0.8):
111             ya = 0
112             return (-1,posact)
113         else:
114             ya = 0
115             return (0, 0)
116
117     elif (posact > 0 and ya == 'derecha'):
118         SDL_Delay(2000)
119         if (ya == 'derecha' and fre > 0.8):
120             ya = 0
```

```
121             return (1,posact)
122         else:
123             ya = 0
124             return (0, 0)
125 
126     else:
127         return (0, 0)
128 
129def callback(foto):
130     global images
131     global si
132     images = CvBridge().imgmsg_to_cv2(foto,
133     desired_encoding='bgr8')
134     si=1
135 
136def hilo_camara():
137 
138     global net
139     global COLORS
140     global class_id
141     global images
142     global m
143 
144#    start_time = time()
145     while True:
146#         _, image = ima.read()#camara solo para
147         pruebas
148         images = cv2.resize(images, (640, 480))
149         dim = images.shape
150         Width = images.shape[1]
151         Height = images.shape[0]
```

```
152     scale = 0.00392
153     blob = cv2.dnn.blobFromImage(images,
154     scale, (416, 416), (0, 0, 0), False, crop=Ture)
155     net.setInput(blob)#imput the blob
156     outs = net.forward(get_output_layers(net))
157
158     class_ids = []
159     confidences = []
160     boxes = []
161     conf_threshold = 0.5
162     nms_threshold = 0.5
163
164     for out in outs:
165        for detection in out:
166           scores = detection[5:]#devuelve la
167           probapilidad
168           de las 80 clases
169           class_id = np.argmax(scores)
170           confidence = scores[class_id]
171           if confidence > conf_threshold:
172              center_x = int(detection[0] * Width)
173              center_y = int(detection[1] * Height)
174              w = int(detection[2] * Width)
175              h = int(detection[3] * Height)
176              x = center_x - w / 2
177              y = center_y - h / 2
178              class_ids.append(class_id)
179              confidences.append(float(confidence))
180              boxes.append([x, y, w, h])
181
182     indices = cv2.dnn.NMSBoxes(boxes,
```

```
183     confidences, conf_threshold, nms_threshold)
184
185     for i in indices:
186
187         i = i[0]
188         box = boxes[i]
189         x = box[0]
190         y = box[1]
191         w = box[2]
192         h = box[3]
193         draw_prediction(images, class_ids[i],
194         confidences[i], int(x), int(y), int(x + w),
195         int(y + h), COLORS, classes)
196
197
198     if (indices==()):
199         ya=0
200
201     if (len(indices) > 1):
202         ya = "para"
203     else:
204         m=descubrir()
205
206     cv2.imshow("imagen", images)
207#      elapsed_time = time() - start_time
208#      print("Elapsed time net: %0.10f seconds." %
209#      elapsed_time)
210
211
212     if cv2.waitKey(1) & 0xFF == ord('s'):
213         break
```

```
214
215def hilo_volante ():
216
217    global event
218    global posact
219    global joy
220    global ff
221    global efect
222    global m
223    efect_id=-1
224
225    while(hilo1.is_alive()):
226
227        try:
228
229            if(efect_id!=0):
230                efect.type =
231                SDL_HAPTIC_CONSTANT
232                efect.constant.type =
233                SDL_HAPTIC_CONSTANT
234                efect.constant.direction.type =
235                SDL_HAPTIC_CARTESIAN
236                efect.constant.direction.dir[0] = m[0]
237                efect.constant.length = 100  #duración
238                del efecto
239                efect.constant.delay = 100  #retraso
240                antes de lanzar el efecto
241                efect.constant.level = 3000  #fuerza del
242                efecto    constante
243                efect.constant.attack_lenght = 100
244                #duración al inicio del efecto
```

```
245             efect.constant.fade_lenghth = 100
246             #duración del desvanecimiento
247             ant=m[0] #Guarda el valor de la
248             dirección actual
249             efect_id =
250             SDL_HapticNewEffect(ff,efect)
251
252         if (m[0] != 0):
253
254             while True:
255
256                 efect.type =
257                 SDL_HAPTIC_CONSTANT
258                 efect.constant.type =
259                 SDL_HAPTIC_CONSTANT
260                 efect.constant.direction.type =
261                 SDL_HAPTIC_CARTESIAN
262                 efect.constant.direction.dir[0] = m[0]
263                 efect.constant.length = 500
264                 efect.constant.delay = 100
265                 efect.constant.level = 3000
266                 efect.constant.attack_lenght = 100
267                 efect.constant.fade_lenghth = 100
268
269                 if(m[0] != ant):
270
271                     if(m[0] == 0):
272
273                 SDL_HapticStopEffect(ff,efect_id)
274                         break
275
```

```
276                 else:
277                     ant = m[0]
278
279                   SDL_HapticUpdateEffect(ff,
280                   effect_id, efect)
281
282               while(m[0]==ant):
283                   SDL_HapticRunEffect(ff, efect_id,
284                   1)
285                   SDL_Delay(700)
286
287               #print(f" Error al lanzar el efeto:
288               {SDL_GetError()}")
289
290       except  KeyboardInterrupt:
291           if cv2.waitKey(1) & 0xFF == ord('a'):
292               break
293
294#Main
295try:
296
297    cont = 0
298    paro = 0
299    si = 0
300
301    configuracion()
302    pub = rospy.Publisher('datos', Joy,
303    queue_size=10)
304    rospy.init_node('proceso', anonymous=True)
305    rospy.Subscriber("camera",Image, callback)
306    rate = rospy.Rate(10)
```

```
307         datoss = Joy()
308
309   hilo1 = threading.Thread(target=hilo_camara)
310   hilo2 = threading.Thread(target=hilo_volante)
311
312   while True:
313       if(si==1):
314           hilo1.start()
315           SDL_Delay(4000)
316           hilo2.start()
317           SDL_Delay(5000)
318           break
319
320   while not rospy.is_shutdown():
321       SDL_PumpEvents(event)
322       acelerador = float(-
323       ((SDL_JoystickGetAxis(joy , 1)-
324       32770.0)/65540.0)) #Acelerador
325       volante = float((SDL_JoystickGetAxis(joy ,
326       0)/32770.0)) #Volante
327       freno = float((-((SDL_JoystickGetAxis(joy ,
328       2)- 32770.0)/65540.0))) #Freno
329
330       if (SDL_JoystickGetButton(joy , 2) == 1):
331           paro = 1
332       elif ( ya != "para" and
333       SDL_JoystickGetButton(joy , 2) == 0):
334           paro = 0
335       if ((ya == "centro" and freno > 32760) or (ya
336       ==   "para" and paro == 0 )):
337           freno = 0
```

```
338
339         datoss.axes = [ volante ,acelerador , freno]
340
341         datoss.buttons = [
342         SDL_JoystickGetButton(joy,  0),
343         SDL_JoystickGetButton(joy , 1),
344         SDL_JoystickGetButton(joy , 2),
345         SDL_JoystickGetButton(joy , 3),
346         SDL_JoystickGetButton(joy , 4),
347         SDL_JoystickGetButton(joy , 5)]
348         datoss.header.stamp = rospy.Time.now()
349         cont = cont + 1
350         datoss.header.seq = cont
351#        rospy.loginfo(datoss)
352         pub.publish(datoss)
353         rate.sleep()
354
355except KeyboardInterrupt:
356
357    cv2.destroyAllWindows()
358    SDL_HapticClose(ff)
359    SDL_HapticClose(joy)
```

ANEXO III: CONTROLADOR

```
1   #Para usar otro robot hay que cambiar:
2   MAX_SPEED, nombre de los motores.
3
4   from controller import Robot,PositionSensor
5   import rospy
6   from sensor_msgs.msg import Joy
7   import time
8
9   ejes = [0,0,0]
10  boton = [0,0,0,0,0,0]
11  MAX_SPEED = 12.3 #Velocidad máxima en rsd/s
12  del robot
13
14  VelocidadL = 0
15  VelocidadR = 0
16  atras = 1
17  def callback(datoss):
18      global ejes
19      global boton
20      datos = datoss
21      ejes = datos.axes
22      boton = datos.buttons
23
24  rospy.init_node ('control', anonymous=True)
25  rospy.Subscriber("datos",Joy, callback)
26
27  # create the Robot instance.
28  robot =Robot()
```

```
29
30 # get the time step of the current world.
31 timestep = int(robot.getBasicTimeStep())
32
33 L = robot.getMotor('left wheel')
34 R = robot.getMotor('right wheel')
35
36 # turn on velocity control for both motors
37 L.setPosition(float('inf'))
38 R.setPosition(float('inf'))
39
40 L.setVelocity(0.0)
41 R.setVelocity(0.0)
42
43 timeStep = int(robot.getBasicTimeStep())
44
45 time.sleep(1)
46
47 while ( robot.step(timeStep) != -1 and not
48 rospy.is_shutdown()):
49
50         v = ejes[1] #Acelerador
51         eje = abs(ejes[0]) #Volante
52         freno = ejes[2] #Freno
53
54        #Control del sentido
55
56         if (boton[1]==1): #activo marcha atrás
57             atras = -1 #cambia el sentido de la
58             velocidad
59         elif (boton[0]==1): #desactivo marcha atras
```

```
60          atras = 1
61
62      #Control de Velocidad
63
64      if (ejes[0]>0.05):
65          v_ant = VelocidadL/MAX_SPEED
66      else:
67          v_ant = VelocidadR/MAX_SPEED
68      if (v < 0.05 or freno > 0.01):
69          v = 0
70      if (v < v_ant and v_ant - v > 0.05):
71          v = v_ant - 0.01
72      if (freno > 0.01 and v > 0.01):
73          v = v - (freno/5)
74
75      #Control de dirección.
76
77      if (ejes[0] > 0.05):
78          VelocidadL = v * MAX_SPEED * atras
79          VelocidadR = v * (1 - eje) * MAX_SPEED
80          * atras
81      elif (ejes[0] < -0.05):
82          VelocidadL = v * (1 - eje) * MAX_SPEED
83          * atras
84          VelocidadR = v * MAX_SPEED *
85      elif(ejes[0] > -0.0500 and ejes[0] < 0.0500):
86          VelocidadL = v * MAX_SPEED * atras
87          VelocidadR = v * MAX_SPEED * atras
88      L.setVelocity(VelocidadL)
89      R.setVelocity(VelocidadR)
```