



## GRADO EN INGENIERÍA DE ROBÓTICA SOFTWARE

Escuela Técnica Superior de Ingeniería de Telecomunicación

Curso académico 2021-2022

### Trabajo fin de grado

Escribe el título del trabajo aquí  
con la segunda línea aquí

**Tutor:** Julio Vega Pérez  
**Autor:** Álvaro Mariscal Ávila



Este trabajo se distribuye bajo los términos de la licencia internacional CC BY-NC-SA International License (Creative Commons AttributionNonCommercial-ShareAlike 4.0). Usted es libre de *(a) compartir*: copiar y redistribuir el material en cualquier medio o formato; y *(b) adaptar*: remezclar, transformar y crear a partir del material. El licenciador no puede revocar estas libertades mientras cumpla con los términos de la licencia:

- *Atribución.* Usted debe dar crédito de manera adecuada, brindar un enlace a la licencia, e indicar si se han realizado cambios. Puede hacerlo en cualquier forma razonable, pero no de forma tal que sugiera que usted o su uso tienen el apoyo de la licenciatante.
- *No comercial.* Usted no puede hacer uso del material con propósitos comerciales.
- *Compartir igual.* Si remezcla, transforma o crea a partir del material, debe distribuir su contribución bajo la misma licencia del original.

# Agradecimientos

---

Unas bonitas palabras...

Quizás un segundo párrafo esté bien. No te olvides de nadie.

Un tercero tampoco viene mal para contar alguna anécdota...

¿Alguien más? Aunque sean *actores* secundarios.

Un quinto párrafo como colofón.

*A alguien especial;  
si no, tampoco pasa nada*

Madrid, xx de xxxxxx de 20xx

*Tu nombre*

# Resumen

---

Escribe aquí el resumen del trabajo. Un primer párrafo para dar contexto sobre la temática que rodea al trabajo.

Un segundo párrafo concretando el contexto del problema abordado.

En el tercer párrafo, comenta cómo has resuelto la problemática descrita en el anterior párrafo.

Por último, en este cuarto párrafo, describe cómo han ido los experimentos.

# Acrónimos

---

**AERO** *Autonomous Exploration Rover*

**AI** *Artificial Intelligence*

**ANN** *Artificial Neural Network*

**API** *Application Programming Interface*

**HRI** *Human-Robot Interaction*

**AGV** *Automated guided vehicle*

**AMR** *Autonomous mobile robot*

# Índice general

---

<b>1. Introducción</b>	<b>1</b>
1.1. Coches autónomos . . . . .	1
1.2. AMRs . . . . .	2
1.3. Visión como sensor principal . . . . .	3
1.4. Arquitectura . . . . .	5
1.5. Deep Learning . . . . .	6
1.5.1. YOLO . . . . .	7
<b>2. Objetivos</b>	<b>9</b>
2.1. Descripción del problema . . . . .	9
2.2. Requisitos . . . . .	10
2.3. Metodología . . . . .	10
2.4. Plan de trabajo . . . . .	11
<b>3. Plataforma de desarrollo</b>	<b>12</b>
3.1. Hardware . . . . .	12
3.1.1. <i>NVIDIA Jetson Nano</i> . . . . .	12
3.1.2. Motores <i>TT</i> . . . . .	13
3.1.3. Controladora de motores <i>L298N</i> . . . . .	13
3.1.4. Cámara <i>Xiaomi</i> . . . . .	14
3.1.5. Batería 10000mAh . . . . .	14
3.1.6. Chasis . . . . .	15
3.2. Software . . . . .	15
3.2.1. <i>Python</i> . . . . .	15
3.2.2. <i>Blender</i> . . . . .	16
3.2.3. <i>Gazebo</i> . . . . .	16
3.2.4. <i>SDF</i> . . . . .	17
3.2.5. <i>ROS</i> . . . . .	17
3.2.6. <i>OpenCV</i> . . . . .	18

3.2.7. <i>Darknet</i> . . . . .	19
3.2.8. <i>PyQt</i> . . . . .	19
3.2.9. <i>JetRacer</i> . . . . .	20
<b>4. Diseño</b>	<b>22</b>
4.1. Entorno simulado . . . . .	22
4.1.1. Modelo de la ciudad . . . . .	22
4.1.2. Modelo del coche autónomo . . . . .	24
4.1.3. Seguimiento de carril . . . . .	30
4.1.4. Detección de objetos . . . . .	30
4.2. Entorno real . . . . .	30
4.3. Snippets . . . . .	30
4.4. Verbatim . . . . .	30
4.5. Ecuaciones . . . . .	31
4.6. Tablas o cuadros . . . . .	31
<b>5. Conclusiones</b>	<b>32</b>
5.1. Conclusiones . . . . .	32
5.2. Corrector ortográfico . . . . .	32
<b>Bibliografía</b>	<b>33</b>

# Índice de figuras

---

1.1.	<i>Tesla AutoPilot</i>	2
1.2.	<i>AMRs Kiva Systems</i> en almacenes de <i>Amazon</i>	3
1.3.	<i>PiCamera</i> usada en la placa <i>Raspberry Pi</i>	4
1.4.	<i>BMW's FIR-based Autoliv Night Vision System</i>	4
1.5.	Imagen de profundidad <i>Kinect</i>	5
1.6.	<i>Kinect</i> desarrollada por <i>Microsoft</i>	5
1.7.	<i>Raspberry Pi 4</i>	6
1.8.	<i>Jetson Nano</i>	6
1.9.	<i>LattePanda Alpha 864s</i>	6
1.10.	<i>AMD Zen</i> como CPU y <i>Navi 23</i> como GPU, usado en <i>Tesla Model S</i> [Ros, 2021]	6
1.11.	Objetos detectados por <i>YOLO</i> en una carretera	7
1.12.	Arquitectura <i>YOLOv3</i> con 53 capas convolucionales	8
3.1.	<i>NVIDIA Jetson Nano</i>	13
3.2.	Motores <i>TT</i>	13
3.3.	Controladora de motores <i>L298N</i>	14
3.4.	Cámara <i>Xiaomi</i>	14
3.5.	Batería 10000mAh	15
3.6.	Chasis	15
3.7.	Creación de una animación 3D con <i>Blender</i>	16
3.8.	Ciudad simulada en <i>Gazebo</i>	17
3.9.	Robot Spot de Boston Dynamics simulado en <i>Gazebo</i>	18
3.10.	Software de impresión 3D <i>Ultimaker Cura</i>	19
3.11.	Interfaz <i>notebook</i> para ajustar controlador P	21
4.1.	Modelo de la ciudad original y la ciudad modificada en <i>Gazebo</i>	23
4.2.	Semáforo y peatón usando plugins para <i>Gazebo</i>	24
4.3.	Diseño de las piezas 2D en <i>FreeCAD</i>	25

4.4. Diseño de las piezas 3D en <i>FreeCAD</i> . . . . .	25
4.5. Modelo 3D estático ensamblado en <i>Blender</i> . . . . .	26
4.6. Diagrama del modelo diseñado con <i>Phobos</i> . . . . .	27
4.7. Modelo dinámico del robot en <i>Gazebo</i> . . . . .	29

# Listado de códigos

---

4.1.	Definición de estados y duraciones del semáforo . . . . .	23
4.2.	Configuración de <i>waypoints</i> , velocidad y distancia a obstáculos del peatón	24
4.3.	Código para cargar el plugin <i>Gazebo ROS Control</i> . . . . .	28
4.4.	Código para crear cámara simulada en <i>Gazebo</i> . . . . .	28
4.5.	Definición de los controladores de los <i>joints</i> del robot . . . . .	29
4.6.	Función para buscar elementos 3D en la imagen . . . . .	30
4.7.	Cómo usar un Slider . . . . .	30

# Listado de ecuaciones

---

4.1. Ejemplo de ecuación con fracciones . . . . .	31
4.2. Ejemplo de ecuación con array y letras y símbolos especiales . . . . .	31

# Índice de cuadros

---

4.1. Parámetros intrínsecos de la cámara . . . . .	31
--	----

---

# **Capítulo 1**

## **Introducción**

---

En la actualidad los vehículos autónomos están en auge, cada vez tenemos más ejemplos de tareas típicamente realizadas por humanos que ya, es posible realizar sin un humano al mando. La conducción autónoma tiene el potencial para cambiar la forma en que nos movemos, aportando seguridad y comfort, si bien es cierto que aún no vemos vehículos circulando sin conductor, mucha de la tecnología necesaria para hacerlo ya está presente en los vehículos actuales. La conducción autónoma va mucho más allá de, únicamente los vehículos que transitan las ciudades, es aplicable a muchos otros ámbitos, por ejemplo entornos industriales, de inspección o incluso de exploración donde el vehículo se enfrenta a situaciones impredecibles y ante las que debe saber reaccionar correctamente.

### **1.1. Coches autónomos**

Cuando se nos viene a la cabeza el concepto de vehículo autónomo, se suele relacionar con coches autónomos, es decir, los vehículos que circulan a diario por las ciudades; coches, autobuses, furgonetas pero sin una persona al volante. Todavía no está presente en las ciudades pero en un corto plazo de tiempo lo estará, el principal inconveniente actual es la regulación, a diferencia de, por ejemplo, el entorno de la aviación, donde desde hace décadas el control de la aeronave es automático a excepción de tareas como el despegue, el aterrizaje o situaciones de emergencia.

Actualmente coches de última generación como *Tesla*, ya incorporan un grado de autonomía elevado en determinadas situaciones, pero siempre con un conductor al volante que debe permanecer atento para poder reaccionar.

Atendiendo al estándar *SAE J3016* los niveles de autonomía se pueden dividir en

Figura 1.1: *Tesla AutoPilot.*

cinco:

1. Sin automatización: avisos y asistencia puntualmente
2. Asistencia a la conducción: centrado de carril **o** control de crucero
3. Automatización parcial: centrado de carril **y** control de crucero
4. Automatización condicionada: conducción automática en atascos
5. Automatización elevada: conducción automática en algunas situaciones
6. Automatización completa: conducción automática en cualquier situación

[SAE, 2018]

## 1.2. AMRs

Los *AMRs* son robots móviles autónomos, capaces de navegar por entornos dinámicos, conviviendo con humanos a su alrededor y sabiendo sobreponerse a situaciones para las que no habían sido programados explícitamente. Son los sucesores de los *AGVs*, vehículos guiados automatizados, este tipo de vehículos requieren una cierta infraestructura dependiendo del tipo de guiado, ya sea filoguiados, a través de pintura o a través de cualquier otra técnica que haga que ese vehículo sólo pueda

funcionar cuando se sabe la infraestructura previa que estará presente en el entorno de trabajo.

Además, presentan muchas dificultades para relacionarse con obstáculos o humanos, donde ante un cambio pequeño del entorno, el robot se detendrá por seguridad. A diferencia de estos, los *AMRs*, son capaces de realizar multitud de tareas en entornos donde la infraestructura necesaria es casi nula, quizás tengan una serie de requisitos en cuanto a conectividad, pero con esa excepción, son robots que pueden ser diseñados para navegar por cualquier tipo de ambiente.

Un ejemplo muy representativo de *AMRs*, es *Kiva Systems*, empresa comprada por *Amazon* para automatizar sus almacenes en tareas de logística a nivel interno, maximizando la productividad y el almacenamiento, tanto en profundidad como en altura y minimizando el coste en personal.



Figura 1.2: *AMRs Kiva Systems* en almacenes de *Amazon*.

### 1.3. Visión como sensor principal

Gracias a un sensor como la cámara podemos obtener una información muy completa del entorno que rodea al robot y con ello poder actuar en consecuencia. La cámara se presenta como el sensor más interesante que puede equipar un robot, cuentan con un tamaño y peso muy reducido, como es el caso de la 1.3, además de un coste muy reducido. Sin embargo, presenta algunas dificultades cuando nos disponemos a tratar la imagen recibida; en cuanto a potencia del dispositivo, para procesar una imagen, y dependiendo de su resolución, es necesario un mínimo de requerimientos técnicos a nivel de *hardware*, para poder conseguir un procesamiento con un nivel adecuado de

*fps*, por otra parte, la imagen recibida deberá haber sido captada en un entorno con buenas condiciones lumínicas. Para ello existen diferentes tipos de sensores *EO/IR*, como por ejemplo, cámaras de visión nocturna, donde el espectro utilizado es *FIR*, con ello se consigue resaltar en la imagen lo que realmente es necesario.



Figura 1.3: *PiCamera* usada en la placa *Raspberry Pi*.

Un ejemplo es *BMW's FIR-based Autoliv Night Vision System* [Raiciu, 2009], trabaja con imágenes de *320x240* y cuenta con un rango de *300 metros*.



Figura 1.4: *BMW's FIR-based Autoliv Night Vision System*.

Cuando se realiza el procesamiento de una imagen, es interesante conocer la distancia, por ejemplo, a la que se encuentra un objeto en concreto, para ello existen soluciones para poder conocer dicha información con una **única** cámara tradicional, es el caso de *cite Open Vision System for Low-Cost Robotics Education* o [Dal, 2021], sin embargo, estos algoritmos se basan en la suposición de que todos los objetos se encuentran situados en un plano imaginario situado en el suelo. Para eliminar esta

restricción, y poder conocer distancias de objetos que no se encuentran situados en el suelo, surgen las *cámaras RGBD*, en las que cada píxel de la imagen proporciona, además del color RGB, una tercera componente llamada *depth*. Una de las primeras cámaras *RGBD* comerciales es la *Kinect* desarrollada por *Microsoft* para su consola *Xbox 360*.



Figura 1.5: Imagen de profundidad *Kinect*.

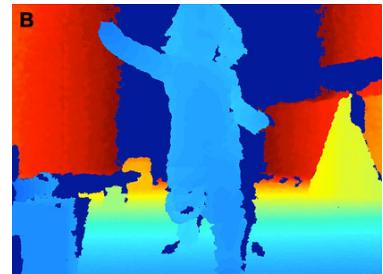


Figura 1.6: *Kinect* desarrollada por *Microsoft*.

La visión artificial ofrece multitud de posibilidades, no solo en robótica, tiene una gran de aplicaciones en campos tan dispares como la medicina, la realidad aumentada, el procesamiento de señales o la agricultura.

## 1.4. Arquitectura

La decisión de qué arquitectura utilizar en un robot es determinante. Es necesario evaluar multitud de factores; consumo de energía, potencia requerida, tamaño y peso, sistema operativo a utilizar, posibilidad de recibir respuesta en *real time*, precio etc...

Existen multitud de arquitecturas utilizadas en proyectos robóticos *x86*, *x86\_64*, *ARMv6*, *ARMv7* o *AArch64*. Pero, actualmente hay dos arquitecturas predominantes *x86\_64* y *AArch64*, ambas de *64 bits*, ya que los *32 bits* han quedado desfasados para la gran mayoría de aplicaciones, además, los nuevos sistemas operativos comienzan a no soportarlos. [J.Pomeyrol, 2019]

Estas dos arquitecturas tienen grandes diferencias en cuanto al diseño de los procesadores y las instrucciones que utilizan:

- Reduced Instruction Set Computer (*RISC*)

- Complex Instruction Set Computer (*CISC*)

El conjunto de instrucciones utilizado por *AArch64* es *CISC*, este conjunto se compone de gran cantidad de instrucciones y muchas de ellas complejas para realizar tareas que, el conjunto *RISC*, puede realizar con varias instrucciones. Este último conjunto es utilizado por la arquitectura *x86\_64*.

Ejemplos representativos de *AArch64*:

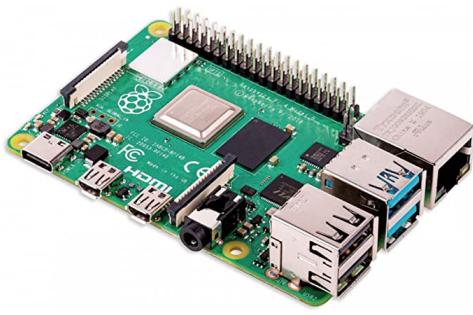


Figura 1.7: *Raspberry Pi 4*.



Figura 1.8: *Jetson Nano*.

Ejemplos representativos de *x86\_64*:



Figura 1.9: *LattePanda Alpha 864s*.



Figura 1.10: *AMD Zen* como CPU y *Navi 23* como GPU, usado en Tesla Model S [Ros, 2021].

## 1.5. Deep Learning

El *Deep Learning* se basa en redes neuronales que parten del *Machine Learning*, que a su vez surge de la *Inteligencia Artificial (IA)*. Sus inicios se remontan al año 1979 cuando *Kunihiko Fukushima* desarrolló una red neuronal de entre 5 y 6 capas llamada *neocognitrón*[Fukushima, 1979], con el objetivo de reconocer caracteres japoneses.

Este tipo de redes neuronales tiene multitud de aplicaciones, pero todas comparten grandes cantidades de datos, conocidos como *datasets*, en cualquier formato; vídeo, imagen, sonido. Algunas de ellas son: clasificación de objetos, procesamiento natural del lenguaje, *Big Data*, análisis médico, conversión de imágenes en blanco y negro a color etc...

### 1.5.1. YOLO

Uno de los algoritmos más populares, capaz de detectar y clasificar objetos provenientes de una imagen es *YOLO* (*You Only Look Once*). Sus principales ventajas son una gran precisión y la posibilidad, con el hardware adecuado, de ejecutar en tiempo real. Este algoritmo hace honor a su nombre y, por tanto, solo realiza una *propagación hacia delante* en cada ejecución.



Figura 1.11: Objetos detectados por *YOLO* en una carretera.

Se basa en el uso de redes neuronales convolucionales, *Convolutional neural network*.

Se diferencia de una red neuronal tradicional, en que la operación de multiplicación de matrices, se sustituye por una operación matemática llamada convolución, consistente en mezclar dos fuentes de información para producir una tercera, en este caso dos funciones.

*YOLO* hace uso, esencialmente, de tres técnicas para conseguir reconocer objetos:

- División de la imagen en celdas: de esta forma se pueden detectar multitud de objetos en una imagen
- Creación de *bounding boxes*: cajas 1.11 dibujando el contorno del objeto detectado y fijando la probabilidad de que el objeto detectado sea correcto
- Intersección sobre la unión: consiste en seleccionar el *bounding box* con mayor probabilidad cuando hay varios superpuestos

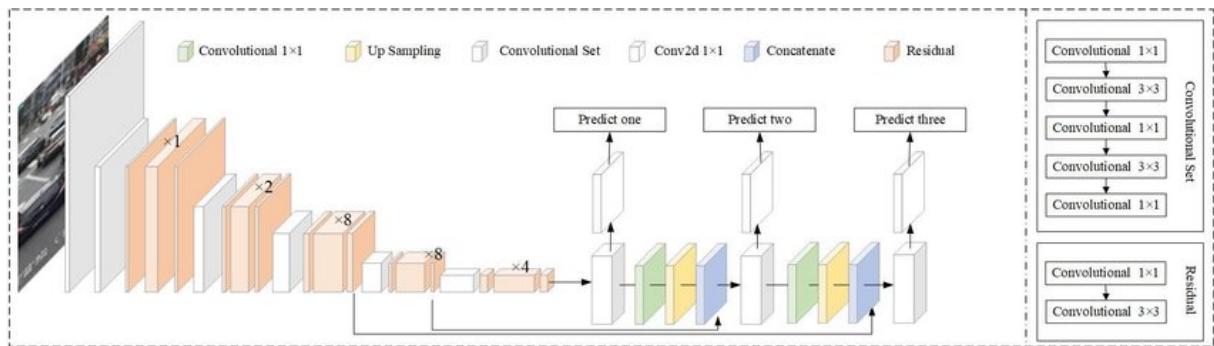


Figura 1.12: Arquitectura *YOLOv3* con 53 capas convolucionales.

Uno de los principales problemas de *YOLO* es la baja de probabilidad de detectar objetos de tamaño reducido, debido principalmente a la baja resolución de la red (está permitido incrementarla pero disminuye su rendimiento) que hace muy difícil su detección.

Además, existen versiones reducidas de este algoritmo como, Tiny-YOLO y Fast YOLO capaces de ser ejecutados en equipos de bajo coste y reducido tamaño.

En los siguientes capítulos trataremos los objetivos a cumplir ...

---

## Capítulo 2

# Objetivos

---

En este capítulo se detallan los objetivos del trabajo realizado, así como, los requisitos que este ha de cumplir, la metodología utilizada y el plan de trabajo seguido para completarlo.

### 2.1. Descripción del problema

El objetivo del trabajo es implementar un coche autónomo bajo una plataforma de bajo coste y reducido tamaño capaz de circular por un circuito o carretera en un entorno dinámico interactuando con objetos propios de una ciudad, como semáforos, señales de stop o peatones. Para ello se ha dividido el objetivo en general en dos subobjetivos:

- (a) Entorno simulado: utilizando el entorno de simulación *Gazebo* se desarrollará el problema anteriormente descrito con la finalidad de, posteriormente, realizar lo mismo en un entorno real.
- (b) Entorno real: utilizando un robot real diseñado a partir de la placa *NVIDIA Jetson Nano*, se desarrollará el problema anteriormente descrito sobre un circuito construido a partir de pistas de *Scalextric*<sup>1</sup>.

Como se dijo anteriormente, en ambos entornos, se ha de completar dos objetivos:

- (a) Seguimiento de carril: el coche autónomo tendrá que ser capaz de realizar un seguimiento del carril utilizando una red neuronal que indicará el centro del carril al que el robot deberá ceñirse.

---

<sup>1</sup><https://scalextric.es/>

- (b) Detección de objetos: mientras el robot realiza el seguimiento del carril debe interactuar con objetos del entorno, esto es, reaccionar cuando ve una señal de stop o un semáforo y rojo y detenerse apropiadamente. Esta detección deberá ser realizada, preferiblemente, en tiempo real.

## 2.2. Requisitos

El trabajo tendrá una serie de requisitos que deberán ser respetados:

1. El sistema operativo utilizado, para ambos entornos, será *GNU/Linux*, concretamente la distribución *Ubuntu 20.04 LTS* ya que cuenta con soporte para múltiples arquitecturas y proporciona un gran rendimiento.
2. El entorno simulado requerirá la presencia de una tarjeta gráfica dedicada ya que es muy recomendable para trabajar con redes neuronales; concretamente de la marca *NVIDIA*, ya que se utilizará la plataforma *CUDA*.
3. El entorno real requerirá un robot con la placa de desarrollo *NVIDIA Jetson Nano* debido a ser una de las placas con *GPU* más económicas.
4. El lenguaje de programación utilizado será *Python* debido a las librerías utilizadas, que se detallarán en el próximo capítulo.

## 2.3. Metodología

Partiendo de los requisitos y objetivos previamente descritos, se procedió a evaluar el hardware necesario; a continuación, se realizó un análisis de diversas bibliotecas de código con el objetivo de seleccionar las que fuesen compatibles y tuviesen un mejor rendimiento en la plataforma de hardware elegida. El siguiente paso fue el diseño del software necesario y cómo integrarlo con las bibliotecas escogidas. Por último se realizaron pruebas periódicas tanto en simulador como en un entorno real, con el objetivo de ir afinando el software para conseguir el resultado final.

## 2.4. Plan de trabajo

El plan de trabajo se ha basado en reuniones semanales o quincenales con el tutor, dependiendo de la carga de trabajo, en las que se iban fijando objetivos específicos y fijando la estrategia para poder completar el proyecto.

Todo el trabajo realizado se ha ido subiendo a un repositorio de trabajo en *GitHub*<sup>2</sup>. De esta forma se podía analizar de forma muy rápida los cambios realizados en el código. También se ha ido desarrollando una *Wiki*<sup>3</sup> en *GitHub* a modo de bitácora en la que se iban detallando los avances del proyecto, así como, los problemas y las limitaciones que iban surgiendo a medida que se cumplían los objetivos.

```
aspell –encoding utf-8 –lang=es –mode=tex check capítulos/capítulo2.tex Okk
```

---

<sup>2</sup><https://github.com/jmvega/tfg-amariscal>

<sup>3</sup><https://github.com/jmvega/tfg-amariscal/wiki>

---

# Capítulo 3

# Plataforma de desarrollo

---

En este capítulo, se explica el hardware y software elegido para desarrollar el trabajo y los motivos de dicha elección.

## 3.1. Hardware

### 3.1.1. *NVIDIA Jetson Nano*

La placa de desarrollo *NVIDIA Jetson Nano*<sup>1</sup> es una plataforma de bajo coste con grandes capacidades computacionales para implementar técnicas de inteligencia artificial gracias a su *GPU* dedicada *NVIDIA Maxwell*<sup>2</sup> con 128 *NVIDIA CUDA cores*<sup>3</sup>, además dispone de una CPU Quad-core basada en la arquitectura Aarch64 lo que permite ejecutar GNU/Linux sin dificultades y ser compatible con numerosas bibliotecas de código. La placa en cuestión, dispone además de pines GPIO lo que permite de forma muy sencilla conectar todo tipo de sensores y actuadores.

Los requisitos en lo que a alimentación se refiere no son excesivos, requiere un mínimo de 5 voltios (V) y 3 amperios (A), lo que permite que una simple *powerbank* de reducido tamaño sea capaz de alimentar la placa, si bien es cierto que la batería debe poder ofrecer tres amperios de forma estable, y no solo como intensidad pico. Existen numerosos proyectos en los que esta placa está presente, tales como JetBot<sup>4</sup> o JetRacer3.2.9.

---

<sup>1</sup><https://developer.nvidia.com/embedded/jetson-nano>

<sup>2</sup><https://developer.nvidia.com/maxwell-compute-architecture>

<sup>3</sup><https://developer.nvidia.com/cuda-gpus>

<sup>4</sup><https://github.com/NVIDIA-AI-IOT/jetbot>



Figura 3.1: *NVIDIA Jetson Nano*.

### 3.1.2. Motores *TT*

Se trata de unos motores<sup>5</sup> de corriente continua con reductora utilizados en la multitud de proyectos de robótica de muy bajo coste<sup>67</sup>. La tensión de alimentación tiene un rango de 3 a 6 voltios y la velocidad mínima en vacío tiene un rango de 90 a 200 revoluciones por minuto (RPM) dependiendo del voltaje, lo que permite conseguir una velocidad reducida para robots de pequeño tamaño.



Figura 3.2: Motores *TT*.

### 3.1.3. Controladora de motores *L298N*

Es un módulo<sup>8</sup> capaz de controlar la dirección y la velocidad de los motores anteriormente citados. La tensión de alimentación es de un mínimo de 6 voltios, lo que hace imposible alimentarla con la placa *NVIDIA Jetson Nano*<sup>3.1</sup> por lo que es necesario una batería externa. Otra posibilidad para utilizar una única batería sería utilizar la salida de 5 voltios (V) que nos ofrece la controladora, sin embargo, dicha salida nunca ofrecerá los 3 amperios (A) requeridos. Este componente permite invertir el sentido de la corriente lo que proporciona un control para mover los motores en el sentido de las agujas del reloj (CW) y en el sentido contrario a las agujas del reloj (CCW). La

<sup>5</sup><https://www.verical.com/datasheet/adafruit-brushless-dc-motors-3777-5912007.pdf>

<sup>6</sup><https://github.com/grimmp/tt-motor-mounting>

<sup>7</sup><https://github.com/bhabegger/diy-telepresence-robot>

<sup>8</sup><https://www.luisllamas.es/arduino-motor-corriente-continua-1298n/>

principal limitación de esta placa es que solo permite controlar dos motores, por lo que si se dispone de 4 motores, se podrán conectar a pares dependiendo del comportamiento deseado. El control se realiza a través de la técnica PWM<sup>9</sup>, que permite enviar de forma precisa la velocidad deseada a través de una señal digital.

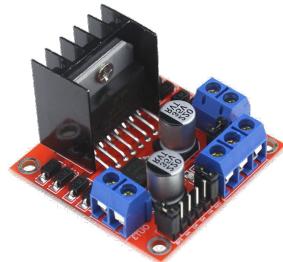


Figura 3.3: Controladora de motores *L298N*.

### 3.1.4. Cámara *Xiaomi*

Se trata de una cámara USB3.4 lo que permite recibir la imagen a través de dicho puerto con una tasa de 30 *frames* por segundo (FPS). Su resolución es 1080p<sup>10</sup> pero permite obtener una imagen de menor resolución a través de su driver.



Figura 3.4: Cámara *Xiaomi*.

### 3.1.5. Batería 10000mAh

Su capacidad es de 10000 miliamperios hora (mAh), el voltaje de funcionamiento es 5 voltios (V) y su intensidad teórica es 3 amperios (A) por lo que permite alimentar

<sup>9</sup><https://circuitdigest.com/tutorial/what-is-pwm-pulse-width-modulation>

<sup>10</sup><https://xiaomiplanets.com/xiaovv-6320s-webcam-5/>

la placa *NVIDIA Jetson Nano*3.1.



Figura 3.5: Batería 10000mAh.

### 3.1.6. Chasis

Se trata de un chasis<sup>3.6</sup> de bajo coste muy común en proyectos relacionados con Arduino. Dispone de soportes para los Motores TT3.2, lo que permite ensamblarlos de forma muy sencilla. El tamaño es suficiente para alojar todos los componentes elegidos utilizando los agujeros predefinidos en el chasis.



Figura 3.6: Chasis.

## 3.2. Software

### 3.2.1. Python

*Python* es un lenguaje de programación de código abierto, interpretado, orientado a objetos y de alto nivel. En la actualidad es el lenguaje más usado a nivel mundial<sup>11</sup>.

---

<sup>11</sup><https://pypl.github.io/PYPL.html>

Está considerado como un lenguaje fácil de aprender gracias a su simple sintaxis, esto le ha llevado a crecer mucho en popularidad durante los últimos años. Además, al ser interpretado, no es necesario utilizar un compilador, lo que provoca un desarrollo mucho más rápido. Python cuenta con una enorme cantidad de bibliotecas y clientes para utilizar software desarrollado en otros lenguajes. Algunas de ellas se describirán a continuación.

### 3.2.2. *Blender*

*Blender*<sup>12</sup> es una plataforma de código abierto dedicada a la creación, simulación, renderizado y animación de modelos 3D con todo tipo de texturas, sombras etc.. Es un desarrollo de la Blender Foundation y está escrito principalmente en *C*. Permite crear diseños de robots desde cero de una forma relativamente rápida y con *addons* como *Phobos*, es posible exportar el modelo a *URDF* con el fin de realizar una simulación[von Szadkowski and Reichel, 2020].



Figura 3.7: Creación de una animación 3D con *Blender*.

### 3.2.3. *Gazebo*

*Ignition Gazebo*<sup>13</sup> es un simulador 3D de código abierto desarrollado por la *Open Source Robotics Foundation (OSRF)*<sup>14</sup>, escrito en *C++*, usado principalmente para

<sup>12</sup><https://www.blender.org/>

<sup>13</sup><https://github.com/gazebosim/gz-sim>

<sup>14</sup><https://www.openrobotics.org/>

simular comportamientos con gran precisión y gráficos de alta calidad en los que intervienen robots en un entorno dinámico<sup>4.1</sup>. Desde Utiliza, por defecto, el motor de físicas *ODE*<sup>15</sup>, escrito también en *C++*. Permite simular todo tipo de sensores y actuadores. Además, ofrece integración con ROS de forma muy sencilla.



Figura 3.8: Ciudad simulada en *Gazebo*.

### 3.2.4. *SDF*

El formato *SDF* (*Simulation Description Format*)<sup>16</sup> o *SDFormat* permite describir entornos, objetos dinámicos o robots de una manera sencilla. Está basado en *XML* y escrito en *C++*. El objetivo de este formato es ejecutar comportamientos en un simulador. Originalmente fue desarrollado como parte de *Gazebo* por lo que existen numerosas bibliotecas donde se encuentran modelos o mundos desarrollados en *SDF* para este simulador<sup>17</sup>. Cabe destacar también, el formato *URDF* que, a diferencia de *SDF*, únicamente puede describir un objeto o robot, pero no el mundo en el que vive<sup>18</sup>.

### 3.2.5. *ROS*

*Robot Operating System ROS*<sup>20</sup> es un middleware robótico, de código abierto, con multitud de bibliotecas y herramientas desarrollado por la *Open Source Robotics*

<sup>15</sup><https://bitbucket.org/odedevs/ode/src/master/>

<sup>16</sup><https://github.com/gazebosim/sdformat>

<sup>17</sup><https://github.com/HuyPhamG/simulatedswarm>

<sup>18</sup><https://newscrewdriver.com/2018/07/31/ros-notes-urdf-vs-gazebo-sdf/>

<sup>20</sup><https://ros.org/>



Figura 3.9: Robot Spot de Boston Dynamics simulado en *Gazebo*.

19

*Foundation (OSRF)*<sup>21</sup>, escrito en *C++* y *Python*. Es considerado actualmente el estándar en robótica. Permite desarrollar aplicaciones complejas en las intervienen diversos procesos llamados nodos<sup>22</sup>, que se comunican entre ellos mediante *topics*<sup>23</sup> y servicios<sup>24</sup>. Una de las principales ventajas de utilizar un middleware como ROS es la capacidad de abstracción que proporciona, de forma que el usuario únicamente programa sobre una interfaz<sup>25</sup> dada, disponible en *C++* y *Python*, sin preocuparse por lo que pasa por debajo.

### 3.2.6. *OpenCV*

*OpenCV*<sup>26</sup> es una librería de visión artificial de código abierto desarrollado por Intel<sup>27</sup>. Está escrita en *C/C++* y cuenta con soporte para aceleración por GPU basadas en CUDA<sup>28</sup> y OpenCL<sup>29</sup> y procesamiento de imagen en tiempo real. Es usada en todo tipo de aplicaciones en las que interviene la visión por ordenador, tales como, detección de objetos, realidad aumentada o reconocimiento de gestos. Además, está disponible en multitud de lenguajes de programación; *C++*, *Python*, *Java*, etc...

---

<sup>21</sup><https://www.openrobotics.org/>

<sup>22</sup><http://wiki.ros.org/Nodes>

<sup>23</sup><http://wiki.ros.org/Topics>

<sup>24</sup><http://wiki.ros.org/Services>

<sup>25</sup><http://wiki.ros.org/Client20Libraries>

<sup>26</sup><https://github.com/opencv/opencv>

<sup>27</sup><https://opencv.org/opencv-platinum-membership/>

<sup>28</sup><https://developer.nvidia.com/cuda-zone>

<sup>29</sup><https://www.khronos.org/opencl/>

### 3.2.7. *Darknet*

*Darknet*<sup>30</sup> es un framework de código abierto que permite ejecutar y entrenar redes neuronales en tiempo real, ya que soporta tanto computación por CPU como GPU. Está escrito en C y CUDA, gracias a estar escrito en un lenguaje considerado de bajo nivel, ofrece un rendimiento aceptable en plataformas de bajo coste como *NVIDIA Jetson Nano*3.1.

### 3.2.8. *PyQt*

*PyQt*<sup>31</sup> es una plataforma de código abierto que permite crear interfaces gráficas (*GUI*) con el framework *Qt* utilizando *Python*, lo que simplifica mucho el desarrollo. Existen multitud de aplicaciones, desde un navegador<sup>32</sup> controlado únicamente con el teclado al estilo *VIM*<sup>33</sup>, hasta un software de impresión 3D como *Cura*3.10.

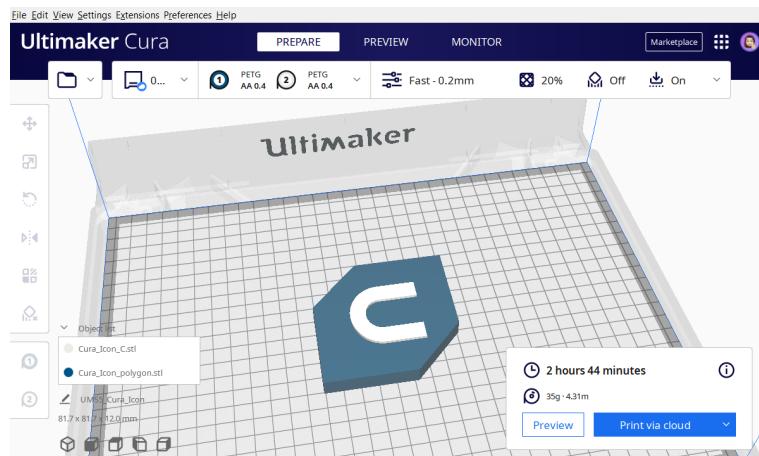


Figura 3.10: Software de impresión 3D *Ultimaker Cura*.

<sup>30</sup><https://pjreddie.com/darknet/>

<sup>31</sup><https://pythonpyqt.com/what-is-pyqt/>

<sup>32</sup><https://github.com/qutebrowser/qutebrowser>

<sup>33</sup><https://github.com/vim/vim>

### 3.2.9. *JetRacer*

La biblioteca *JetRacer*<sup>34</sup> permite entrenar una red neuronal para seguir un circuito o una ruta determinada. Utiliza *notebooks* de *Jupyter* para poder reducir la complejidad en el entrenamiento de la red y en el ajuste del controlador<sup>3.11</sup>.

A su vez esta librería utiliza otras tres cruciales para poder implementar su software:

- PyTorch<sup>35</sup>: es una librería de código abierto con multitud de herramientas para implementar algoritmos de *Deep Learning* basada en *Python*[Today, 2021]. Es un desarrollo de *Facebook's AI Research Lab*. Soporta aceleración por GPU, lo que es esencial para poder ejecutar redes neuronales. Junto a *Tensorflow* y *Keras*, son los tres *frameworks* de referencia en lo que a *Deep Learning* se refiere.
- PyTorch to TensorRT<sup>36</sup>: permite convertir modelos de PyTorch a modelos optimizados aprovechando los tensores de las gráficas dedicadas y realizando inferencia utilizando operaciones con *FP16* y *FP32*.
- Torchvision<sup>37</sup>: contiene multitud de *datasets*, modelos preentrenados y algoritmos relacionados con el procesamiento de imagen.

Mapa

<http://wiki.ros.org/Client>

Referencias del capítulo 2 al 3?

h! h!! t!, t!!, T! no compila

---

<sup>34</sup><https://github.com/NVIDIA-AI-IOT/jetracer>

<sup>35</sup><https://github.com/pytorch/pytorch>

<sup>36</sup><https://github.com/NVIDIA-AI-IOT/torch2trt>

<sup>37</sup><https://github.com/pytorch/vision>

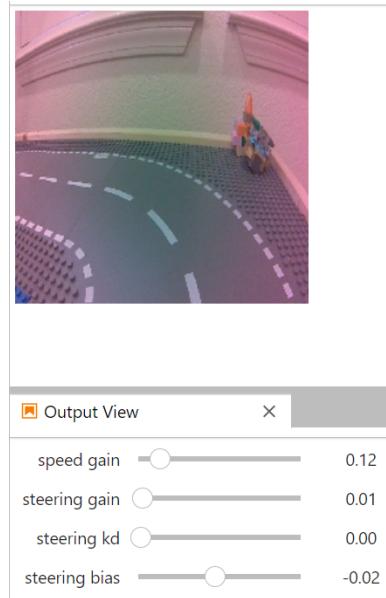


Figura 3.11: Interfaz *notebook* para ajustar controlador P.

---

# Capítulo 4

## Diseño

---

En este capítulo se describe el trabajo realizado y los experimentos llevados a cabo para validar el desarrollo.

El trabajo, como se explicó en el capítulo 2, se ha dividido en dos fases; primero se realiza una simulación del coche autónomo en una ciudad simplificada, para a continuación, reproducir ese mismo escenario en un entorno real.

### 4.1. Entorno simulado

Para poder simular el comportamiento deseado en un entorno simulado es necesario disponer de dos elementos; un modelo del coche autónomo y un mundo dinámico con el que el vehículo interactuará, todo ello ejecutado dentro de un simulador, en este caso, como se explicó anteriormente, en *Gazebo*.

#### 4.1.1. Modelo de la ciudad

Existen numerosos mundos diseñados para Gazebo disponibles en repositorios de GitHub<sup>12</sup>. En el caso que se plantea, es necesario una ciudad, por lo que partiendo de una ciudad<sup>3</sup> de gran tamaño se reduce su tamaño con el objetivo de, posteriormente, poder reproducir ese escenario en el mundo real. En la figura 4.1 se observan ambas ciudades.

---

<sup>1</sup>[https://github.com/chaolmu/gazebo\\_models\\_worlds\\_collection](https://github.com/chaolmu/gazebo_models_worlds_collection)

<sup>2</sup><https://github.com/mlherd/Dataset-of-Gazebo-Worlds-Models-and-Maps>

<sup>3</sup>[https://github.com/chaolmu/gazebo\\_models\\_worlds\\_collection/blob/master/worlds/small\\_city.world](https://github.com/chaolmu/gazebo_models_worlds_collection/blob/master/worlds/small_city.world)



Figura 4.1: Modelo de la ciudad original y la ciudad modificada en *Gazebo*.

El modelo de esta ciudad es estático, es decir, no contiene elementos dinámicos, por lo que es necesario añadir dos tipos de elementos; un semáforo y un peatón que cruce un paso de peatones. Para ello, es necesario añadir dos plugins para *Gazebo*. En primer lugar, y usando un modelo *SDF* que integra el *plugin* de cambio de color en el semáforo<sup>4</sup>. Dicho plugin tiene asociado un fichero de configuración en formato YAML, código 4.1, en el que se especifica la secuencia de estados del semáforo y su duración en el tiempo.

---

```
light_sequence:
- { color: "green", duration: 10, flashing: false }
- { color: "yellow", duration: 1, flashing: false }
- { color: "red", duration: 50, flashing: false }
```

---

Código 4.1: Definición de estados y duraciones del semáforo

El segundo plugin proporciona movimiento a un modelo de un humano llamado actor. Está disponible en otro repositorio<sup>5</sup> de GitHub y permite que el humano se desplace en el mundo de forma realista. El plugin citado también permite configuración para especificar la ruta de puntos o *waypoints* que el peatón ha de seguir, así como, su velocidad o la distancia mínima a la que debe situarse respecto a un obstáculo. Dicha configuración se realiza directamente en el fichero SDF, como muestra el código 4.2.

Ambos modelos se sitúan en una paso de peatones, creado a partir de láminas blancas, produciendo el resultado de la figura 4.2.

---

<sup>4</sup>[https://github.com/robustify/gazebo\\_traffic\\_light](https://github.com/robustify/gazebo_traffic_light)

<sup>5</sup><https://github.com/BruceChanJianLe/gazebo-plugin-autonomous-actor>

---

```

<actor name="actor">
  [...]
  <plugin name="trajectory" filename="libTrajectoryActorPlugin.so">
    <target>
      2.4028 -6.9143 1.1 1.570796 -0.0 3.141593
    </target>
    <target>
      2.4028 6.6816 1.1 1.570796 -0.0 3.141593
    </target>
    <velocity>0.75</velocity>
    <obstacle_margin>1.5</obstacle_margin>
    <obstacle></obstacle>
  </plugin>
</actor>

```

---

Código 4.2: Configuración de *waypoints*, velocidad y distancia a obstáculos del peatón



Figura 4.2: Semáforo y peatón usando plugins para *Gazebo*.

#### 4.1.2. Modelo del coche autónomo

El primer paso para desarrollar un modelo es diseñar las piezas a utilizar, para ello, como se explicó en el capítulo 2, se utilizará *FreeCAD*. El segundo paso es ensamblar el robot con *Blender* utilizando el *add-on Phobos* para definir los links y joints del robot con el objetivo de dotar al robot de movimiento, generando un fichero *URDF*. Por último, es necesario modificar el *URDF* para añadir cámaras y controladores para los motores que serán usados en el simulador *Gazebo*.

### Diseño de las piezas en *FreeCAD*

A través de la herramienta *Sketcher*<sup>6</sup>, disponible en *FreeCAD*, es posible realizar un diseño de piezas en 2D con restricciones de horizontalidad, verticalidad, igualdad entre rectas, medidas, ángulos etc y después convertir dicha pieza en 3D proporcionando un volumen a la pieza. Utilizando dicha herramienta se diseña el chasis y las ruedas del vehículo tomando las medidas reales, lo que da como resultado el diseño de las piezas en 2D de la figura 4.3.

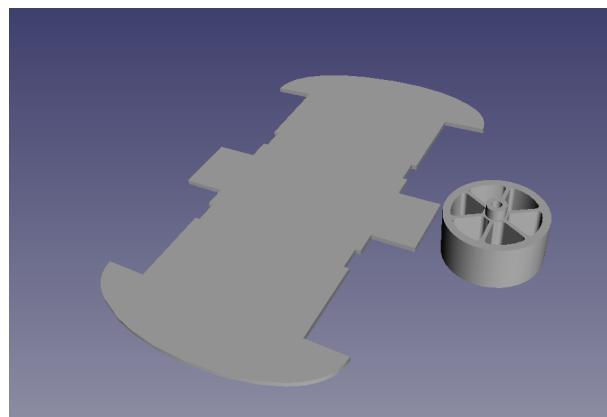


Figura 4.3: Diseño de las piezas 2D en *FreeCAD*.

A continuación se proporciona una anchura al chasis de 3mm y a las ruedas de xmm, obteniendo el modelo 3D de la figura 4.4.

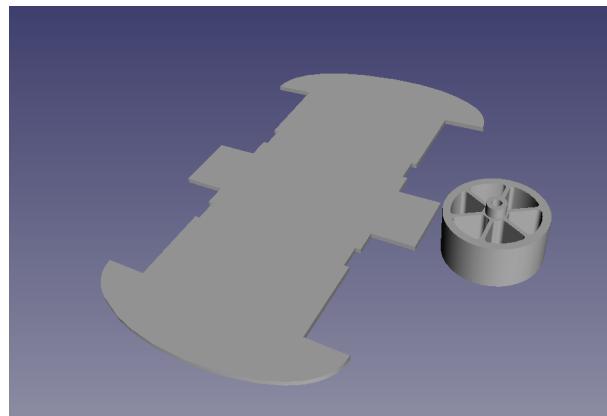


Figura 4.4: Diseño de las piezas 3D en *FreeCAD*.

---

<sup>6</sup>[https://wiki.freecadweb.org/Sketcher\\_Workbench](https://wiki.freecadweb.org/Sketcher_Workbench)

El siguiente paso es exportar las piezas diseñadas en formato *STL* para utilizarlas en *Blender*.

### Ensamblado del robot en *Blender*

Una vez importadas las piezas diseñadas en *FreeCAD*, se crean dos unidades del chasis y cuatro ruedas, a continuación se mueven y rotan en el espacio para situarlas correctamente. Con el objetivo de dotar de mayor realismo al modelo, se importan modelos de los motores y la placa utilizados, disponibles en y en. Y se diseñan dos piezas para emular la batería y la cámara del robot. Con todo ello se conforma el modelo estático del robot, figura 4.5.

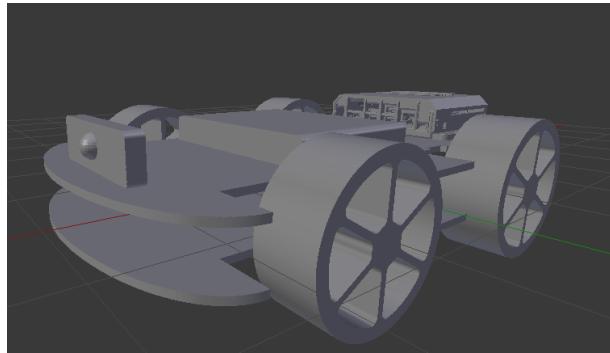


Figura 4.5: Modelo 3D estático ensamblado en *Blender*.

Este modelo es estático, es decir, no es posible simular un movimiento. Para ello es necesario un *add-on* para *Blender* como *Phobos*, que como se explicó anteriormente, permite crear *links* y *joints* y definir una jerarquía entre ellos.

En primer lugar, es necesario definir los elementos visuales que compondrán el modelo, en este caso serán todas las piezas del robot, dichas piezas serán del tipo *mesh*, ya que no son objetos primitivos, como cajas o esferas.

A continuación, se define un *link* principal que será el cuerpo del robot, de este *link* dependerán las cuatro ruedas del vehículo. Con el objetivo de simular colisiones, se crea un modelo de colisión de cada una de las piezas y un modelo inercial, que estará relacionado con el peso simulado de cada pieza.

Por último, se definen los *joints* de cada rueda. A través de esta funcionalidad es posible aplicar fuerza o velocidad a las ruedas del robot. Existen diferentes tipos de *joints* en *Phobos*; *fixed*, *revolute*, *continuous*, *prismatic*, en este caso, son necesarios *joints* de tipo *continuous*, ya que las ruedas realizan un giro sin un límite fijado, y no un movimiento transversal.

Una vez definidos todos los elementos del modelo y su jerarquía, en el menú lateral de *Blender* se encontrará el diagrama del modelo, figura 4.6.

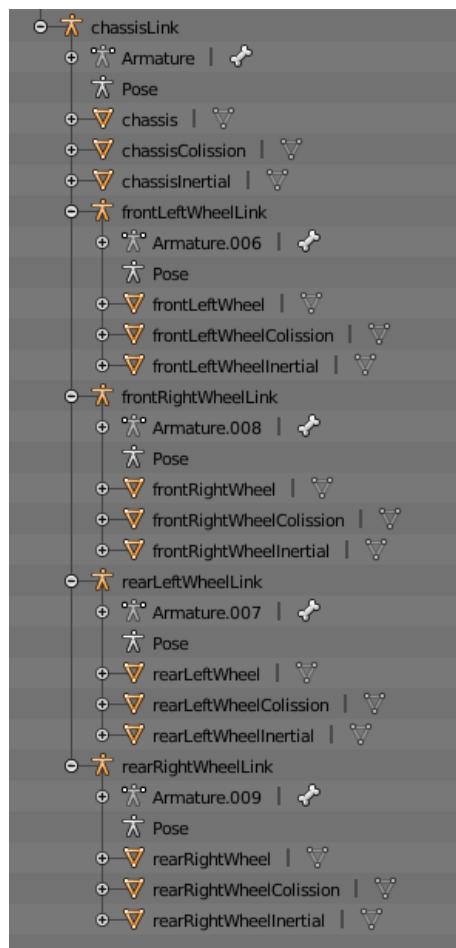


Figura 4.6: Diagrama del modelo diseñado con *Phobos*.

El siguiente paso es exportar el modelo a formato *URDF*, que como se explicó anteriormente, es el aceptado por *Gazebo* para simular modelos de robots. Las mallas, que componen el modelo visual y de colisiones, se exportan en formato *DAE*, y se cargan desde el modelo *URDF*.

## Adición de plugins para Gazebo

Con el objetivo de mover el robot utilizando el middleware ROS, es necesario cargar el plugin *Gazebo ROS Control* mediante el código 4.3.

---

```
<gazebo>
  <plugin name="gazebo_ros_control" filename="libgazebo_ros_control.so">
    <robotNamespace>/autonomous_vehicle</robotNamespace>
  </plugin>
</gazebo>
```

---

Código 4.3: Código para cargar el plugin *Gazebo ROS Control*

Otro elemento a añadir en el fichero URDF, es la existencia de dos cámaras; una cámara onboard y otra situada encima del robot que proporcionará una visión de la ruta realizada. Dichas cámaras simuladas publicarán su imagen en un topic de ROS con una resolución y un formato fijado mediante el código 4.4.

---

```
<gazebo reference="onboardCameraLink">
  <sensor name='cam_sensor' type='camera'>
    [...]
    <camera name='onboardCameraLink'>
      <horizontal_fov>1.570000</horizontal_fov>
      <image>
        <width>320</width>
        <height>240</height>
        <format>R8G8B8</format>
      </image>
      [...]
    </camera>
    [...]
  </sensor>
</gazebo>
```

---

Código 4.4: Código para crear cámara simulada en *Gazebo*

Para mover el robot simulado también será necesario un fichero *YAML*, en el que se especifiquen los controladores de los *joints* y el nombre de los *topics* en los que se publicarán los mensajes mediante el código 4.5.

Con todo ello, se implementa un *launcher*<sup>7</sup> que lanza el modelo dinámico del robot

---

<sup>7</sup><https://github.com/jmvega/tfg-amariscal/src/launcher.....>

---

```
autonomous_vehicle:  
joint_state_controller:  
    type: joint_state_controller/JointStateController  
    publish_rate: 10  
front_right_wheel_velocity_controller:  
    type: velocity_controllers/JointVelocityController  
    joint: frontRightWheelJoint  
front_left_wheel_velocity_controller:  
    type: velocity_controllers/JointVelocityController  
    joint: frontLeftWheelJoint  
rear_right_wheel_velocity_controller:  
    type: velocity_controllers/JointVelocityController  
    joint: rearRightWheelJoint  
rear_left_wheel_velocity_controller:  
    type: velocity_controllers/JointVelocityController  
    joint: rearLeftWheelJoint
```

---

Código 4.5: Definición de los controladores de los *joints* del robot

y se muestra en la ciudad simulada, figura 4.7.

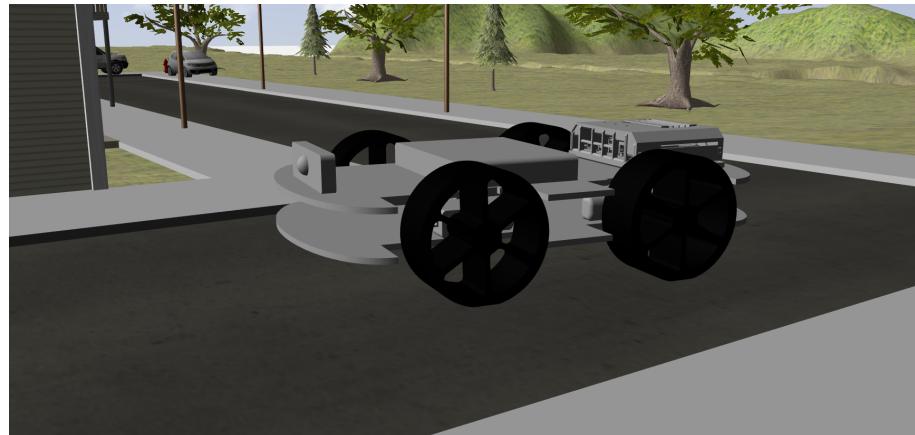


Figura 4.7: Modelo dinámico del robot en *Gazebo*.

### 4.1.3. Seguimiento de carril

### 4.1.4. Detección de objetos

## 4.2. Entorno real

## 4.3. Snippets

Puede resultar interesante, para clarificar la descripción, mostrar fragmentos de código (o *snippets*) ilustrativos. En el Código 4.6 vemos un ejemplo escrito en C++.

---

```
void Memory::hypothesizeParallelograms () {
    for(it1 = this->controller->segmentMemory.begin(); it1++) {
        squareFound = false; it2 = it1; it2++;
        while ((it2 != this->controller->segmentMemory.end()) && (!squareFound))
        {
            if (geometry::haveACommonVertex((*it1), (*it2), &square)) {
                dist1 = geometry::distanceBetweenPoints3D ((*it1).start, (*it1).end);
                dist2 = geometry::distanceBetweenPoints3D ((*it2).start, (*it2).end);
            }
        // [...]
```

---

Código 4.6: Función para buscar elementos 3D en la imagen

En el Código 4.7 vemos un ejemplo escrito en Python.

---

```
def mostrarValores():
    print (w1.get(), w2.get())

master = Tk()
w1 = Scale(master, from_=0, to=42)
w1.pack()
w2 = Scale(master, from_=0, to=200, orient=HORIZONTAL)
w2.pack()
Button(master, text='Show', command=mostrarValores).pack()

mainloop()
```

---

Código 4.7: Cómo usar un Slider

## 4.4. Verbatim

Para mencionar identificadores usados en el código —como nombres de funciones o variables— en el texto, usa el entorno literal o verbatim

`hypothesizeParallelograms()`. También se puede usar este entorno para varias líneas, como se ve a continuación:

```
void Memory::hypothesizeParallelograms () {
    // add your code here
}
```

## 4.5. Ecuaciones

Si necesitas insertar alguna ecuación, puedes hacerlo. Al igual que las figuras, no te olvides de referenciarlas. A continuación se exponen algunas ecuaciones de ejemplo: Ecuación 4.1 y Ecuación 4.2.

$$H = 1 - \frac{\sum_{i=0}^N \frac{(\frac{d_{js} + d_{je}}{2})}{N}}{M} \quad (4.1)$$

Ecuación 4.1: Ejemplo de ecuación con fracciones

$$v(\text{entrada}) = \begin{cases} 0 & \text{if } \epsilon_t < 0,1 \\ K_p \cdot (T_t - T) & \text{if } 0,1 \leq \epsilon_t < M_t \\ K_p \cdot M_t & \text{if } M_t < \epsilon_t \end{cases} \quad (4.2)$$

Ecuación 4.2: Ejemplo de ecuación con array y letras y símbolos especiales

## 4.6. Tablas o cuadros

Si necesitas insertar una tabla, hazlo dignamente usando las propias tablas de LATEX, no usando pantallazos e insertándolas como figuras... En el Cuadro 4.1 vemos un ejemplo.

Parámetros	Valores
Tipo de sensor	Sony IMX219PQ[7] CMOS 8-Mpx
Tamaño del sensor	3.674 x 2.760 mm (1/4"format)
Número de pixels	3280 x 2464 (active pixels)
Tamaño de pixel	1.12 x 1.12 um
Lente	f=3.04 mm, f/2.0
Ángulo de visión	62.2 x 48.8 degrees
Lente SLR equivalente	29 mm

Cuadro 4.1: Parámetros intrínsecos de la cámara

---

# Capítulo 5

# Conclusiones

---

Escribe aquí un párrafo explicando brevemente lo que vas a contar en este capítulo, que básicamente será una recapitulación de los problemas que has abordado, las soluciones que has prouesto, así como los experimentos llevados a cabo para validarlos. Y con esto, cierras la memoria.

## 5.1. Conclusiones

Enumera los objetivos y cómo los has cumplido.

Enumera también los requisitos implícitos en la consecución de esos objetivos, y cómo se han satisfecho.

No olvides dedicar un par de párrafos para hacer un balance global de qué has conseguido, y por qué es un avance respecto a lo que tenías inicialmente. Haz mención expresa de alguna limitación o peculiaridad de tu sistema y por qué es así. Y también, qué has aprendido desarrollando este trabajo.

Por último, añade otro par de párrafos de líneas futuras; esto es, cómo se puede continuar tu trabajo para abarcar una solución más amplia, o qué otras ramas de la investigación podrían seguirse partiendo de este trabajo, o cómo se podría mejorar para conseguir una aplicación real de este desarrollo (si es que no se ha llegado a conseguir).

## 5.2. Corrector ortográfico

Una vez tengas todo, no olvides pasar el corrector ortográfico de L<sup>A</sup>T<sub>E</sub>Xa todos tus ficheros *.tex*. En Windows, el propio editor TeXworks incluye el corrector. En Linux, usa **aspell** ejecutando el siguiente comando en tu terminal:

```
aspell --lang=es --mode=tex check capitulo1.tex
```

# Bibliografía

---

- [Dal, 2021] Dal, A. (2021). Distance(webcam) estimation with single-camera opencv-python.
- [Fukushima, 1979] Fukushima, K. (1979). Neocognitron: A self-organizing neural network model for a mechanism of pattern recognition unaffected by shift in position. *NHK Broadcasting Science Research Laboratories*.
- [J.Pomeyrol, 2019] J.Pomeyrol (2019). Canonical concreta el soporte de 32-bit para ubuntu 20.04 lts. Muy Linux.
- [Raiciu, 2009] Raiciu, T. (2009). How night vision works. *autoevolution.com*.
- [Ros, 2021] Ros, I. (2021). Tesla monta cpus zen de amd y gpus rdna 2 en sus coches.
- [SAE, 2018] SAE, I. (2018). Sae j3016 levels of driving automation. *SAE J3016*.
- [Today, 2021] Today, C. (2021). Why tesla using pytorch for autopilot. *Corporates Today*.
- [Vega, 2018a] Vega, J. (2018a). *Educational framework using robots with vision for constructivist teaching Robotics to pre-university students*. Doctoral thesis on computer science and artificial intelligence, University of Alicante.
- [Vega, 2018b] Vega, J. (2018b). JdeRobot-Kids framework for teaching robotics and vision algorithms. In *II jornada de investigación doctoral*. University of Alicante.
- [Vega, 2019] Vega, J. (2019). El profesor Julio Vega, finalista del concurso 'Ciencia en Acción 2019'. URJC, on-line newspaper interview.
- [Vega and Cañas, 2019] Vega, J. and Cañas, J. (2019). PyBoKids: An innovative python-based educational framework using real and simulated Arduino robots. *Electronics*, 8:899–915.

[Vega et al., 2012] Vega, J., Perdices, E., and Cañas, J. (2012). *Attentive visual memory for robot localization*, pages 408–438. IGI Global, USA. Text not available. This book is protected by copyright.

[von Szadkowski and Reichel, 2020] von Szadkowski, K. and Reichel, S. (2020). Phobos: A tool for creating complex robot models. *Journal of Open Source Software*, 5(45):1326.