



## GRADO EN INGENIERÍA DE ROBÓTICA SOFTWARE

Escuela de Ingeniería de Fuenlabrada

Curso académico 2022-2023

### Trabajo Fin de Grado

Extensión de la herramienta VisualCircuit a ROS2  
para programar aplicaciones robóticas.

**Autor:** David Tapiador de Vera

**Tutor:** Jose María Cañas Plaza

# Agradecimientos

---

Después de tanto sufrir, por fin llega el momento de terminar. Ha sido un camino duro, incluyendo noches sin dormir y mucho estrés acumulado, pero por fin se acaba.

Tengo que agradecer a mis padres y hermana por ayudarme día a día a mejorar y superar los momentos malos.

A mis amigos por obligarme a salir incluso cuando menos ánimos tenía y ayudarme a desconectar de todo.

Y sobretodo tengo que agradecer a mi apoyo fundamental. Al pilar de mi vida y al que me hace seguir adelante día tras día, mi perrete Koby. Él sí que me obliga a salir aunque esté lloviendo, nevando, con las calles congeladas, con una pandemia... Simplemente gracias.

*Vida antes que muerte,  
fuerza antes que debilidad,  
viaje antes que destino.*

Brandon Sanderson

# 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

---

**ROS** *Robot Operating System*

**IMU** *Inertial Measurement Unit*

**LIDAR** *Laser Imaging Detection and Ranging*

**ODE** *Open Dynamics Engine*

**USB** *Universal Serial Bus*

**VFF** *Vector Force Field*

**URDF** *Unified Robotics Description Format*

**RVIZ** *ROS Visualization*

**RGBD** *Red Green Blue - Depth*

**FPS** *Frames Per Second*

**PID** *Proportional-Integral-Derivative controller*

# Índice general

---

<b>1. Introducción</b>	<b>1</b>
1.1. La primera sección . . . . .	1
1.2. Segunda sección . . . . .	1
1.2.1. Números . . . . .	2
1.2.2. Listas . . . . .	2
<b>2. Objetivos y metodología de Trabajo</b>	<b>4</b>
2.1. Descripción del problema . . . . .	4
2.2. Requisitos . . . . .	4
2.3. Metodología . . . . .	4
2.4. Plan de trabajo . . . . .	4
<b>3. Herramientas y plataforma de desarrollo</b>	<b>5</b>
3.1. Lenguaje de programación . . . . .	5
3.2. ROS2 (Robot Operating System 2) . . . . .	5
3.3. Gazebo . . . . .	6
3.4. Turtlebot2 . . . . .	7
3.4.1. Base Kobuki . . . . .	7
3.4.2. Cuerpo Turtlebot2 . . . . .	8
3.4.3. Turtlebot2 simulado . . . . .	9
3.5. RVIZ2 . . . . .	10
3.6. Sensores . . . . .	11
3.6.1. Cámara ASUS Xtion Pro . . . . .	11
3.6.2. RPLIDAR A2 . . . . .	11
3.7. VisualCircuit . . . . .	12
<b>4. Desarrollo de bloques driver</b>	<b>14</b>
4.1. Introducción a VisualCircuit . . . . .	14
4.2. Creación de bloques drivers . . . . .	16

<b>5. Sigue personas</b>	<b>24</b>
5.1. Desarrollo inicial sigue-personas . . . . .	24
<b>6. Máquina de estados (VFF)</b>	<b>30</b>
6.1. Conclusiones . . . . .	30
6.2. Corrector ortográfico . . . . .	31
<b>7. Conclusiones</b>	<b>32</b>
7.1. Conclusiones . . . . .	32
7.2. Corrector ortográfico . . . . .	33
<b>Bibliografía</b>	<b>34</b>

# Índice de figuras

---

1.1.	Robot aspirador Roomba de iRobot . . . . .	2
3.1.	Comunicación entre nodos ROS . . . . .	6
3.2.	Simulador Gazebo . . . . .	7
3.3.	Kobuki base . . . . .	8
3.4.	Turtlebot2 . . . . .	8
3.5.	Turtlebot2 simulado . . . . .	9
3.6.	RVIZ2 Vs mundo gazebo . . . . .	10
3.7.	Cámara ASUS-XTION . . . . .	11
3.8.	RPLIDAR A2 . . . . .	11
3.9.	VisualCircuit . . . . .	12
3.10.	Ejemplo VisualCircuit . . . . .	13
4.1.	Creando un bloque en VisualCircuit . . . . .	14
4.2.	Guardando un bloque en VisualCircuit . . . . .	15
4.3.	Ejemplo de proyecto en VisualCircuit . . . . .	15
4.4.	Modelo bloque driver sensores . . . . .	16
4.5.	Estructura mensaje Image . . . . .	19
4.6.	Estructura mensaje LaserScan . . . . .	20
4.7.	Estructura mensaje Twist . . . . .	22
5.1.	Modelo de persona en gazebo . . . . .	24
5.2.	Circuito sigue-personas inicial . . . . .	25
5.3.	Circuito del bloque PID sigue-persona . . . . .	28
5.4.	Circuito sigue-personas inicial . . . . .	29

# Listado de códigos

---

3.1. Hola mundo en python . . . . .	5
4.1. Modelo de código para bloques drivers . . . . .	17
4.2. Clase del nodo suscriptor para bloques drivers . . . . .	18
4.3. Función main para bloques drivers de sensores . . . . .	18
4.4. Clase del nodo suscriptor para cámara . . . . .	20
4.5. Cambios main bloque cámara . . . . .	20
4.6. Clase del nodo suscriptor para láser . . . . .	21
4.7. Cambios main bloque láser . . . . .	21
4.8. Bloque MotorDriverROS2 . . . . .	23
5.1. Modificación al bloque detector de objetos . . . . .	26
5.2. Código bloque decisión sigue-persona . . . . .	27
5.3. Código bloque rotación sigue-persona . . . . .	28

# Listado de ecuaciones

---

# Índice de cuadros

---

---

# Capítulo 1

## Introducción

---

Escribe aquí un párrafo explicando brevemente lo que vas a contar en este capítulo. En este primer capítulo, el de introducción, se trata de dar un contexto amplio y atractivo del trabajo. Comienza hablando de un contexto general y acaba hablando del contexto más específico en el que se enmarca el proyecto. Es el capítulo idóneo para incluir todas las referencias bibliográficas que hayan tratado este tema; suponen un fuerte respaldo al trabajo.

### 1.1. La primera sección

En los textos puedes poner palabras en *cursiva*, para aquellas expresiones en sentido *figurado*, palabras como *robota*, que está fuera del diccionario castellano, o bien para resaltar palabras de una colección: (a) es la primera letra del abecedario, (b) es la segunda, etc.

Al poner las dos líneas del anterior párrafo, este aparecerá separado del anterior. Si no las pongo, los párrafos aparecerán pegados. Sigue el criterio que consideres más oportuno.

### 1.2. Segunda sección

No olvides incluir imágenes y referenciarlas, como la Figura UwU 1.1.

Ni tampoco olvides de poner las URLs como notas al pie. Por ejemplo, si hablo de la Robocup<sup>1</sup>.

---

<sup>1</sup><http://www.robocup.org>



Figura 1.1: Robot aspirador Roomba de iRobot.

### 1.2.1. Números

En lugar de tener secciones interminables, como la Sección 1.1, divídelas en subsecciones.

Para hablar de números, mételos en el entorno *math* de L<sup>A</sup>T<sub>E</sub>X, por ejemplo,  $1,5Kg$ . También puedes usar el símbolo del Euro como aquí:  $1.500\text{€}$ .

### 1.2.2. Listas

Cuando describas una colección, usa `itemize` para ítems o `enumerate` para enumerados. Por ejemplo:

- *Entorno de simulación.* Hemos usado dos entornos de simulación: uno en 3D y otro en 2D.
  - *Entornos reales.* Dentro del campus, hemos realizado experimentos en Biblioteca y en el edificio de Gestión.
1. Primer elemento de la colección.
  2. Segundo elemento de la colección.

**Referencias bibliográficas** Cita, sobre todo en este capítulo, referencias bibliográficas que respalden tu argumento. Para citarlas basta con poner la instrucción `\cite` con el identificador de la cita. Por ejemplo: libros como [12], artículos como [11], URLs como [10], tesis como [8], congresos como [9], u otros trabajos fin de grado como [6].

Las referencias, con todo su contenido, están recogidas en el fichero `bibliografia.bib`. El contenido de estas referencias está en formato BibTeX. Este formato se puede obtener en muchas ocasiones directamente, desde plataformas como [Google Scholar](#) u otros repositorios de recursos científicos.

Existen numerosos estilos para reflejar una referencia bibliográfica. El estilo establecido por defecto en este documento es APA, que es uno de los estilos más comunes, pero lo puedes modificar en el archivo `memoria.tex`; concretamente, cambiando el campo `apalike` a otro en la instrucción `\bibliographystyle{apalike}`.

Y, para terminar este capítulo, resume brevemente qué vas a contar en los siguientes.

---

# **Capítulo 2**

# **Objetivos y metodología de Trabajo**

---

*Quizás algún fragmento de libro inspirador...*

Autor, Título

Escribe aquí un párrafo explicando brevemente lo que vas a contar en este capítulo. En este capítulo lo ideal es explicar cuáles han sido los objetivos que te has fijado conseguir con tu trabajo, qué requisitos ha de respetar el resultado final, y cómo lo has llevado a cabo; esto es, cuál ha sido tu plan de trabajo.

## **2.1. Descripción del problema**

Cuenta aquí el objetivo u objetivos generales y, a continuación, concrétalos mediante objetivos específicos.

## **2.2. Requisitos**

Describe los requisitos que ha de cumplir tu trabajo.

## **2.3. Metodología**

Qué paradigma de desarrollo software has seguido para alcanzar tus objetivos.

## **2.4. Plan de trabajo**

Qué agenda has seguido. Si has ido manteniendo reuniones semanales, cumplimentando objetivos parciales, si has ido afinando poco a poco un producto final completo, etc.

---

## Capítulo 3

# Herramientas y plataforma de desarrollo

---

El desarrollo de nuevo contenido para la plataforma de VisualCircuit, ha necesitado usar distintas herramientas, como por ejemplo programación, ROS2, gazebo..., por lo que voy a hacer una pequeña descripción de cada una, así como el uso que se le ha dado dentro del proyecto.

### 3.1. Lenguaje de programación

Python es un lenguaje interpretado de alto nivel. Este lenguaje busca facilitar la legibilidad del código, convirtiéndolo en uno de los más comunes a día de hoy. Es un lenguaje de programación multiparadigma, ya que soporta tanto programación orientada a objetos, como programación imperativa y funcional.

---

```
print("Hello World")
```

---

Código 3.1: Hola mundo en python

Dentro del TFG se usará para la programación dentro de la plataforma VisualCircuit (3.7).

### 3.2. ROS2 (Robot Operating System 2)

ROS<sup>1</sup> o Robot Operating System es un *middleware*<sup>2</sup> formado por un conjunto de herramientas y librerías de software libre empleadas para el desarrollo de aplicaciones robóticas. Su objetivo es ofrecer una plataforma estándar para todas las ramas de la robótica.

---

<sup>1</sup>ROS: <http://wiki.ros.org/es>

<sup>2</sup>Middleware: software que se sitúa entre las aplicaciones y el sistema operativo

ROS se basa en una arquitectura *cliente-servidor* centralizado que, mediante suscriptores y publicadores, permite enviar información, ya sean medidas de sensores, cambios de estado, decisiones usando árboles de decisión, órdenes a los actuadores, etc.

Para comunicarse con los servidores (o como se llaman en ROS, *topics*) se usan nodos. Estos nodos pueden contar con varios publicadores y suscriptores simultáneos. Cada *topic* se define con un tipo de mensaje, que será el único que se pueda enviar y recibir a través de él. Estos tipos de mensajes pueden ser mensajes simples como una cadena de caracteres o tipos compuestos con otros tipos, permitiéndonos crear topics adecuados a las necesidades de cada proyecto.

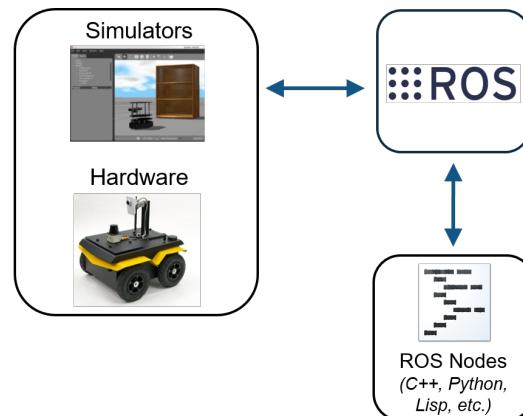


Figura 3.1: Comunicación del nodo Master con los nodos Intermedios y con distintos sensores y actuadores. Imagen obtenida de [2]

ROS2 lo usaremos para obtener información del robot turtlebot2 (3.4), tanto real como simulado, como por ejemplo su posición en el entorno simulado o las últimas medidas de sus sensores, y para comandarle instrucciones (velocidades a sus motores).

### 3.3. Gazebo

Gazebo<sup>3</sup> es un simulador 3D de código abierto orientado a la robótica que permite fusionar escenarios realistas con robots simulados, ofreciendo un entorno seguro para probar algoritmos. Éste utiliza el motor de físicas ODE<sup>4</sup>, aunque se puede configurar con otros motores, como Bullet<sup>5</sup> o DART<sup>6</sup>.

<sup>3</sup>Gazebo: <https://classic.gazebosim.org/>

<sup>4</sup>Open Dynamics Engine: <https://www.ode.org/>

<sup>5</sup>Bullet: <https://pybullet.org/wordpress/>

<sup>6</sup>DART: <https://dartsim.github.io/>

Al estar orientado a la robótica, permite integrar fácilmente modelos de robots reales con sensores (incluso simulando sus ruidos) y enviar a través de los distintos topics de ROS o ROS2 (3.2) alguna información directa del simulador, como la posición, medidas de los sensores simulados o incluso información de objetos no programables (del entorno).

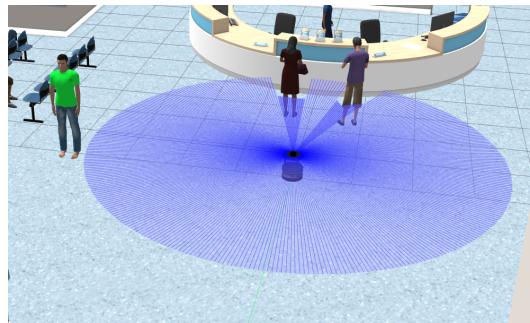


Figura 3.2: Ejemplo de ejecución en gazebo.

A lo largo de todo el proyecto, usaremos gazebo para simular el turtlebot2 (3.4) al igual que los distintos entornos que iremos usando para probar los programas.

## 3.4. Turtlebot2

La URJC de Fuenlabrada, en sus laboratorios de robótica, cuenta con varios robots Turtlebot2<sup>7</sup> a disposición de los alumnos del grado. Estos son perfectos para la enseñanza e investigación en robótica, por su sencilla introducción a temas como ROS o el uso de sensores. Los Turtlebot2 están formados por dos partes principales: una base Kobuki y una estructura superior.

### 3.4.1. Base Kobuki

La base del Turtlebot2 se llama *Kobuki*. En apariencia, es similar a un robot de limpieza como podrían ser los Roomba. En cuanto al hardware, lleva integrados tres bumpers (sensores de contacto), odometría, sensor de caída y varios giroscopios. Tiene una velocidad lineal máxima de 0.7 m/s y angular de 180 grados/s. Su batería le permite una autonomía de entre 3 y 7 horas. Cuenta con varios puertos, entre ellos un USB para poder conectar nuestro portatil y ejecutar los distintos algoritmos.

---

<sup>7</sup>Turtlebot2: <https://www.turtlebot.com/turtlebot2/>

Algunos de los paquetes de ROS2 que instalaremos para poder usarlo son los drivers del kobuki para ROS2-Humble<sup>8</sup> de IntelligentRoboticsLabs, compañeros de la URJC. Siguiendo las instrucciones de instalación que se encuentran en dicho repositorio de github, accedemos a varios paquetes básicos para el uso de kobuki, como *kobuki\_ros* o *kobuki\_node*, entre otros.



Figura 3.3: Base kobuki. Imagen obtenida de [5]

### 3.4.2. Cuerpo Turtlebot2

El cuerpo del turtlebot2 (también conocido como *TurtleBot Structure*) está formado por una serie de plataformas y tubos que se atornillan a la base kobuki y permiten fijar nuevos sensores, como podrían ser una cámara o un láser, o actuadores como brazos robóticos. También ofrece un sitio cómodo para poder colocar el portátil encima del robot y así poder conectarlo a la base mediante USB.



Figura 3.4: Turtlebot2. Imagen obtenida de [4]

---

<sup>8</sup>Drivers kobuki ROS2-Humble: <https://github.com/IntelligentRoboticsLabs/Robots/tree/humble/kobuki>

### 3.4.3. Turtlebot2 simulado

Para algunas partes del proyecto, como el desarrollo de drivers para ROS2 (4) o el VFF usando máquinas de estados (6), hemos usado el simulador para probar y desarrollar los algoritmos. Para esto, he tenido que usar un modelo del turtlebot2 que cuenta con los mismos sensores (cámara, RPLIDAR, bumper, etc) que el real, así como los mismos topics.

Para integrar el modelo del robot en el simulador, necesitamos su representación en URDF<sup>9</sup>, una forma estandarizada de crear los modelos de los robots incluyendo sus sensores y actuadores, partes móviles etc.

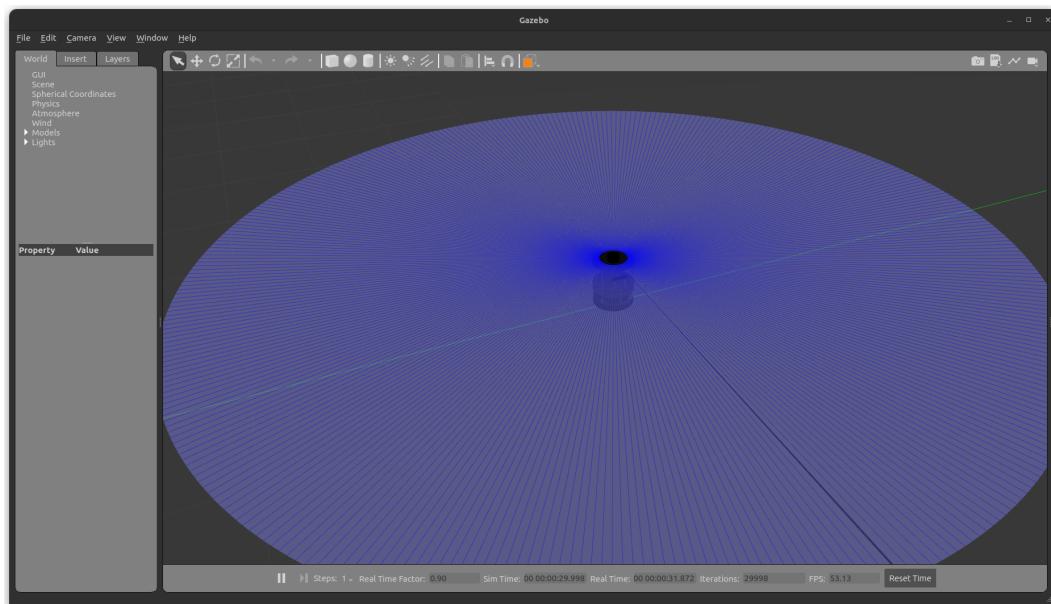


Figura 3.5: Turtlebot2 en gazebo.

<sup>9</sup>URDF: Unified Robot Description Format

### 3.5. RVIZ2

RVIZ2 es una herramienta de visualización 3D para robots, el ambiente y las medidas de los sensores de éstos.

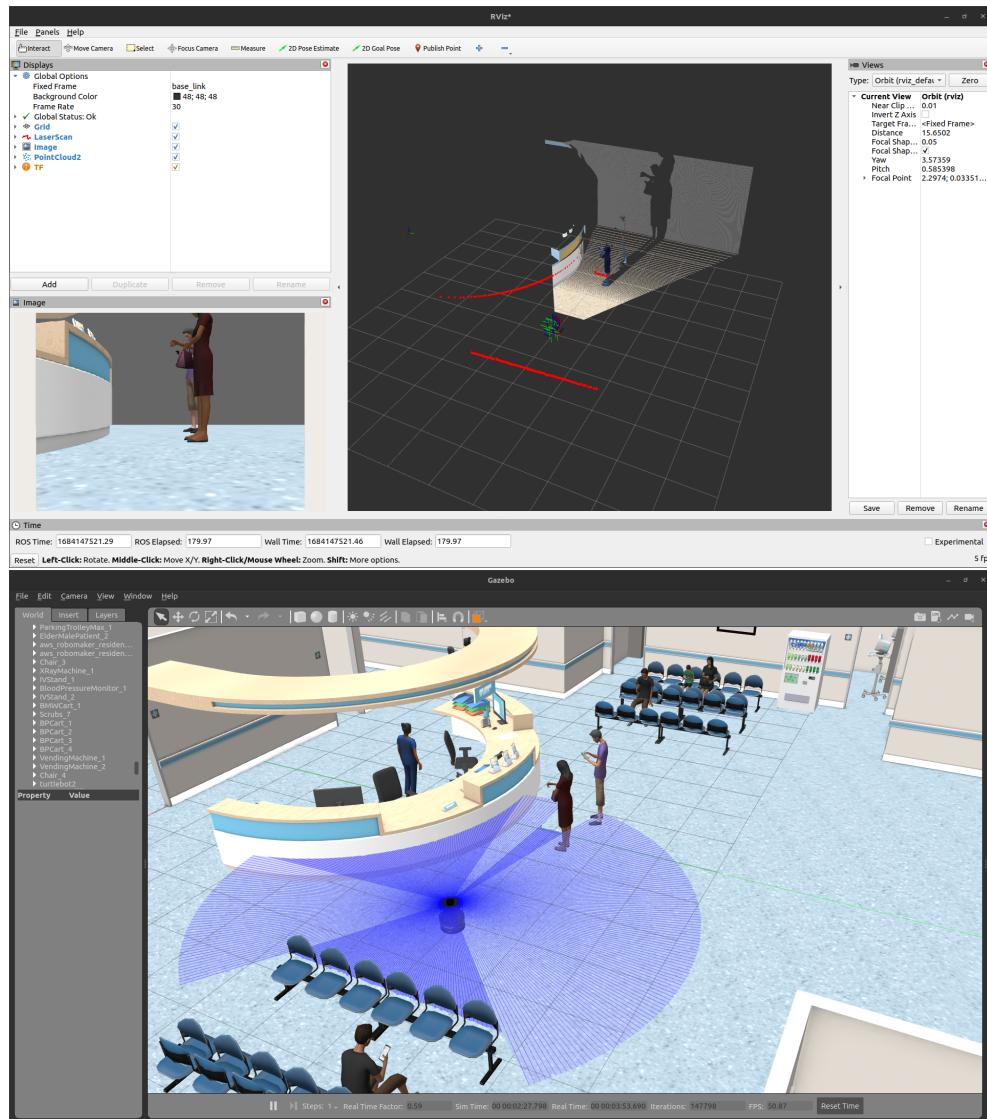


Figura 3.6: Ejemplo de RVIZ2 frente al mundo gazebo real.

RVIZ2 se usará bastante durante el proyecto tanto para depurar como para observar las medidas de los distintos sensores a tiempo real.

## 3.6. Sensores

Como hemos mencionado anteriormente, un robot se compone, a grandes rasgos, de sensores y actuadores. Para este proyecto se han usado varios de ellos, por lo que aquí hay una pequeña introducción a cada uno.

### 3.6.1. Cámara ASUS Xtion Pro

La cámara *ASUS Xtion* es una cámara RGB-D<sup>10</sup>, que ofrece tanto imagen como una nube de puntos con la distancia medida para cada pixel de la imagen. Esta cámara ofrece una imagen de 720p, con una frecuencia de 60fps. En la parte de profundidad, es capaz de captar desde 0.8m hasta 3.5 con un ángulo efectivo de 70°. Se conecta mediante USB directamente al ordenador.

En el proyecto, como debemos usarla con ROS2, usaremos el paquete creado por un usuario de internet<sup>11</sup>.



Figura 3.7: Cámara ASUS-XTION. Imagen obtenida de [1]

### 3.6.2. RPLIDAR A2

Se trata de un láser de 360° con un rango de medida desde 0.2m hasta 16m y una frecuencia de muestreo que se puede ajustar desde 5Hz hasta 15Hz. Usando los drivers mencionados en el apartado del Turtlebot2 (3.4.1) encontraremos un paquete para poder activar y usar este sensor con ROS2.



Figura 3.8: Sensor RPLIDAR A2. Imagen obtenida de [3]

---

<sup>10</sup>**RGB-D**: RedGreenBlue-Depth, hace referencia las cámaras que captan la imagen y las distancias de cada pixel.

<sup>11</sup>**Drivers ASUS-Xtion ROS2**: [https://github.com/mgonzs13/ros2\\_asus\\_xtion](https://github.com/mgonzs13/ros2_asus_xtion)

### 3.7. VisualCircuit

VisualCircuit<sup>12</sup> es un editor visual online basado en programación por bloques de código orientado al desarrollo de aplicaciones robóticas. Está desarrollado sobre IceStudio<sup>13</sup>.

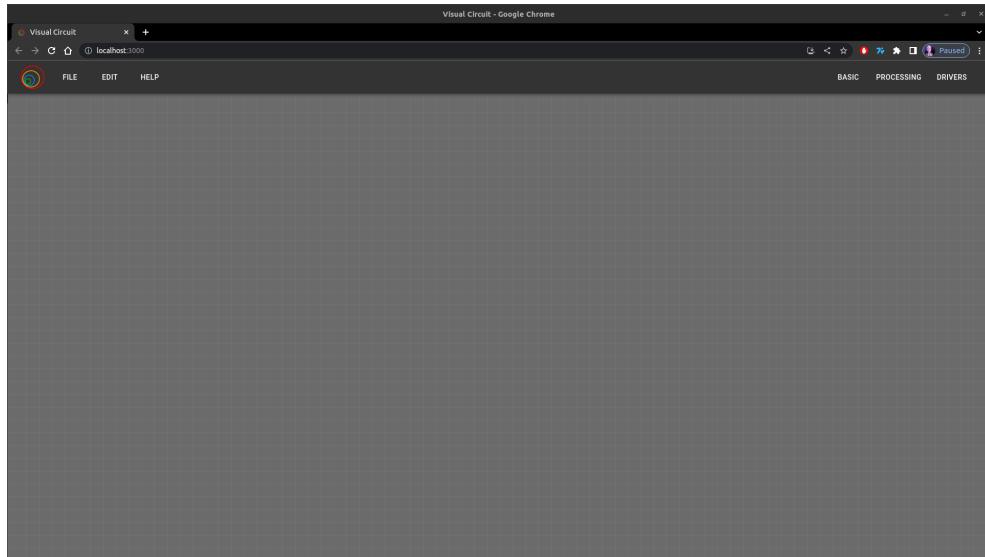


Figura 3.9: Página de VisualCircuit.

Los bloques que se pueden usar están divididos en varias pestañas: *basics*, *processing* y *drivers*.

- *Basic*: bloques simples, como inputs y outputs, bloques para definir parámetros y constantes, o bloques para insertar nuestro código.
- *Processing*:
  - *Control*: bloques de control (PID).
  - *OpenCV*: bloques relacionados con OpenCV<sup>14</sup> y edición de imagen (filtros de color, detección de contornos, erosión, etc).
  - *TensorFlow*: un bloque para detección de objetos.

<sup>12</sup>VisualCircuit Docs: <https://jderobot.github.io/VisualCircuit/>

<sup>13</sup>IceStudio Project: <https://github.com/FPGAwars/icestudio>

<sup>14</sup>OpenCV: <https://opencv.org/>

- *Drivers*: drivers que conectan con los sensores y actuadores

- *Control*: motordriver (ROS) y teleoperador.
- *OpenCV*: lector de imágenes desde archivos y desde cámaras, pantalla para mostrar las imágenes.
- *ROS-Sensors*: cámara, odometría e IMU<sup>15</sup> usando ROS.

Ésta será la herramienta principal en torno a la que girará este proyecto, tanto buscando añadir funcionalidades como en desarrollar aplicaciones con ella.

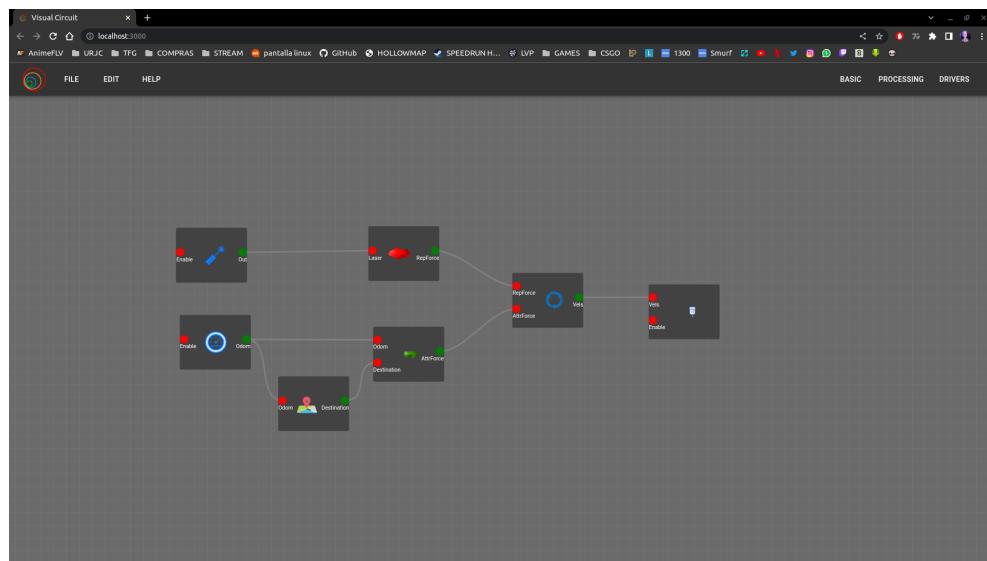


Figura 3.10: Ejemplo de proyecto de VisualCircuit.

<sup>15</sup>IMU: Inertial Measurement Unit

---

## Capítulo 4

# Desarrollo de bloques driver

---

Como ya he explicado, VisualCircuit es una plataforma de programación online mediante el uso de bloques, pero para que esté actualizado, hay que ir añadiendo bloques nuevos que ofrezcan esas nuevas funciones y añadirlos a las listas de bloques estándar que ofrece la página.

En este capítulo profundizaremos en el funcionamiento de la plataforma VisualCircuit así como en el proceso seguido para desarrollar nuevos bloques para poder añadir ROS2 a la misma.

### 4.1. Introducción a VisualCircuit

En VisualCircuit, antes de realizar este proyecto, ya existían bloques dedicados específicamente a la robótica para algunos sensores (cámara, odometría e IMU<sup>1</sup>) y para los motores usando ROS melodic<sup>2</sup>, pero al tratarse de una versión antigua, decidimos que ya era momento de actualizar a ROS2 humble<sup>3</sup>, ya que era la versión estable más moderna hasta el momento.

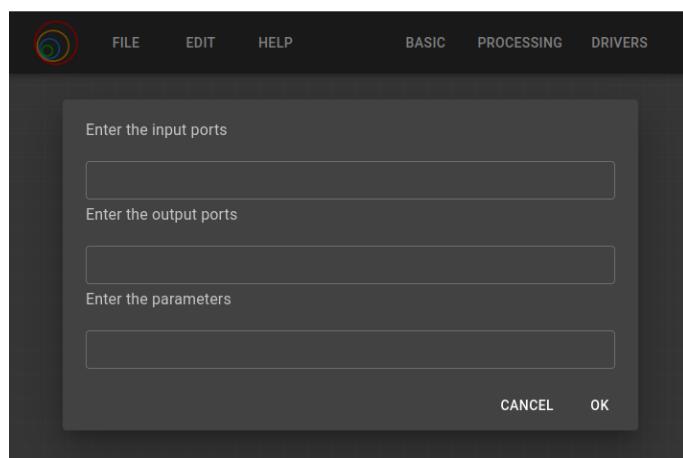


Figura 4.1: Creando un bloque en VisualCircuit.

<sup>1</sup>IMU: Inertial Measurement Unit

<sup>2</sup>ROS melodic: <http://wiki.ros.org/melodic>

<sup>3</sup>ROS humble: <https://docs.ros.org/en/humble/index.html>

Para crear nuestro propio bloque, debemos añadir varios bloques prefabricados. Para introducir nuestro código principal usaremos el bloque *code*, como se puede ver en la figura 4.1. Al crearlo, se nos permite definir el número de entradas, salidas y parámetros que tendrá nuestro código.

Una vez definidos, tenemos que usar bloques de *Input*, *Output* y *Constant* y así, usando “*Save as*”, podemos exportarlo para usarlo como un bloque nuevo.

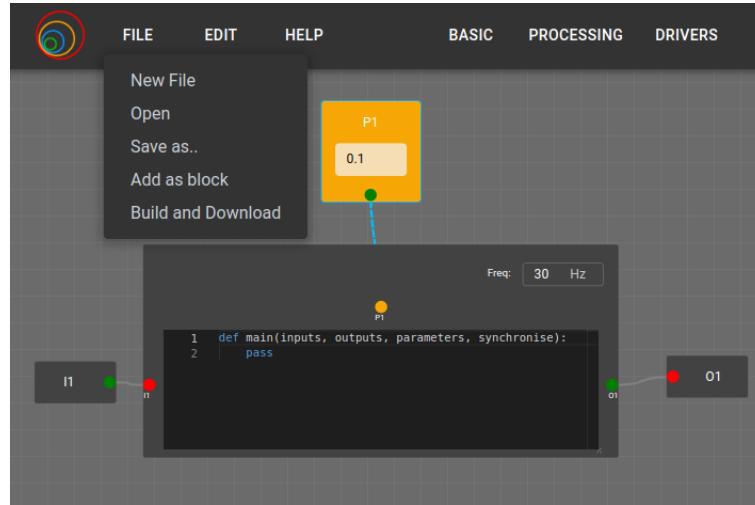


Figura 4.2: Guardando un bloque en VisualCircuit.

Cuando tengamos varios bloques generados, podemos usar la opción *FILE* → *Add as block* para añadir como nuevos bloques los que acabamos de crear y así formar un circuito completo con el comportamiento que queramos.

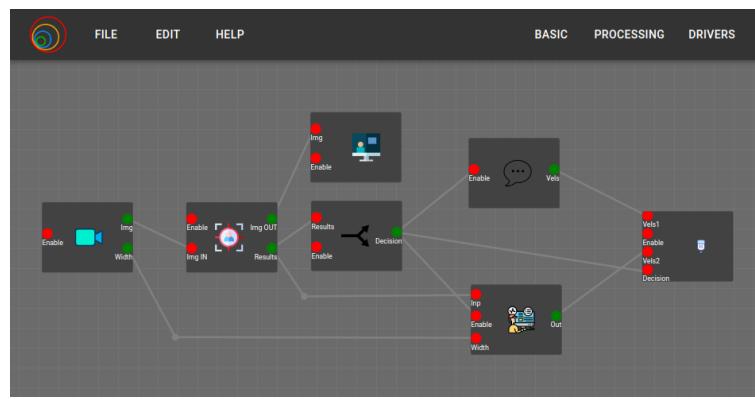


Figura 4.3: Ejemplo de proyecto en VisualCircuit.

Este es el funcionamiento general que debemos de seguir para usar VisualCircuit. Ahora, para crear bloques drivers nuevos para usar con ROS2 debemos centrarnos más en el código.

## 4.2. Creación de bloques drivers

Los bloques drivers que vamos a desarrollar son para implementar la cámara, el láser y los motores usando un topic de ROS2. Para los sensores (cámara y laser) tenemos que crear la estructura que queremos seguir, ya que ambos serán similares entre sí.

Usando un *input* para activar/desactivar el bloque habilitaremos el uso de máquinas de estados con nuestro bloque. También queremos compartir la medida del sensor, por lo que añadimos un *output*. Por último, usamos una constante donde definiremos el *topic* del que obtendremos las lecturas del sensor. Esto lo hacemos para que el usuario no tenga que cambiar el código del bloque para cambiar el *topic* del que obtiene la información.

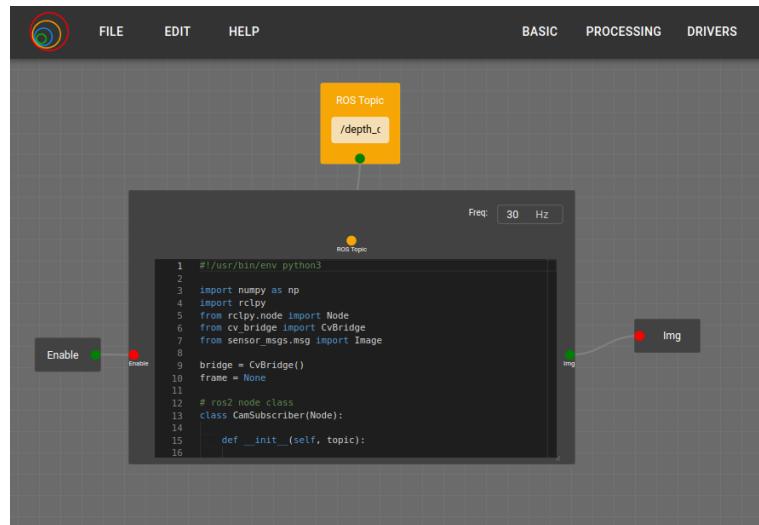


Figura 4.4: Modelo para crear bloques driver de sensores.

En cuanto al código que usaremos, seguiremos las indicaciones de los manuales de ROS2-humble<sup>4</sup> para usar nodos suscriptores/publicadores con python. El código general para los bloques de sensores (suscriptores) será el siguiente:

---

<sup>4</sup><https://docs.ros.org/en/humble/Tutorials/Beginner-Client-Libraries/Writing-A-Simple-Py-Publisher-And-Subscriber.html#write-the-subscriber-node>

---

```

import numpy as np
import rclpy
from rclpy.node import Node
from cv_bridge import CvBridge
from std_msgs/msg import String #SENSOR MSG TYPE

bridge = CvBridge()
data = None

# ros2 node class
class SENSORSubscriber(Node):
    def __init__(self, topic):
        super().__init__('sensor_subscriber')
        self.subscription = self.create_subscription(
            String, topic, self.callback, 10)

    self.subscription # prevent unused variable warning

    def callback(self, msg):
        global data
        # Modify msg as needed and save into global variable
        data = msg

    def main(inputs, outputs, parameters, synchronise):
        global data
        auto_enable = False
        try:
            enable = inputs.read_number('Enable')
        except Exception:
            auto_enable = True
        rclpy.init()
        sensor_sub = SENSORSubscriber(parameters.read_string("ROSTopic"))

        try:
            while auto_enable or inputs.read_number('Enable'):
                data = None
                rclpy.spin_once(sensor_sub)
                if data is not None:
                    outputs.share_string("Output", data)
                    synchronise()
        except Exception as e:
            print('Error:', e)
            pass
        finally:
            print("Exiting")
            synchronise()
            SENSORSubscriber.destroy_node()
            rclpy.shutdown()

```

---

Código 4.1: Modelo de código para bloques drivers.

Si lo analizamos por partes, la clase ***SENSORSubscriber*** (código 4.2) contiene una función para inicializar la clase, donde se define el nombre del nodo, se crea el suscriptor y se inicia el suscriptor, y una función callback a la que se llamará de forma periódica, actualizando el valor de la variable global a la última medida del sensor.

---

```
# ros2 node class
class SENSORSubscriber(Node):
    def __init__(self, topic):
        super().__init__('sensor_subscriber')
        self.subscription = self.create_subscription(
            String, topic, self.callback, 10)

    self.subscription # prevent unused variable warning

    def callback(self, msg):
        global data
        # Modify msg as needed and save into global variable
        data = msg
```

---

Código 4.2: Clase del nodo suscriptor para los bloques drivers.

En la función main (código 4.3) tenemos dos partes, la secuencia *try-except* donde se declara si se está usando una maquina de estados (*enable*) o si debe estar activo constantemente (*autoenable* al haber dado error la lectura del cable de *enable*). En la segunda parte (bucle *while*) se reinicia el valor de la variable global y se llama a “*rclpy.spin\_once()*” para obtener la última medida del sensor y compartirlo por el cable “*output*”.

---

```
def main(inputs, outputs, parameters, synchronise):
    global data
    auto_enable = False
    try:
        enable = inputs.read_number('Enable')
    except Exception:
        auto_enable = True
    rclpy.init()
    sensor_sub = SENSORSubscriber(parameters.read_string("ROSTopic"))

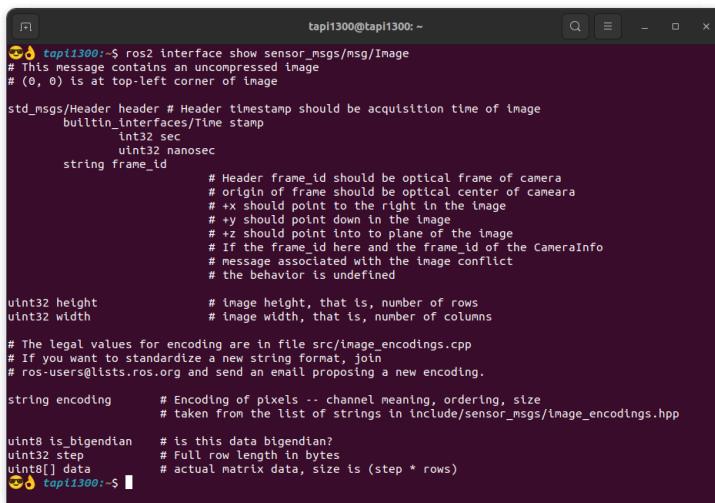
    try:
        while auto_enable or inputs.read_number('Enable'):
            data = None
            rclpy.spin_once(sensor_sub)
            if data is not None:
                outputs.share_string("Output", data)
```

---

Código 4.3: Función main para los bloques drivers de sensores.

Una vez que tenemos el modelo general para estos bloques, hay que modificarlos para que funcionen con la cámara y con el láser. Para ello, cambiaremos el tipo de mensaje que se envía, y haremos que el *callback* obtenga del mensaje de ROS sólo la información que nos interesa, ya que éste viene con cabeceras y otros datos.

Para el bloque correspondiente a la cámara, el mensaje es del tipo *sensor\_msgs.msg.Image* que, usando el comando “`ros2 interface show sensor_msgs/msg/Image`”, podemos ver cuál es su estructura:



```
tapi1300:~$ ros2 interface show sensor_msgs/msg/Image
# This message contains an uncompressed image
# (0, 0) is at top-left corner of image
std_msgs/Header header # Header timestamp should be acquisition time of image
    builtin_interfaces/time_stamp
        int32 sec
        uint32 nanosec
    string frame_id
        # Header frame_id should be optical frame of camera
        # origin of frame should be optical center of camera
        # +x should point to the right in the image
        # +y should point down in the image
        # +z should point into to plane of the image
        # If the frame_id here and the frame_id of the CameraInfo
        # message associated with the image conflict
        # the behavior is undefined
    uint32 height
        # image height, that is, number of rows
    uint32 width
        # image width, that is, number of columns
    # The legal values for encoding are in file src/image_encodings.cpp
    # If you want to standardize a new string format, join
    # ros-users@lists.ros.org and send an email proposing a new encoding.
    string encoding
        # Encoding of pixels -- channel meaning, ordering, size
        # taken from the list of strings in include/sensor_msgs/image_encodings.hpp
    uint8 is_bigendian
    uint32 step
    uint8[] data
        # actual matrix data, size is (step * rows)
tapi1300:~$
```

Figura 4.5: Estructura de tipo de mensaje *sensor\_msgs/msg/Image*.

Podemos ver en la figura 4.5 que la imagen que obtendremos estará en el campo *data* del mensaje, pero ROS tiene una función llamada *CvBridge*<sup>5</sup>, que nos permite transformar un mensaje del tipo *sensor\_msgs/msg/Image* en una imagen de numpy<sup>6</sup>. Como los hilos de VisualCircuit comparten las imágenes como un array de numpy, debemos convertir la imagen de numpy a un array de numpy, usando la función *numpy.asarray*.

<sup>5</sup>CvBridge: [http://wiki.ros.org/cv\\_bridge](http://wiki.ros.org/cv_bridge)

<sup>6</sup>Numpy: <https://numpy.org/>

---

```

class CamSubscriber(Node):
    def __init__(self, topic):
        super().__init__('cam_subscriber')
        self.subscription = self.create_subscription(
            Image, topic, self.callback, 10)
        self.subscription # prevent unused variable warning
    def callback(self, msg):
        global frame
        frame = np.asarray(bridge.imgmsg_to_cv2(msg, "bgr8"),
                           dtype=np.uint8)

```

---

Código 4.4: Clase del nodo suscriptor para la cámara.

En cuanto a la función main, sólo habría que cambiar la función que usamos para compartir, ya que ahora compartimos una imagen:

---

```

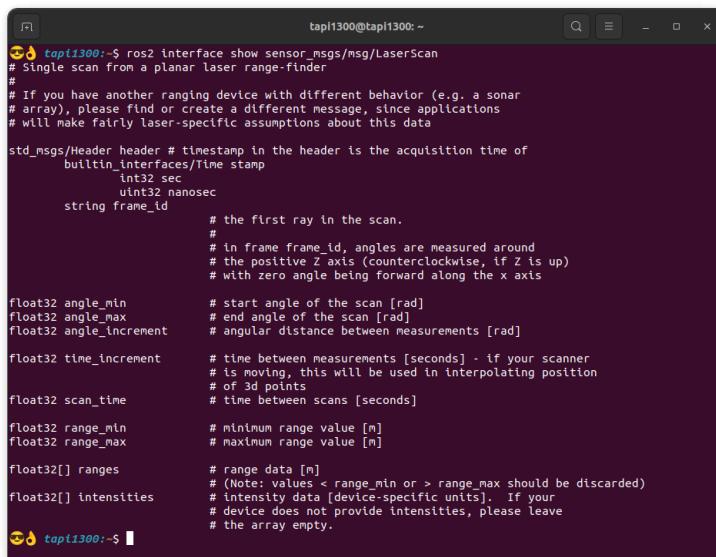
while auto_enable or inputs.read_number('Enable'):
    frame = None
    rclpy.spin_once(camera_subscriber)
    if frame is not None:
        outputs.share_image("Out", frame)

```

---

Código 4.5: Cambios a la función main del bloque driver de la cámara.

Para el láser, podemos hacer lo mismo que con la cámara y revisar la estructura del tipo de mensaje, en este caso usando “`ros2 interface show sensor_msgs/msg/LaserScan`”:



```

tapi1300@tapi1300: ~
ros2 interface show sensor_msgs/msg/LaserScan
# Single scan from a planar laser range-finder
#
# If you have another ranging device with different behavior (e.g. a sonar
# array), please find or create a different message, since applications
# will make fairly laser-specific assumptions about this data
std_msgs/Header header # timestamp in the header is the acquisition time of
                       # builtin_interfaces/Time stamp
                       # int32 sec
                       # uint32 nanosec
string frame_id          # the first ray in the scan.
                         #
                         # in frame frame_id, angles are measured around
                         # the positive Z axis (counterclockwise, if Z is up)
                         # with zero angle being forward along the X axis
float32 angle_min         # start angle of the scan [rad]
float32 angle_max         # end angle of the scan [rad]
float32 angle_increment   # angular distance between measurements [rad]
float32 time_increment     # time between measurements [seconds] - if your scanner
                           # is moving, this will be used in interpolating position
                           # of 3D points
float32 scan_time          # time between scans [seconds]
float32 range_min           # minimum range value [m]
float32 range_max           # maximum range value [m]
float32[] ranges             # range data [m]
                             # (Note: values < range_min or > range_max should be discarded)
float32[] intensities       # intensity data [device-specific units]. If your
                           # device does not provide intensities, please leave
                           # the array empty.

```

Figura 4.6: Estructura de tipo de mensaje `sensor_msgs/msg/LaserScan`.

Podemos ver en la figura 4.6 que la lectura del láser estará en el campo *ranges* del mensaje, que es un *array* de *floats* por lo que, para guardarlo en un array local usaremos la función de python “*.extend()*”, que nos permite añadir una entrada más a un array. Esto lo haremos en el callback, ya que es donde actualizamos el valor de la variable global.

---

```
class LaserSubscriber(Node):
    def __init__(self, topic):
        super().__init__('laser_subscriber')
        self.subscription = self.create_subscription(
            LaserScan, topic, self.callback, 10)
        self.subscription # prevent unused variable warning
    def callback(self, msg):
        global measure
        measure = []
        for i in range(len(msg.ranges)):
            measure.append(str(msg.ranges[i]))
```

---

Código 4.6: Clase del nodo suscriptor para el láser.

En cuanto a la función main, sólo habría que cambiar la función que usamos para compartir, ya que ahora compartimos un array:

---

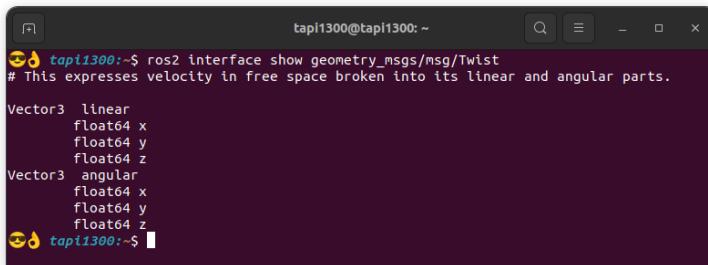
```
while auto_enable or inputs.read_number('Enable'):
    measure = None
    rclpy.spin_once(laser_subscriber)
    if measure is not None:
        outputs.share_array("Out", measure)
    synchronise()
```

---

Código 4.7: Cambios a la función main del bloque driver del láser.

Ya tenemos los bloques de los sensores, pero aún nos queda el que corresponde a los motores. En este caso, la configuración del bloque será distinta, ya que no necesitamos salida y sí necesitamos una entrada para que nos digan qué velocidades mandar al robot. Por esto, tendremos un bloque de código, un parámetro para el *topic* de ROS2 y dos entradas, una para *enable* (máquinas de estados) y otra para las velocidades que debemos enviar.

El tipo de mensaje que suelen admitir los robots es “geometry\_msgs/msg/Twist”, y si miramos su estructura usando “`ros2 interface show geometry_msgs/msg/Twist`”, nos encontramos lo siguiente:



```
tapi1300:~$ ros2 interface show geometry_msgs/msg/Twist
# This expresses velocity in free space broken into its linear and angular parts.

Vector3 linear
  float64 x
  float64 y
  float64 z
Vector3 angular
  float64 x
  float64 y
  float64 z
tapi1300:~$
```

Figura 4.7: Estructura de tipo de mensaje *geometry\_msgs/msg/Twist*.

Como podemos ver en la figura 4.7, este tipo de mensajes lleva dos paquetes de 3 floats, el primero para las velocidades lineales y el segundo para las angulares. Para crear un bloque más general que pueda adaptarse a todo tipo de robots, el bloque recibirá un array de 6 floats permitiendo introducir velocidades lineales y angulares en las 3 dimensiones que nos permite el mensaje, ya que aunque la mayoría de robots terrestres sólo usen una velocidad lineal y una angular, así permitimos el uso de este bloque con otros robots como drones o robots con ruedas omnidireccionales.

Para poder crear el nodo publicador iremos, al igual que hicimos con el suscriptor, a los manuales de ROS2-humble<sup>7</sup> y a partir de ahí crear nuestro nodo publicador.

---

```
import numpy as np
import rclpy
from rclpy.node import Node
from cv_bridge import CvBridge
from geometry_msgs.msg import Twist

bridge = CvBridge()
velocities = 0
```

---

<sup>7</sup><https://docs.ros.org/en/humble/Tutorials/Beginner-Client-Libraries/Writing-A-Simple-Py-Publisher-And-Subscriber.html#write-the-publisher-node>

---

```

# ros2 node class
class VelPublisher(Node):
    def __init__(self, topic):
        super().__init__('vel_publisher')
        self.publisher_ = self.create_publisher(Twist, topic, 1)
        timer_period = 0.5 # seconds
        self.timer = self.create_timer(timer_period, self.timer_callback)
    def timer_callback(self):
        global velocities
        msg = Twist()
        try:
            msg.linear.x = float(velocities[0])
            msg.linear.y = float(velocities[1])
            msg.linear.z = float(velocities[2])
            msg.angular.x = float(velocities[3])
            msg.angular.y = float(velocities[4])
            msg.angular.z = float(velocities[5])
        except IndexError:
            print("bad length for input array")
        return
        self.publisher_.publish(msg)

def main(inputs, outputs, parameters, synchronise):
    global velocities
    auto_enable = False
    try:
        enable = inputs.read_number('Enable')
    except Exception:
        auto_enable = True
    rclpy.init()
    vel_publisher = VelPublisher(parameters.read_string('ROSTopic'))
    try:
        while auto_enable or inputs.read_number('Enable'):
            velocities = inputs.read_array('Vels')
            if velocities != None:
                rclpy.spin_once(vel_publisher)
            synchronise()
    except KeyboardInterrupt:
        vel_publisher.destroy_node()
    rclpy.shutdown()

```

---

Código 4.8: Bloque MotorDriverROS2 completo.

Al ser un publicador, funciona con un temporizador para publicar el mensaje actualizado. Como podemos ver, se lee un array, se guarda en la variable global y cuando vuelva a ejecutarse el temporizador, se enviará la última actualización de las velocidades.

---

# Capítulo 5

## Sigue personas

---

Ahora que ya tenemos integrado ROS2 dentro de la plataforma de VisualCircuit, vamos a crear varios proyectos usando los bloques drivers que hemos creado. El primero de ellos será un comportamiento de *follow-person* usando reconocimiento visual.

### 5.1. Desarrollo inicial sigue-personas

En primer lugar, debemos preparar el entorno de pruebas, por lo que aprovecharemos un modelo de persona teleoperada que creó Carlos Caminero<sup>1</sup>, compañero de la carrera.



Figura 5.1: Modelo de persona teleoperable en gazebo.

El mundo que tenía Carlos creado incluía muchos elementos del entorno que no son necesarios en nuestro caso, por lo que modificaremos el mundo para dejar únicamente al robot y a la persona. El modelo del robot que usaremos será el que mencionamos en el capítulo 3.4.3, ya que incluye tanto cámara como láser.

---

<sup>1</sup>[https://github.com/RoboticsLabURJC/2021-tfg-carlos-caminero/tree/main/amazon\\_hospital/hospital\\_world](https://github.com/RoboticsLabURJC/2021-tfg-carlos-caminero/tree/main/amazon_hospital/hospital_world)

El comportamiento sigue-persona que buscamos desarrollar con VisualCircuit consiste en rotar en círculos hasta encontrar a una persona mediante algoritmos de detección visual de objetos y mantener a la persona centrada en la imagen, al igual que mantenernos a una distancia constante, usando así la función de lectura de distancias de la cámara. Por ello sólamente la cámara como sensor, ya que esta función mencionada nos permite no usar el láser y evitar un código más complejo.

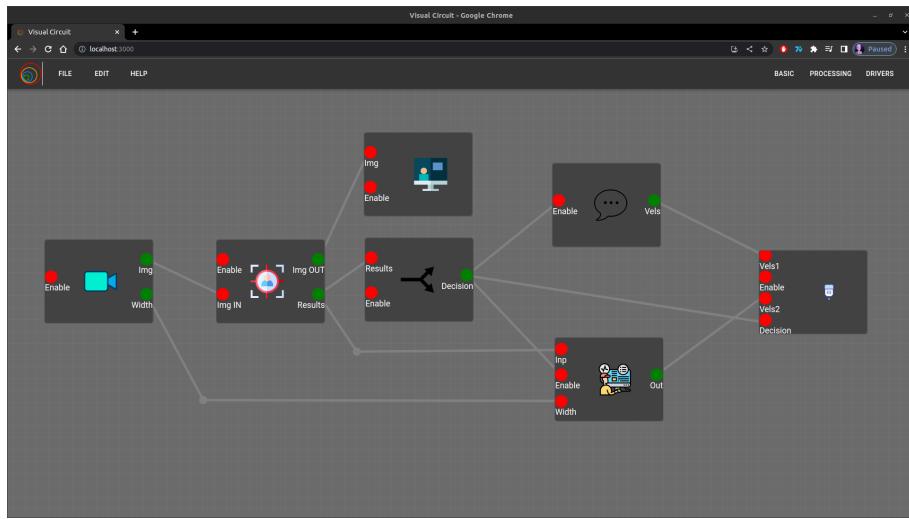


Figura 5.2: Circuito inicial del algoritmo sigue-persona.

La lógica que seguirá será la siguiente: recibir la imagen del *topic* de la cámara y compartirla con el bloque de detección de objetos, enviar la imagen con las detecciones al bloque *screen* para visualizar en tiempo real lo que está analizando el robot, y también mandaremos los resultados a un bloque que decidirá qué hacer. Este bloque activa un PID en caso de que haya una persona en la imagen, o el comportamiento de rotación en caso de que no se haya encontrado ninguna. En ambos casos, se envía la decisión a los bloques que generan las velocidades y también al bloque *MotorDriver* que hemos creado en el punto 4.2, que recibe tanto las distintas velocidades, como la decisión que se ha tomado, y envía al *topic* la adecuada.

La detección visual utiliza yolov3<sup>2</sup>, un algoritmo de detección de objetos a tiempo real que permite identificarlos tanto en video como en imágenes usando redes neuronales (darknet<sup>3</sup>). El bloque que ya está integrado en VisualCircuit nos sirve para nuestra aplicación, pero he tenido que modificarlo para poder extraer también la localización de la *Bounding Box* que corresponde a la persona y compartirla con otros bloques.

<sup>2</sup>YouOnlyLookOnce (YOLO): <https://pjreddie.com/darknet/yolo/>

<sup>3</sup>DarkNet: <https://pjreddie.com/darknet/>

Para ello, después de obtener los nombres de los objetos encontrados, recorremos toda la lista comprobando si hay alguna persona, en caso de haberla enviamos la *Bounding Box* correspondiente, sino, enviamos un *array* con cuatro valores “-1” para indicar que está vacío.

---

```

*****  

#forward Pass  

results = net.forward(outputNames)  

findObjects(results,frame)  

is_person = False  

for i in classIds:  

    if(className[i] == "person"):  

        is_person = True  

        break  

to_send = [-1,-1,-1,-1]  

if(is_person):  

    to_send = bbox[i]  

outputs.share_image("Img OUT", frame)  

outputs.share_array("Results", to_send)  

synchronise()  

*****

```

---

Código 5.1: Modificación al bloque de la detección de objetos.

El siguiente bloque (5.2) es el que toma las decisiones de qué comportamiento seguir. Para ello, primero esperaremos hasta recibir algún resultado de la visión y así no movernos antes haber podido analizar la situación.

Una vez que tengamos resultados, miraremos si es una *Bounding Box* válida, en caso de serlo, la decisión será seguir lo que indique el bloque PID. En caso de ser una caja vacía (*array* de “-1”) activaremos un contador para aplicar un filtro de paso bajo. Este filtro nos permite evitar cambiar de comportamiento por pequeños errores en la detección de objetos. Está establecido a 10, por lo que al llegar a 10 imágenes seguidas sin una persona en la imagen, cambiaremos de comportamiento al de la rotación.

---

```

def main(inputs, outputs, parameters, synchronise):
    auto_enable = True
    try:
        enable = inputs.read_number("Enable")
    except Exception:
        auto_enable = True

    while(True):
        # Wait for results
        results = inputs.read_array("Results")
        try:
            if(results.any()):
                break
        except Exception:
            continue

        not_to_enable = 0
        to_enable = 1
        lowpass_filter = 10
        counter = 0
        print("EMPEZAMOS")

        while(auto_enable or inputs.read_number('Enable')):
            results = inputs.read_array("Results")
            if(results[0] != -1):
                # Follow
                counter = 0
                outputs.share_number("Decision", 1)
            elif(counter < lowpass_filter):
                # Follow but low-pass filter
                counter += 1
                outputs.share_number("Decision", 2)
            else:
                # Rotation
                outputs.share_number("Decision", 0)

```

---

Código 5.2: Código del bloque de decisiones del sigue-persona.

En cuanto al bloque que envía la velocidad correspondiente al comportamiento de rotación, tiene un bucle que lee el cable que le llega, en caso de ser un "1" (*True*) informa en la terminal que estamos rotando y envía una velocidad angular de 1rad/s para el eje Z.

```
import numpy as np

def main(inputs, outputs, parameters, synchronise):
    try:
        while 1:
            if(inputs.read_number('Enable')):
                print("ROT")
                vels = [0,0,0,0,0,1]
                to_write = np.array(vels, dtype='<U64')
                outputs.share_array("Vels", to_write)
                synchronise()
    except Exception as e:
        print("Error")
```

Código 5.3: Código del bloque de la rotación del sigue-persona.

En paralelo al anterior, también se puede activar el bloque PID<sup>4</sup>. Su estructura consiste en tres entradas (Resultados de la detección de objetos, ancho de la imagen y *enable*), tres parámetros para las tres constantes del controlador y una salida para la velocidad lineal y angular final que aplicaremos al robot.

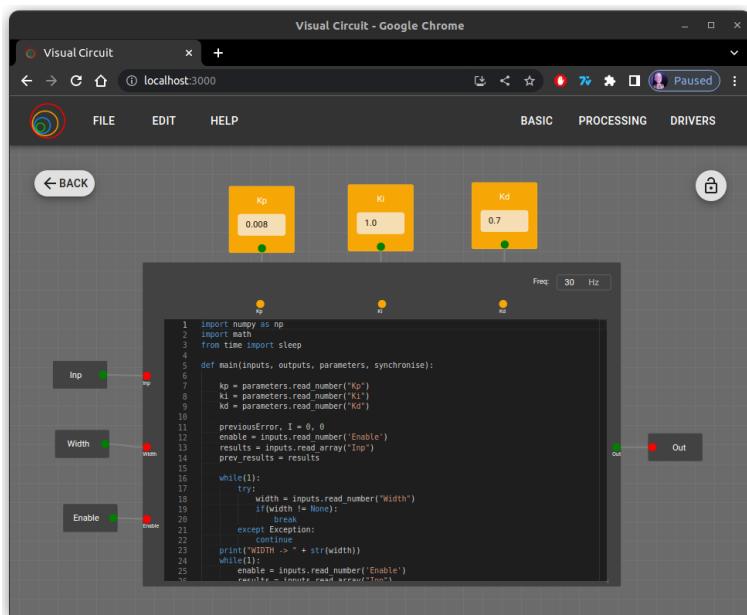


Figura 5.3: Circuito del bloque PID del sigue-persona en VisualCircuit.

Analizando el código del bloque,

---

<sup>4</sup>**PID:** Controlador proporciona, integral y derivativo.

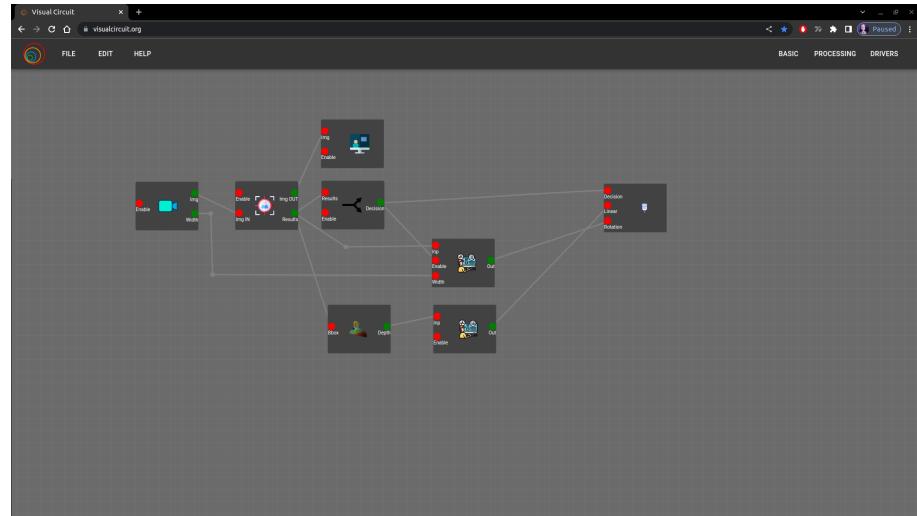


Figura 5.4: Circuito inicial del algoritmo sigue-persona.

---

# Capítulo 6

# Máquina de estados (VFF)

---

*Quizás algún fragmento de libro inspirador...*

Autor, Título

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.

## 6.1. 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).

## 6.2. 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
```

---

# Capítulo 7

# Conclusiones

---

*Quizás algún fragmento de libro inspirador...*

Autor, Título

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.

## 7.1. 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).

## 7.2. 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

---

- [1] I. Amazon.com. Imagen de la cámara asus-xtion., 2011.
- [2] S. Castro. Comunicación entre nodos intermedios, hardware/software y nodo master, 2017.
- [3] R. Components. Imagen del sensor laser rplidar a2., 2011.
- [4] I. Open Source Robotics Foundation. Imagen de un turtlebot2.
- [5] Robosavvy. Base kobuki para turtlebot2, 2022.
- [6] J. Vega. Navegación y autolocalización de un robot guía de visitantes. Master thesis on computer science, Rey Juan Carlos University, September 2008.
- [7] J. Vega. De la tiza al robot. Technical report, June 2015.
- [8] J. Vega. *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, September 2018.
- [9] J. Vega. JdeRobot-Kids framework for teaching robotics and vision algorithms. In *II jornada de investigación doctoral.* University of Alicante, June 2018.
- [10] J. Vega. El profesor Julio Vega, finalista del concurso 'Ciencia en Acción 2019'. URJC, on-line newspaper interview, July 2019.
- [11] J. Vega and J. Cañas. PyBoKids: An innovative python-based educational framework using real and simulated Arduino robots. *Electronics*, 8:899–915, August 2019.
- [12] J. Vega, E. Perdices, and J. Cañas. *Attentive visual memory for robot localization*, pages 408–438. IGI Global, USA, September 2012. Text not available. This book is protected by copyright.