



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.

En primer lugar, agradecer a mi tutor Jose María por tener tanta paciencia y haberme ayudado tanto a lo largo del proyecto.

A mis padres y hermana por ayudarme a seguir adelante pase lo que pase.

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. Por no dejarme agachar la cabeza incluso en los peores momentos. Por obligarme a levantarme tras cada caída. Por darme tanto sin siquiera darte cuenta.

Te quiero más que a mi vida, Koby. 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

FPS *Frames Per Second*

IMU *Inertial Measurement Unit*

LIDAR *Laser Imaging Detection and Ranging*

NASA *National Aeronautics and Space Administration*

ODE *Open Dynamics Engine*

PID *Proportional-Integral-Derivative controller*

RGBD *Red Green Blue - Depth*

ROS *Robot Operating System*

RVIZ *ROS Visualization*

URDF *Unified Robotics Description Format*

USB *Universal Serial Bus*

VFF *Visual Force Field*

Índice general

1. Introducción	1
1.1. Evolución histórica de la Robótica	1
1.1.1. Educación en Robótica	4
1.2. Programación de robots	5
1.2.1. Lenguajes de programación visuales	5
2. Objetivos y metodología de Trabajo	8
2.1. Descripción del problema	8
2.2. Requisitos	8
2.3. Metodología	8
2.4. Plan de trabajo	8
3. Herramientas y plataforma de desarrollo	10
3.1. Lenguaje de programación Python	10
3.2. ROS2 (Robot Operating System 2)	10
3.2.1. Visualizador RVIZ2	12
3.3. TurtleBot2	13
3.3.1. Base Kobuki	13
3.3.2. Cuerpo Turtlebot2	14
3.3.3. Cámara ASUS Xtion Pro	14
3.3.4. RPLIDAR A2	15
3.4. Simulador robótico Gazebo	15
3.4.1. TurtleBot2 simulado	16
3.5. VisualCircuit	17
4. Desarrollo de bloques driver	20
4.1. Bloques sensores	20
4.1.1. Bloque cámara ROS2	24
4.1.2. Bloque láser ROS2	26

4.2. Bloque MotorDriverROS2	27
5. Aplicación sigue personas	31
5.1. Desarrollo inicial sigue-personas (sólo rotación)	31
5.2. Modificaciones al sigue-personas para incluir movimiento lineal.	38
5.3. Pruebas en el robot real y resultado final.	42
6. Aplicación Visual Field Force	44
6.1. Campo de fuerzas virtuales (VFF)	44
6.1.1. Diseño del circuito y escenario	44
6.1.2. Bloques específicos	46
6.1.3. Validación experimental	51
6.2. VFF mediante máquina de estados	53
6.2.1. Diseño del circuito y escenario	53
6.2.2. Bloques específicos	54
6.2.3. Validación experimental	58
7. Conclusiones	60
7.1. Conclusiones	60
Bibliografía	61

Índice de figuras

1.1.	El Ajedrecista	1
1.2.	Robots Unimate y Shakey	2
1.3.	Robots Nao y Pepper	3
1.4.	Robots Mars Rovers	3
1.5.	Ejemplos aplicaciones actuales robótica	4
1.6.	Laboratorio Robótica	5
1.7.	Scratch	6
1.8.	Plataforma VisualCircuit	7
3.1.	Comunicación entre nodos ROS.	11
3.2.	RVIZ2 Vs mundo gazebo	12
3.3.	Kobuki base	13
3.4.	TurtleBot2	14
3.5.	Cámara ASUS-XTION	14
3.6.	RPLIDAR A2	15
3.7.	Simulador Gazebo.	16
3.8.	TurtleBot2 simulado	17
3.9.	VisualCircuit	17
3.10.	Creando un bloque en VisualCircuit	18
3.11.	Guardando un bloque en VisualCircuit	19
3.12.	Ejemplo de proyecto en VisualCircuit	19
4.1.	Modelo bloque driver sensores	21
4.2.	Estructura mensaje Image	24
4.3.	Secuencia bloque cámara ROS2 simulado	25
4.4.	Secuencia bloque cámara ROS2 real	25
4.5.	Estructura mensaje LaserScan	26
4.6.	Circuito pruebas bloque láser ROS2	27
4.7.	Ejemplo bloque láser ROS2	27

4.8. Estructura mensaje Twist	28
4.9. Secuencia bloque cámara ROS2 simulado	30
4.10. Secuencia bloque MotorDriverROS2 real	30
5.1. Modelo de persona en gazebo	32
5.2. Circuito sigue-personas inicial	32
5.3. Circuito del bloque PID sigue-persona	35
5.4. Secuencia sigue-personas rotación	38
5.5. Circuito sigue-personas inicial	39
5.6. Estructura mensaje PointCloud2	40
5.7. Secuencia sigue-personas resultado final	43
6.1. Circuito VFF	45
6.2. Mundo para VFF	45
6.3. Estructura mensaje Odometría	48
6.4. Circuito VFF sólo fuerza repulsiva	51
6.5. Secuencia VFF fuerza repulsiva sin obstáculos	51
6.6. Secuencia bloque MotorDriverROS2 real	52
6.7. Circuito VFF	52
6.8. Secuencia prueba VFF	52
6.9. Diagrama máquina de estados	53
6.10. Circuito VFF con FSM	54
6.11. Mundo VFF con FSM	54
6.12. Secuencia VFF con FSM	59

Listado de códigos

4.1.	Modelo de código para bloques drivers	22
4.2.	Clase del nodo suscriptor para bloques drivers	22
4.3.	Función main para bloques drivers de sensores	23
4.4.	Clase del nodo suscriptor para cámara	24
4.5.	Cambios main bloque cámara	25
4.6.	Clase del nodo suscriptor para láser	26
4.7.	Cambios main bloque láser	27
4.8.	Bloque MotorDriverROS2	29
5.1.	Modificación al bloque detector de objetos	33
5.2.	Código bloque decisión sigue-persona	34
5.3.	Código bloque rotación sigue-persona	35
5.4.	Código bloque PID sigue-persona	36
5.5.	Código bloque MotorDriver sigue-persona	37
5.6.	Código bloque MotorDriver sigue-persona modificado	39
5.7.	Código bloque Camera-Depth sigue-persona	41
5.8.	Código bloque PID lineal sigue-persona	42
5.9.	Comandos para lanzar kobuki y cámara	43
6.1.	Funciones para obtener la fuerza repulsiva	47
6.2.	Funciones para obtener la fuerza repulsiva	48
6.3.	Bloque generador de ubicaciones	49
6.4.	Bloque fuerza atractiva	50
6.5.	Bloque fuerzas a velocidades	50
6.6.	Bloque generador aleatorio de destinos	56
6.7.	Bloque return home	57
6.8.	Bloque forces to vels	58

Listado de ecuaciones

Índice de cuadros

Capítulo 1

Introducción

La robótica es la ciencia encargada del estudio, diseño, fabricación y utilización de robots, combinando mecánica, electrónica e informática. La palabra robot viene del término *robota* que, traducido del checoslovaco, sería algo similar a trabajo forzado. Hoy en día se define como un sistema que utiliza una serie de elementos *hardware* (sensores, actuadores y procesadores) y que está controlado por un *software* para realizar una tarea concreta.

1.1. Evolución histórica de la Robótica

Aunque el término *robot* apareció en los años 20, los autómatas, que son máquinas que imitan la figura y movimientos de un ser animado, existían desde mucho antes. Varios ejemplos de ellos son el robot de Leonardo, un autómata humanoide diseñado por Leonardo Da Vinci en 1495 que no llegó a ser construido, o el ajedrecista que Leonardo Torres Quevedo construyó en 1912 que, usando electroimanes por debajo del tablero, era capaz de jugar distintos finales simples (con pocas piezas en el tablero) contra humanos, consiguiendo siempre la victoria.

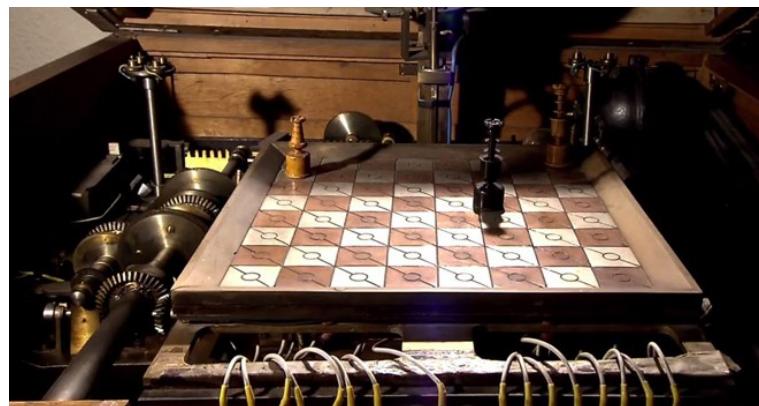


Figura 1.1: El Ajedrecista. Imagen obtenida de [Hostalia, 2017]

En la segunda mitad del siglo XX, con el gran avance de los ordenadores, se

empiezan a ver los primeros robots tal y como se conocen hoy en día. De esta época debemos destacar al robot industrial que desarrolló la compañía Unimate en 1952, al igual que el robot Shakey, un pequeño robot que apareció en 1972 capaz de navegar y evitar obstáculos en una habitación cerrada sin interacción humana.

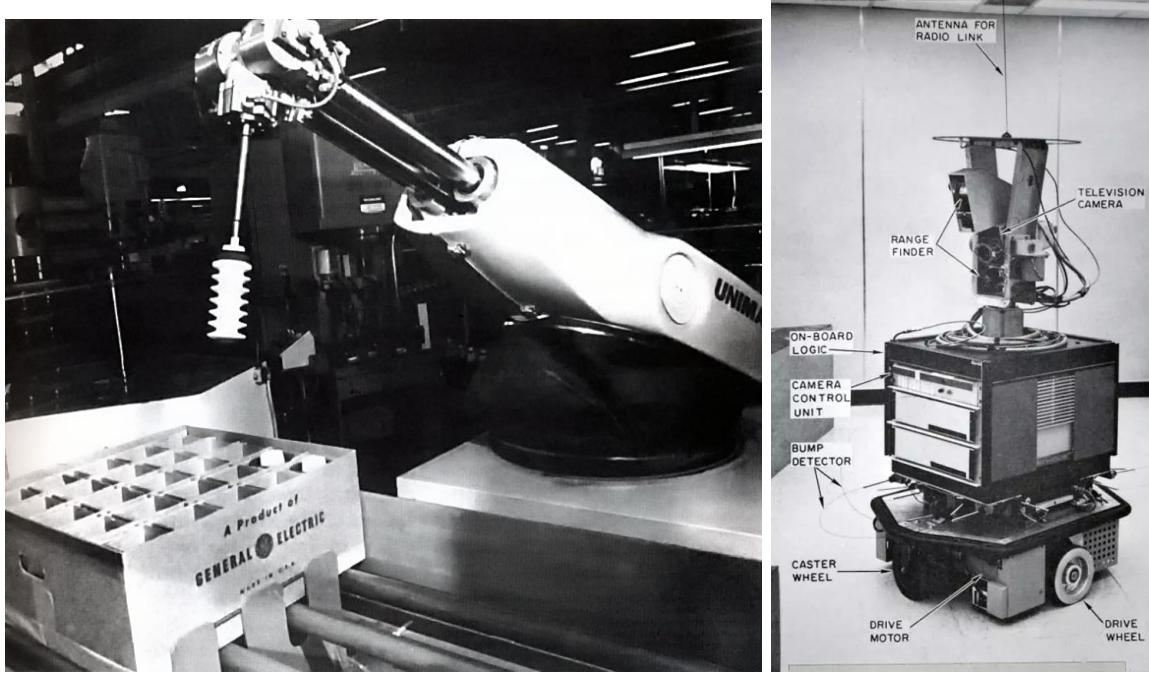


Figura 1.2: Robots Unimate y Shakey. Imágenes obtenidas respectivamente de [RobotsInAction, 2019] y [elDiario.es, 2017]

En el 2000, Honda presenta su robot ASIMO (*Advanced Step in Innovative Mobility*), un humanoide que demostró un gran avance en técnicas complejas como caminar y correr a velocidades de hasta 9km/h.

A partir de él surgieron varios robots humanoides con nuevas tecnologías, como *QRIO* de *Sony* en 2004 siendo capaz de reconocer caras, el pequeño robot *Nao* en 2008 con su habilidad para interactuar con el ser humano o *Pepper*, un robot con forma humana pero que se desplaza con ruedas que apareció en 2014 y que se usaba sobre todo como guía o recepcionista, hasta que su desarrollo y producción se abandonó en 2021.

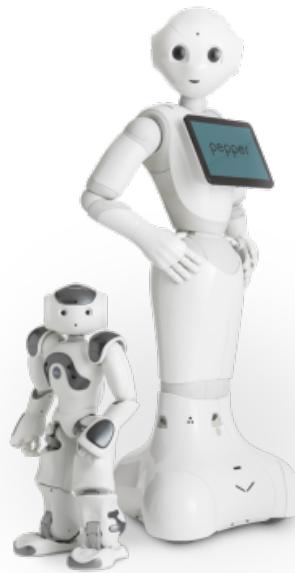


Figura 1.3: Robots Nao y Pepper. Imagen obtenida de [Robotics24/7, 2022]

Parejo a estos robots, la NASA estaba desarrollando sus propios robots para mandar a Marte. El *MARS-ROVER*, una plataforma móvil con un brazo mecánico, sensores de proximidad, láser y cámaras, salió a la luz ya en los años 70. Finalmente su sucesor, el *Sojourner Rover* fue el primero en aterrizar en el planeta rojo en 1997. En 2004, se lanzaron el *Spirit* y el *Oportunity* con el objetivo de encontrar evidencia de agua, contando con muchos más sensores e instrumentos científicos que sus predecesores. Las últimas misiones, *Curiosity* y el *Perseverance*, que aterrizaron en 2012 y 2021 respectivamente, tenían el objetivo de buscar indicios de vida en Marte, tanto pasada como presente, a la vez que comprobar si el desarrollo de vida humana sería posible.

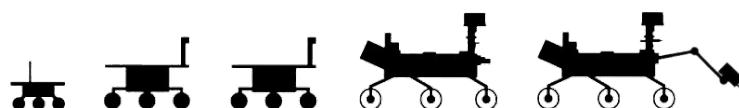


Figura 1.4: Robots Mars Rovers.

Hoy en día, el avance de la robótica ha llegado a una gran variedad de aplicaciones distintas. Entre ellas, podríamos destacar las aplicaciones domésticas con robots de limpieza como los famosos *Roomba* o iRobot, el sector de la agricultura con vehículos autónomos y monitorización de cultivos, en logística tanto para organizar las mercancías dentro de almacenes como para el reparto, e incluso para conducción autónoma con empresas como Tesla o Google desarrollando sus propios vehículos que

son capaces de conducir grandes distancias sin interacción humana.

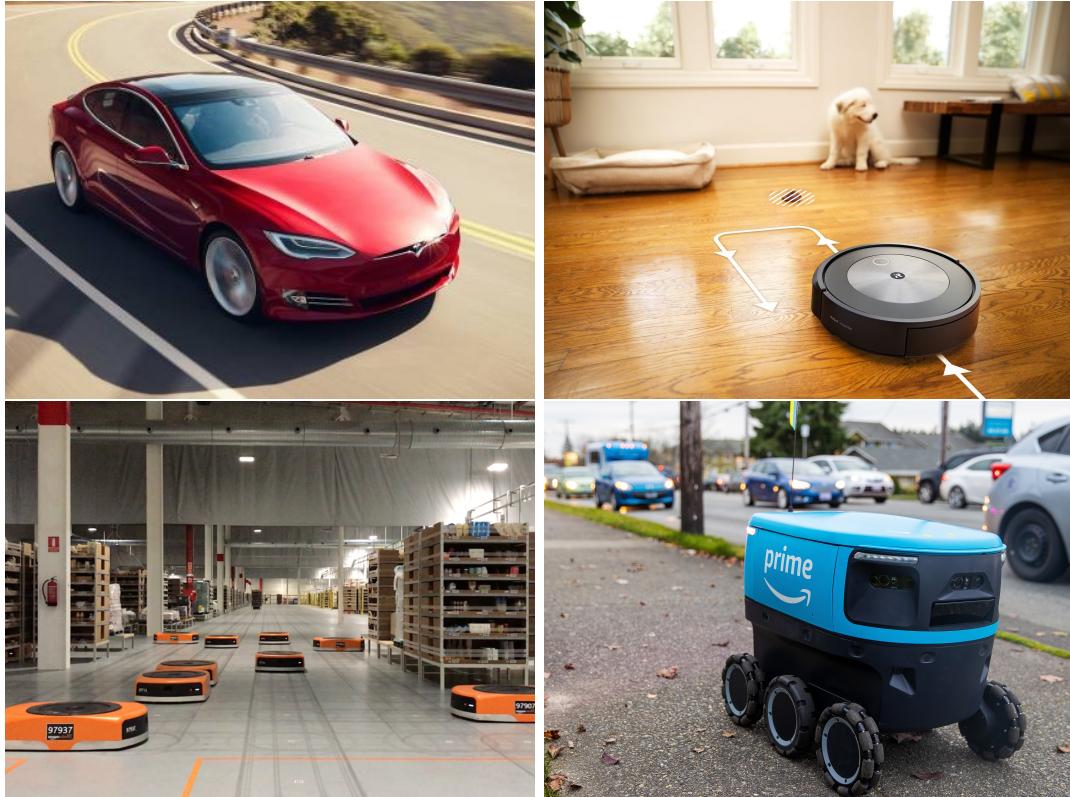


Figura 1.5: Ejemplos de aplicaciones actuales de la robótica.

1.1.1. Educación en Robótica

Actualmente la robótica es un mercado en alza, lo que hace que la cantidad de expertos en el sector sea escasa.

Los conocimientos de programación es algo que, hasta hace poco, se consideraba algo de nicho y que sólo se enseñaba en algunas universidades, haciendo que su avance y desarrollo sea lento. Hoy en día se considera algo tan fundamental, que incluso en algunas escuelas primarias se comienzan a desarrollar los conocimientos en torno a la programación y la robótica con niños de 5 años en las aulas y en talleres.

El gran avance en la robótica ha causado una gran demanda de profesionales y, gracias a ello, han surgido grados universitarios como el grado en Ingeniería de Robótica Software, impartido por la Universidad Rey Juan Carlos en el campus de Fuenlabrada. Como indica su nombre, este grado está orientado mayoritariamente a la programación, con lenguajes como *python*, *C++* o *Java*, y abordando temas como inteligencia artificial, ciberseguridad o visión artificial entre otros, todo esto sin dejar de lado el apartado físico de la robótica, con asignaturas como sensores y actuadores o

mecatrónica, donde se enseña a crear robots desde cero.

Este grado también da acceso a los estudiantes al laboratorio de robótica¹, donde podrán programar usando robots reales como el robot Pepper o el TurtleBot23.3 usado en este trabajo.



Figura 1.6: Laboratorio de Robótica ETSIT.

1.2. Programación de robots

El *software* de los robots ha ganado cada vez más importancia a medida que las tareas a realizar se vuelven más complejas. Gran parte del avance en la robótica se debe a las herramientas que facilitan el desarrollo de este software, como el *middleware* robótico. El *middleware* más extendido en el mundo de la robótica es ROS (3.2). Este nos ofrece una gran variedad de herramientas, desde abstracción del hardware, hasta comunicación entre distintas partes del robot. Para usar ROS, lo más común es usar lenguajes de programación como *python3* o *C++*, aunque ésta no es la única forma de programar robots.

1.2.1. Lenguajes de programación visuales

Un lenguaje de programación visual es aquel que permite a los usuarios crear software mediante elementos gráficos y no únicamente mediante texto, como ocurre con los lenguajes de programación tradicional.

¹Laboratorio de robótica: <https://labs.eif.urjc.es/index.php/laboratorios/edificio-laboratorio-iii/laboratorios-laboratorio-13104/>

Un gran ejemplo de este tipo de programación es *Scratch*². Esta plataforma nos permite programar el comportamiento de imágenes conocidas como *sprites* mediante el uso de bloques simples para crear historias interactivas, animaciones o incluso juegos, permitiendo a los usuarios compartir sus creaciones e investigar cómo lo hacen otros. Esta plataforma es muy usada en entornos académicos (primaria y secundaria) como introducción a la programación por su simpleza a la hora de entender conceptos básicos como bucles o condicionales.

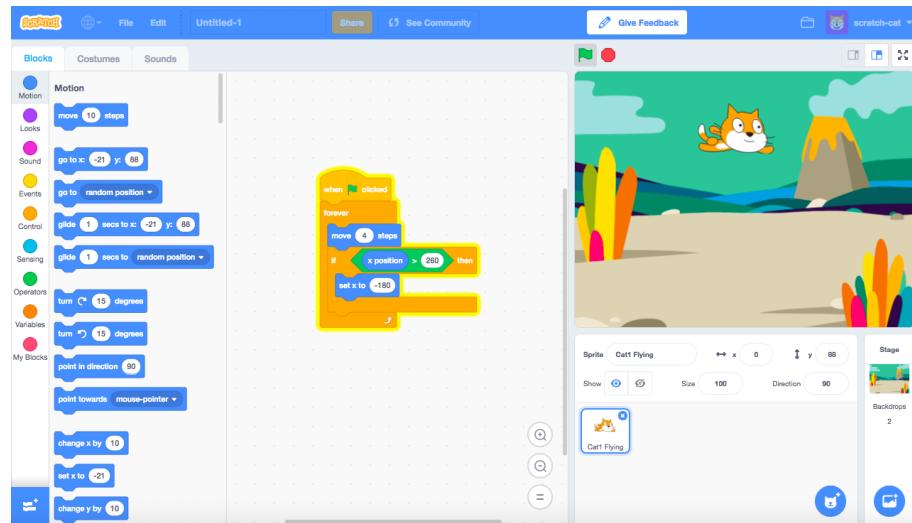


Figura 1.7: Plataforma Scratch.

VisualCircuit, la plataforma en la que se basa este trabajo de fin de grado, también utiliza la programación visual mediante el uso de bloques que se pueden colocar y unir mediante cables para crear circuitos complejos. Estos cables envían información entre bloques, desde simples mensajes de texto hasta imágenes o matrices de valores. La componente visual permite entender el funcionamiento del software sin necesidad de ver cada bloque por dentro. VisualCircuit nos permite crear aplicaciones robóticas de manera rápida y sencilla sin necesidad de tener grandes conocimientos de programación o robótica gracias a las librerías de bloques prefabricados que ofrece, destacando bloques de sensores y actuadores (láser, cámara, motores...) o bloques de edición de imágenes.

²Scratch: <https://scratch.mit.edu/>

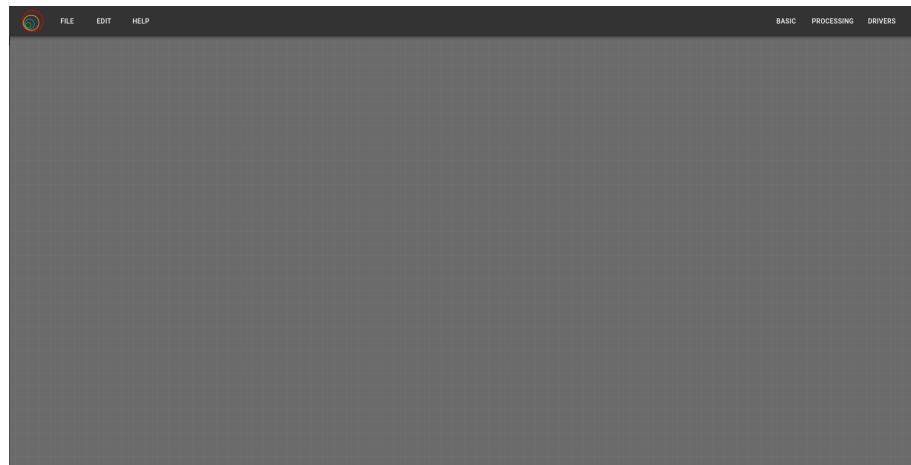


Figura 1.8: Plataforma VisualCircuit.

En el apartado 3.5 se profundizará en el funcionamiento de esta plataforma.

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

COSAS GUARDADAS PARA HACER

- En la revision del cap 4 pone cómo hacer este cap
- Cacho guardado para hacer el capitulo2.

***** En VisualCircuit, antes de realizar este proyecto, ya existían bloques dedicados específicamente a la robótica para algunos sensores (cámara, odometría e IMU¹) y para los motores usando ROS Noetic², pero al tratarse de una versión antigua, decidimos que ya era momento de actualizar a ROS2 Humble³, ya que era la versión estable más moderna hasta el momento. *****

¹**IMU**: Inertial Measurement Unit

²**ROS Noetic**: <http://wiki.ros.org/noetic>

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

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 middleware, ROS2, simulador robótico Gazebo..., por lo que se va 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

Python¹ 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.

Python cuenta con un gran número de bibliotecas y módulos que se usarán a lo largo de las distintas prácticas realizadas, como la librería *math* que permite usar operaciones matemáticas complejas (senos, cosenos, generación de números pseudo-aleatorios...).

Este lenguaje de programación cuenta con varias versiones. La que se usará durante el TFG será python 3.8, ya que la programación dentro de la plataforma VisualCircuit (3.5) se hace en este lenguaje.

3.2. ROS2 (Robot Operating System 2)

ROS² o Robot Operating System es un *middleware*³ formado por un conjunto de herramientas y librerías de software libre empleadas para el desarrollo de aplicaciones robóticas. Su objetivo es ofrecer una plataforma de programación estándar para todas

¹Python ORG: <https://www.python.org/>

²ROS: <http://wiki.ros.org/es>

³Middleware: software que se sitúa entre las aplicaciones y el sistema operativo

las ramas de la robótica.

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.

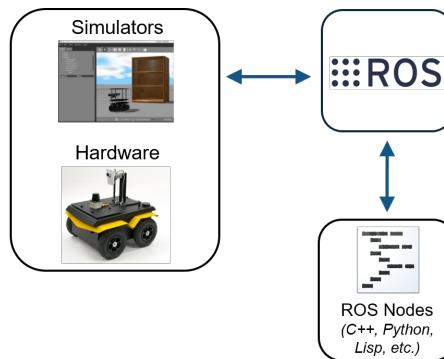


Figura 3.1: Comunicación del nodo Master con los nodos Intermedios y con distintos sensores y actuadores. Imagen obtenida de [Castro, 2017]

ROS cuenta con dos versiones principales, ROS y ROS2⁴. ROS está pensado para usarse únicamente en sistemas Ubuntu, mientras que ROS2 también se puede usar en OS X y Windows. En cuanto a lenguajes de programación, ROS usa C++03 y python2 (aunque se puede cambiar reduciendo su optimización), frente a los lenguajes de ROS2 que permite usar tanto C++11, C++14 y python3.5 (versión mínima). Dentro de ambas versiones hay distintas distribuciones, como por ejemplo ROS Kinetic, Melodic o Noetic, o ROS2 Foxy, Galactic o Humble.

A lo largo del proyecto se usará ROS2 Humble para obtener información del robot TurtleBot2 (3.3), 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).

⁴Versiones de ROS: <https://docs.ros.org/>

3.2.1. Visualizador RVIZ2

ROS2 cuenta con su propio depurador pensado para funcionar con topics. Esta herramienta es RVIZ2⁵, una herramienta de visualización 3D que nos permite interpretar gráficamente las medidas de los sensores, así como la información de otros topics (ubicación del robot o de otros objetos, mapas 2D y 3D...).

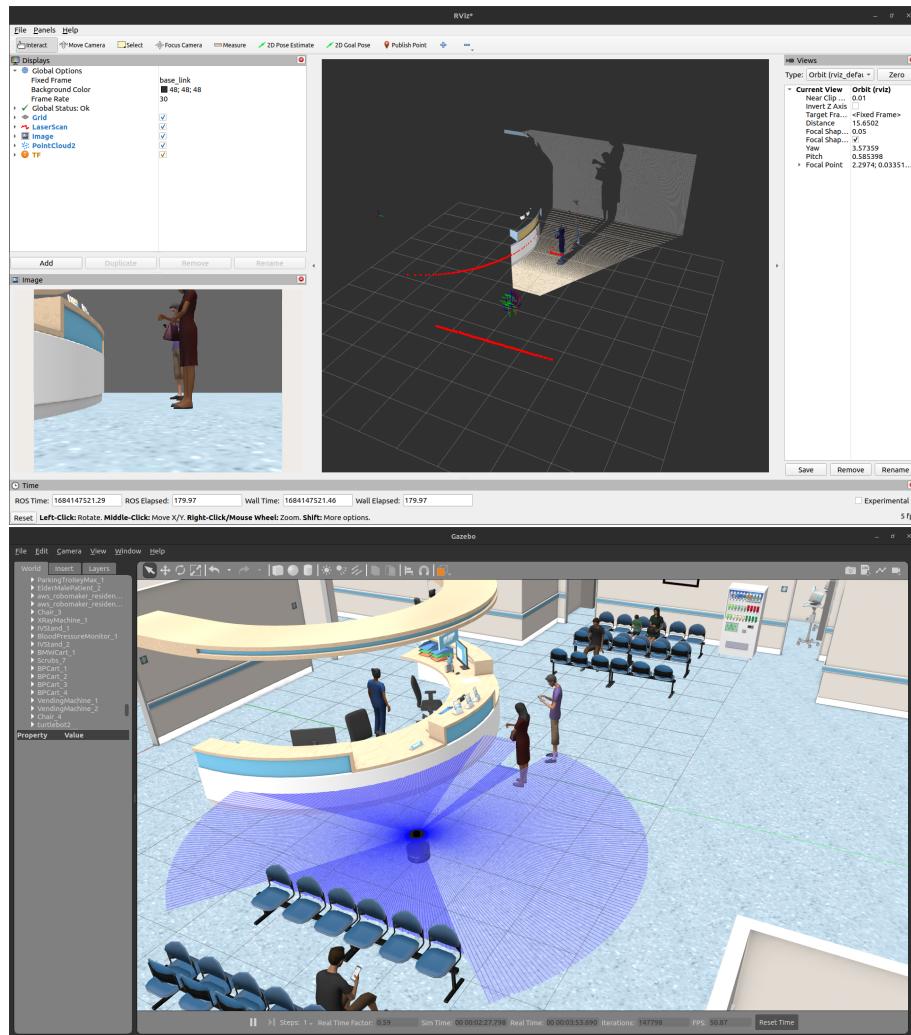


Figura 3.2: Ejemplo de RVIZ2 frente al mundo gazebo.

⁵RVIZ2: <https://github.com/ros2/rviz/tree/humble>

3.3. TurtleBot2

La URJC de Fuenlabrada, en sus laboratorios de robótica, cuenta con varios robots TurtleBot2⁶ 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.

3.3.1. 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.

Algunos de los paquetes de ROS2 que instalaremos para poder usarlo son los drivers del kobuki para ROS2-Humble⁷ 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 [Robosavvy, 2022]

⁶TurtleBot2: <https://www.turtlebot.com/turtlebot2/>

⁷Drivers kobuki ROS2-Humble: <https://github.com/IntelligentRoboticsLabs/Robots/tree/humble/kobuki>

3.3.2. 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 [Open Source Robotics Foundation,]

3.3.3. Cámara ASUS Xtion Pro

La cámara *ASUS Xtion* es una cámara RGB-D⁸, 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 70º. Se conecta mediante USB directamente al ordenador.

En el proyecto, como debemos usarla con ROS2, usaremos el paquete creado por un usuario de internet⁹.



Figura 3.5: Cámara ASUS-XTION. Imagen obtenida de [Amazon.com, 2011]

⁸**RGB-D:** RedGreenBlue-Depth, hace referencia las cámaras que captan la imagen y las distancias de cada pixel.

⁹**Drivers ASUS-Xtion ROS2:** https://github.com/mgonzs13/ros2_asus_xtion

3.3.4. RPLIDAR A2

Se trata de un láser de 360º 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 (3.3.1) encontraremos un paquete para poder activar y usar este sensor con ROS2.



Figura 3.6: Sensor RPLIDAR A2. Imagen obtenida de [Components, 2011]

3.4. Simulador robótico Gazebo

*Gazebo*¹⁰ 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 ODE¹¹, aunque se puede configurar con otros motores, como Bullet¹² o DART¹³.

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).

Actualmente Gazebo cuenta con dos versiones principales, *Gazebo classic*¹⁴ e *ignition Gazebo*¹⁵. La versión que se usará será *Gazebo11*, la última versión de *Gazebo classic*.

¹⁰**Gazebo:** <https://classic.gazebosim.org/>

¹¹**Open Dynamics Engine:** <https://www.ode.org/>

¹²**Bullet:** <https://pybullet.org/wordpress/>

¹³**DART:** <https://dartsim.github.io/>

¹⁴**Gazebo classic:** <https://classic.gazebosim.org/>

¹⁵**Ignition gazebo:** <https://gazebosim.org/home>

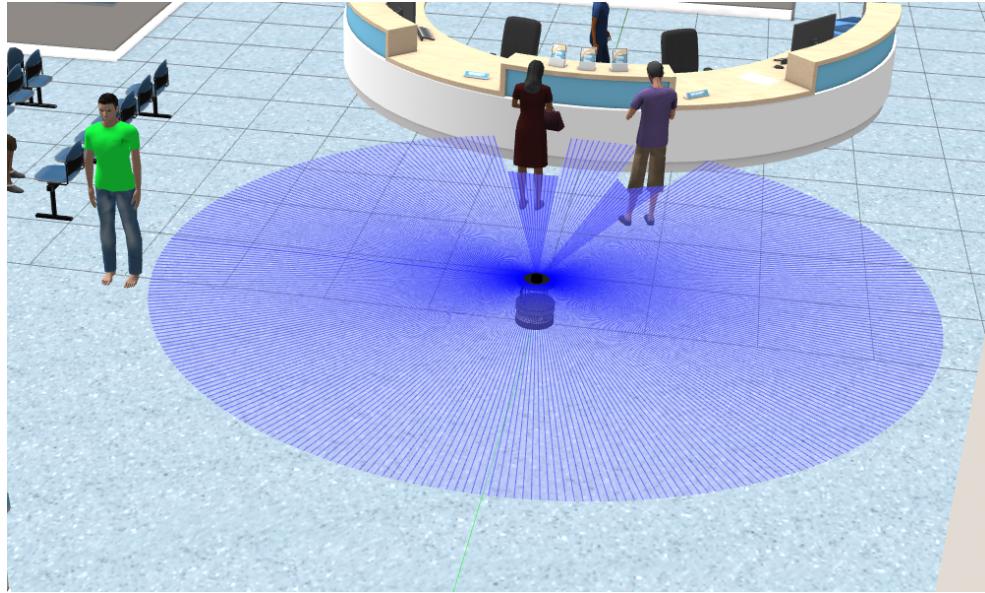


Figura 3.7: Ejemplo de ejecución en gazebo.

3.4.1. TurtleBot2 simulado

Para algunas partes del proyecto, como el desarrollo de *drivers* para ROS2 (4) o el comportamiento VFF usando máquinas de estados (6), el simulador ha servido para probar y desarrollar los algoritmos. Para esto, se ha usado un modelo del TurtleBot2 que cuenta con los mismos sensores (cámara, *RPLIDAR*, *bumper*, etc) que el real, así como los mismos *topics*.

Este modelo del robot se obtiene del repositorio de *RoboticsInfrastructure*¹⁶, donde podemos acceder los modelos *URDF*¹⁷ tanto de la base *kobuki*¹⁸ como del cuerpo del Turtlebot¹⁹ (incluyendo los sensores cámara y láser).

¹⁶**Robotics Infrastructure:** https://github.com/JdeRobot/RoboticsInfrastructure/tree/humble-devel/CustomRobots/Turtlebot2/turtlebot2_simulated

¹⁷**URDF:** Unified Robot Description Format

¹⁸**kobuki_description:** https://github.com/JdeRobot/RoboticsInfrastructure/tree/humble\protect\discretionary{\char\hyphenchar\font}{}{}devel/CustomRobots/Turtlebot2/turtlebot2_simulated/kobuki_description

¹⁹**TurtleBot2:** https://github.com/JdeRobot/RoboticsInfrastructure/tree/humble\protect\discretionary{\char\hyphenchar\font}{}{}devel/CustomRobots/Turtlebot2/turtlebot2_simulated/turtlebot2

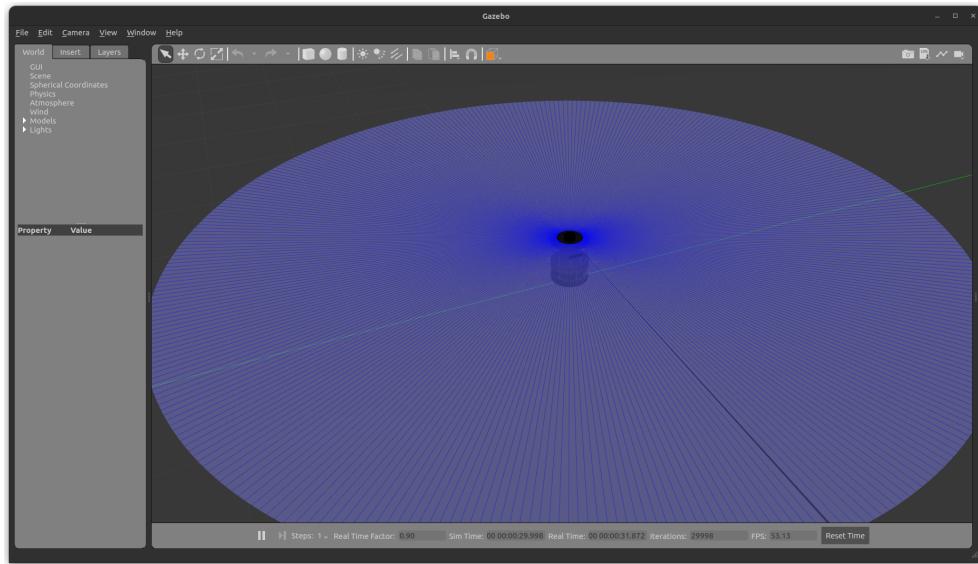


Figura 3.8: TurtleBot2 en gazebo.

3.5. VisualCircuit

VisualCircuit²⁰ es un editor visual online basado en programación por bloques de código en lenguaje *python* orientado al desarrollo de aplicaciones robóticas. Está desarrollado sobre IceStudio²¹.

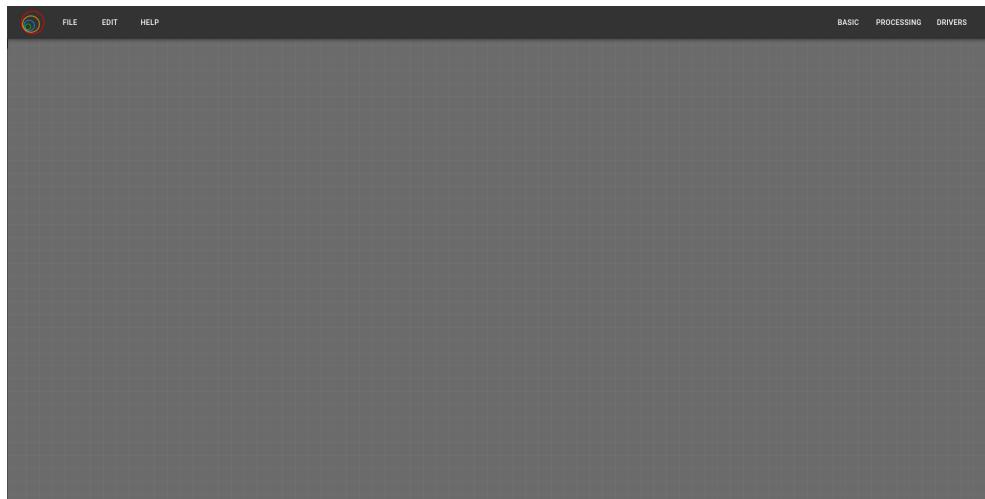


Figura 3.9: Página de VisualCircuit.

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

²⁰VisualCircuit Docs: <https://jderobot.github.io/VisualCircuit/>

²¹IceStudio Project: <https://github.com/FPGAwars/icestudio>

- *Basic*: bloques simples, como inputs y outputs, bloques para definir parámetros y constantes, o bloques para insertar código propio.
- *Processing*:
 - *Control*: bloques de control (PID).
 - *OpenCV*: bloques relacionados con OpenCV²² y edición de imagen (filtros de color, detección de contornos, erosión, etc).
 - *TensorFlow*: un bloque para detección de objetos.
- *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 IMU²³ usando ROS.

Para crear un bloque propio, se deben añadir varios bloques prefabricados. Para introducir el código principal se usa el bloque genérico, como se puede ver en la figura 3.10. Al crearlo, se permite definir el número de entradas, salidas y parámetros que tendrá el código.

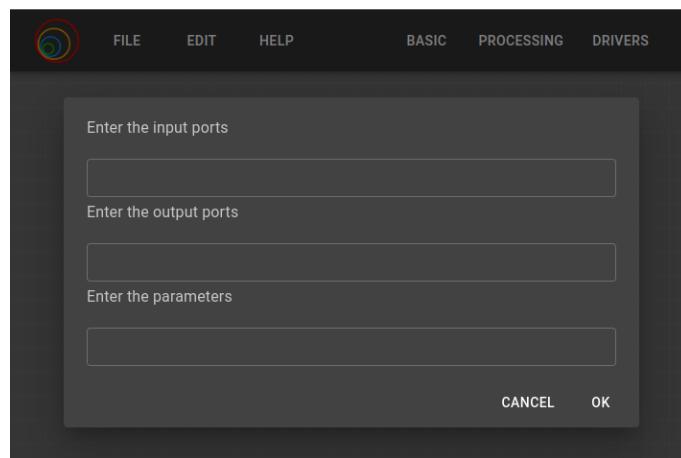


Figura 3.10: Creando un bloque en VisualCircuit.

Una vez definidos, se deben usar bloques de *Input*, *Output* y *Constant* y así, pulsando en “Save as”, se exporta para usarlo como un bloque nuevo.

²²OpenCV: <https://opencv.org/>

²³IMU: Inertial Measurement Unit

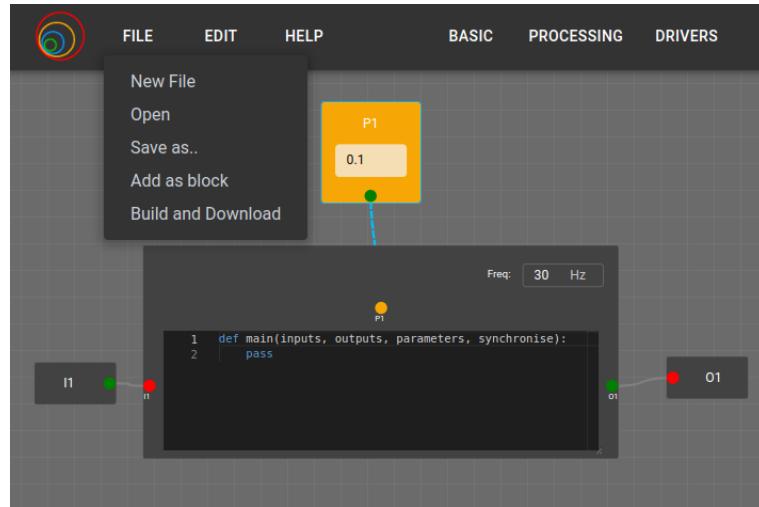


Figura 3.11: Guardando un bloque en VisualCircuit.

Cuando se hayan generado varios bloques, se puede usar la opción *FILE* → *Add as block* para añadirlos como nuevos bloques y así formar un circuito completo con el comportamiento que se deseé.

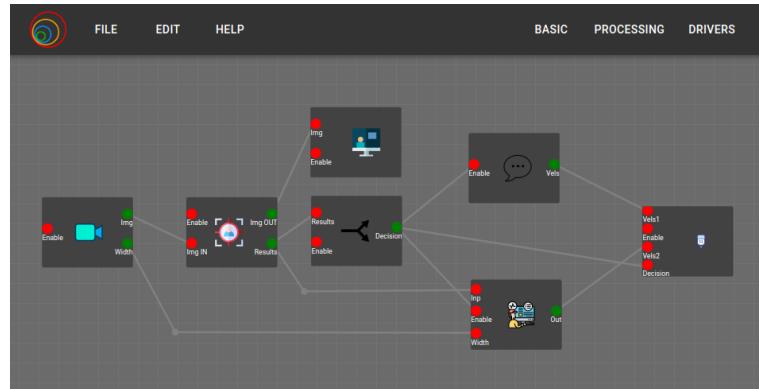


Figura 3.12: Ejemplo de proyecto en VisualCircuit.

Cuando se empezó este proyecto de fin de grado, la versión activa era la v3.4, aunque ha ido avanzando y actualizándose durante el desarrollo del mismo.

Capítulo 4

Desarrollo de bloques driver

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

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

Los bloques drivers que se van a implementar son para la cámara, el láser y los motores usando un topic de ROS2, ya sea para suscribirse (sensores) o para publicar (actuadores).

A continuación se explica el proceso seguido para desarrollar cada uno de los drivers.

4.1. Bloques sensores

Para desarrollar los bloques correspondientes a los sensores, primero una plantilla para sensores y después implementaremos cada uno de los sensores a partir de ella. Esta plantilla va a consistir de un *input* para activar/desactivar el bloque, habilitando el uso de máquinas de estados con nuestro bloque, un *output* para compartir la medida del sensor con otros bloques y una constante donde se definirá el *topic* permitiendo al usuario cambiarlo sin modificar el código del bloque y habilitando su uso tanto para robots simulados como reales.

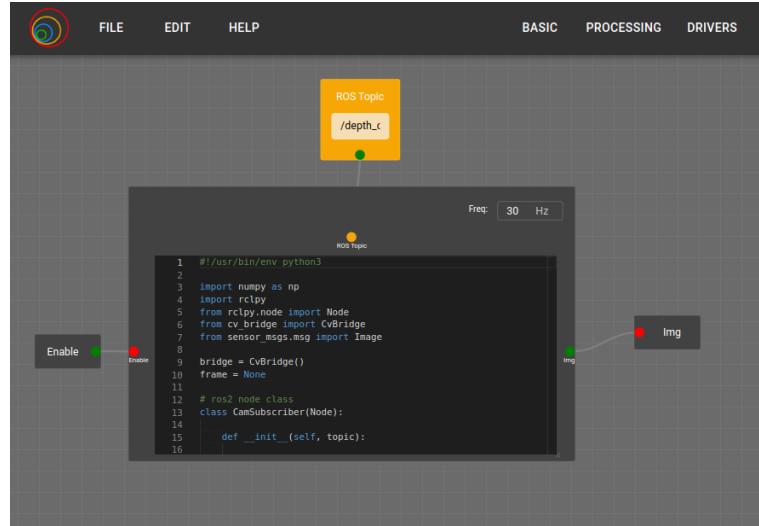


Figura 4.1: Modelo para crear bloques driver de sensores.

En cuanto al código que usaremos, seguiremos las indicaciones de los manuales de ROS2-humble¹ para usar nodos suscriptores/publicadores con python. El código general para los bloques de sensores (suscriptores) será el siguiente:

```

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

```

¹<https://docs.ros.org/en/humble/Tutorials/Beginner-Client-Libraries/Writing-A-Simple-Py-Publisher-And-Subscriber.html#write-the-subscriber-node>

```

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 máquina 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 = SENSORSSubscriber(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)

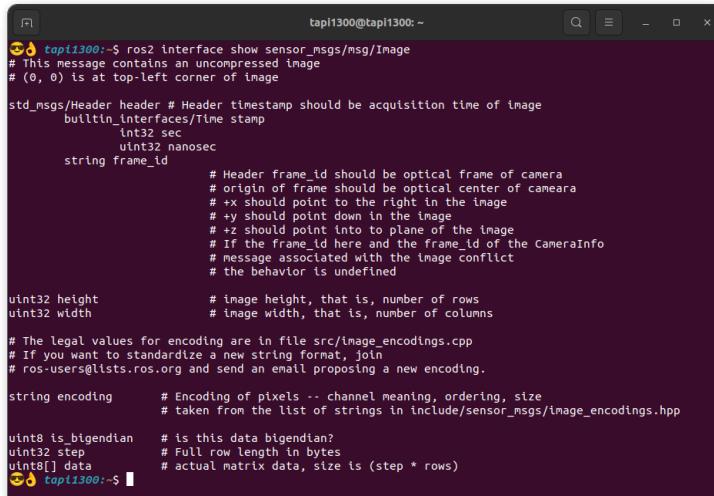
```

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.

4.1.1. Bloque cámara ROS2

Para el bloque cámaraROS2, el mensaje es del tipo *sensor_msgs.msg.Image* que, usando el comando “`ros2 interface show sensor_msgs/msg/Image`”, se puede ver cuál es su estructura:



```
tapi1300@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 4.2: Estructura del tipo de mensaje *sensor_msgs/msg/Image*.

Como se observa en la figura 4.2, la imagen que obtendremos estará en el campo *data* del mensaje. Los hilos de VisualCircuit comparten las imágenes como un array de numpy, por lo que será necesario convertir la imagen a imagen de numpy y después a un array de numpy. Para ello, ROS tiene una función llamada *CvBridge*², que permite transformar un mensaje del tipo *sensor_msgs/msg/Image* a una imagen de numpy³, que se pasará a array de numpy usando la función *numpy.asarray()*.

```
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.

²CvBridge: http://wiki.ros.org/cv_bridge

³Numpy: <https://numpy.org/>

En cuanto a la función main, sólo habría que cambiar la función que se usa para compartir para adaptarla al tipo de mensaje, en este caso 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.

Finalmente, se realizaron pruebas tanto en simulador como en el robot real para probar que el bloque funcionase correctamente, como se puede ver en las siguientes capturas:

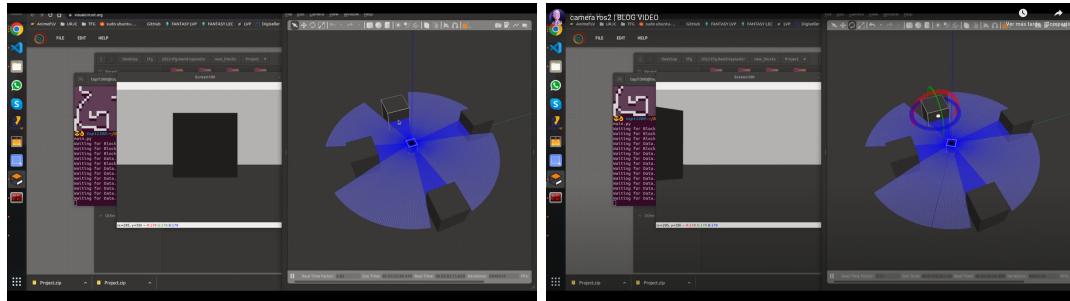


Figura 4.3: Secuencia de imágenes del bloque cámara ROS2 con TurtleBot2 simulado. Imagenes obtenidas de Youtube⁴.

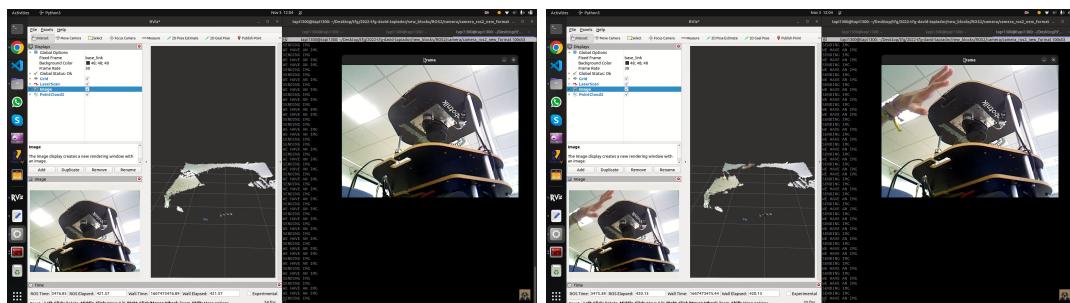


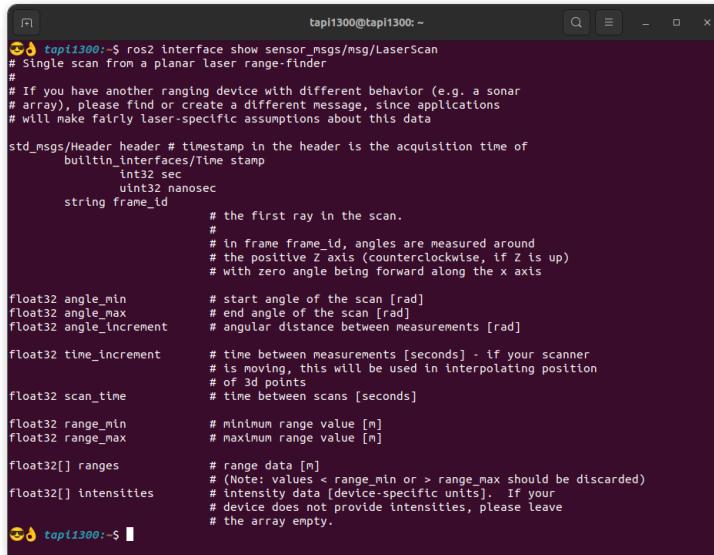
Figura 4.4: Secuencia de imágenes del bloque cámara ROS2 con TurtleBot2 real. Imagenes obtenidas de Youtube⁵.

⁴Vídeo bloque cámara simulación: https://www.youtube.com/watch?v=a5J6Qccc5xk&t=73s&ab_channel=Tapi1300

⁵Vídeo bloque cámara real: https://www.youtube.com/watch?v=MNaFWD9-ats&ab_channel=Tapii

4.1.2. Bloque láser ROS2

Para el láser, se seguirá un proceso similar al que se ha usado con la cámara. Para ello se comienza revisando la estructura del tipo de mensaje, e 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
                       # built-in 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)
                           # intensity data [device-specific units]. If your
                           # device does not provide intensities, please leave
                           # the array empty.
```

Figura 4.5: Estructura del tipo de mensaje `sensor_msgs/msg/LaserScan`.

Se observa en la figura 4.5 que la lectura del láser estará en el campo `ranges` del mensaje, que es un *array* de *floats*, por lo que para guardarla en un array local se usará la función de python “`.extend()`”, que permite añadir una entrada más a un array. Esto se hará en el callback, ya que es donde se actualiza 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 `share`, adaptándola de nuevo al tipo de mensaje, en este caso 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.

Para probar el correcto funcionamiento del bloque, se ha creado un circuito simple en el que se reciban los mensajes del topic del láser y se impriman en una terminal los valores recibidos, tanto con el robot real como con el simulado:

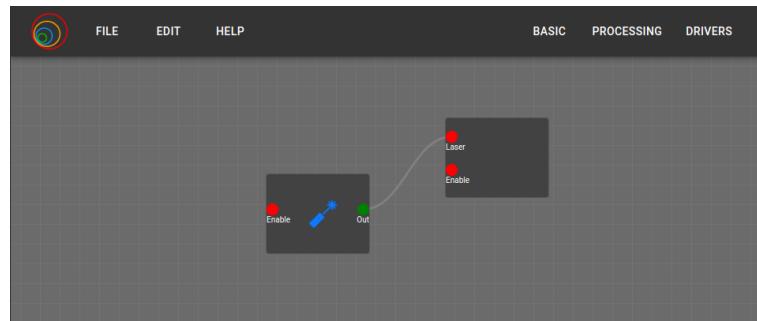


Figura 4.6: Circuito de pruebas del bloque laserROS2.

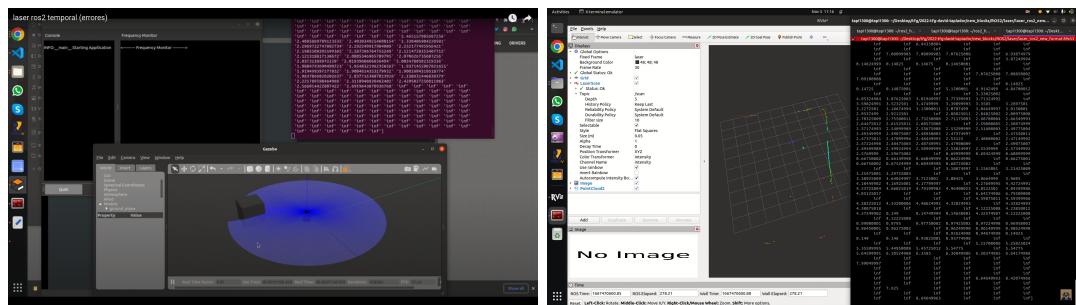


Figura 4.7: Ejemplo del bloque laserROS2 con TurtleBot2 simulado y real. Imagenes obtenidas de Youtube⁶.

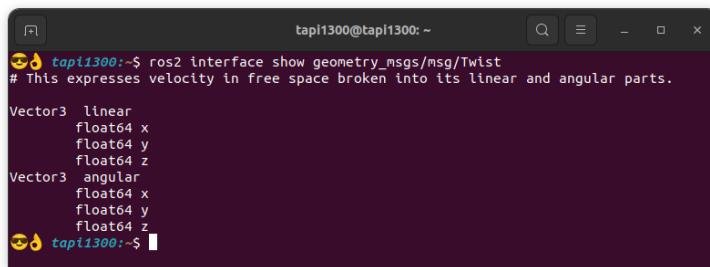
4.2. Bloque MotorDriverROS2

Para desarrollar el bloque MotorDriverROS2 la configuración será distinta de los anteriores, ya que no será necesaria una salida sino una entrada para recibir las

⁶Vídeo bloque láser real: https://www.youtube.com/watch?v=-MweaxUVsCg&ab_channel=Tapii

velocidades que mandar al robot. Por esto, habrá 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 se deben enviar.

El tipo de mensaje que suelen admitir los robots para su movimiento es “geometry_msgs/msg/Twist”, y si revisamos su estructura usando “`ros2 interface show geometry_msgs/msg/Twist`”, nos encontraremos lo siguiente:



```
tapi1300@tapi1300: ~
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 4.8: Estructura de tipo de mensaje *geometry_msgs/msg/Twist*.

Como se puede ver en la figura 4.8, 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 se permite 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 se hizo con el suscriptor, a los manuales de ROS2-humble⁷ y a partir de ahí crear el nodo publicador necesario para este bloque.

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

```

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

# 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 se puede 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.

Al igual que con los bloques de los sensores, se han realizado pruebas del bloque tanto con el robot simulado como con el robot TurtleBot2 real para comprobar que en ambos escenarios funcionase sin problemas. En este caso, se han combinado el bloque de la cámara con el bloque MotorDriverROS2 para que se aprecie mejor el movimiento.

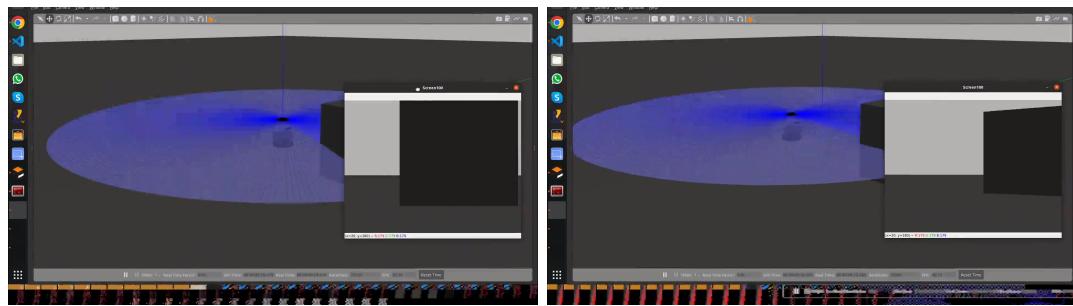


Figura 4.9: Secuencia de imágenes del bloque MotorDriverROS2 con TurtleBot2 simulado. Imagenes obtenidas de Youtube⁸.

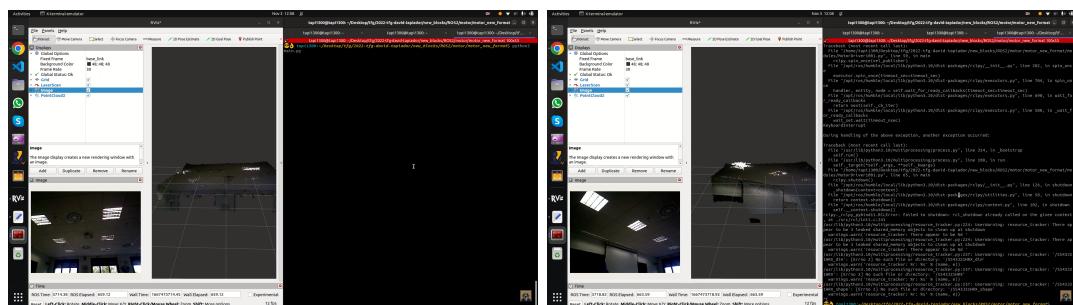


Figura 4.10: Secuencia de imágenes del bloque MotorDriverROS2 con TurtleBot2 real. Imagenes obtenidas de Youtube⁹.

⁸Vídeo bloque MotorDriverROS2 simulación: https://www.youtube.com/watch?v=a5J6Qccc5xk&t=73s&ab_channel=Tapi1300

⁹Vídeo bloque MotorDriverROS2 real: https://www.youtube.com/watch?v=MNaFWD9-ats&ab_channel=Tapii

Capítulo 5

Aplicación 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 (sólo rotación)

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.

En una primera aproximación, buscaremos sólo mantener a la persona centrada usando únicamente movimiento angular y, una vez que tengamos esta parte funcionando, añadiremos la parte del comportamiento que implica seguir a la persona también linealmente.

En primer lugar, debemos preparar el entorno de pruebas, por lo que aprovecharemos un modelo de persona teleoperada que creó Carlos Caminero¹, compañero de la carrera.

¹https://github.com/RoboticsLabURJC/2021-tfg-carlos-caminero/tree/main/amazon_hospital/hospital_world



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 3.4.1, ya que incluye tanto cámara como láser.

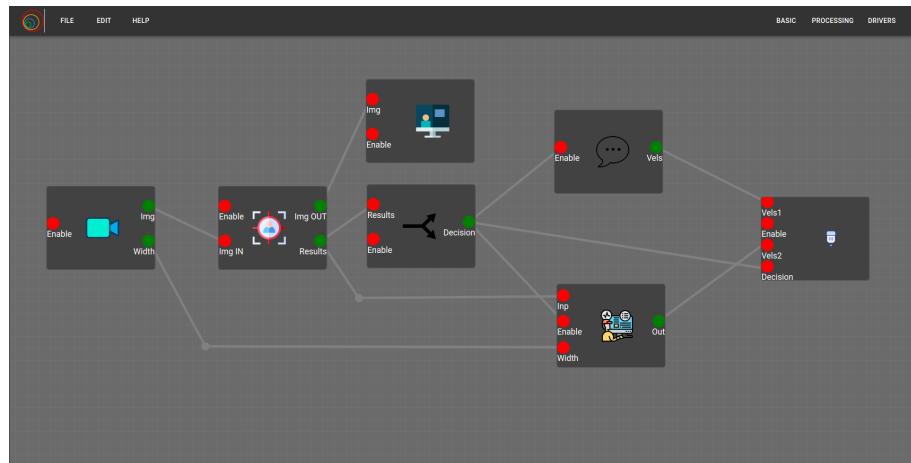


Figura 5.2: Circuito inicial del algoritmo sigue-persona.

La lógica que seguirá será la siguiente: recibir la imagen del *topic* de la cámara y compartir la 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 ??, que recibe tanto las distintas velocidades, como la decisión que se ha tomado, y envía al *topic* la adecuada.

La detección visual utiliza yolov3², un algoritmo de detección de objetos a tiempo real que permite identificarlos tanto en video como en imágenes usando redes neuronales (darknet³). 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.

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

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

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

sin una persona en la imágen, 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 PID⁴. 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.

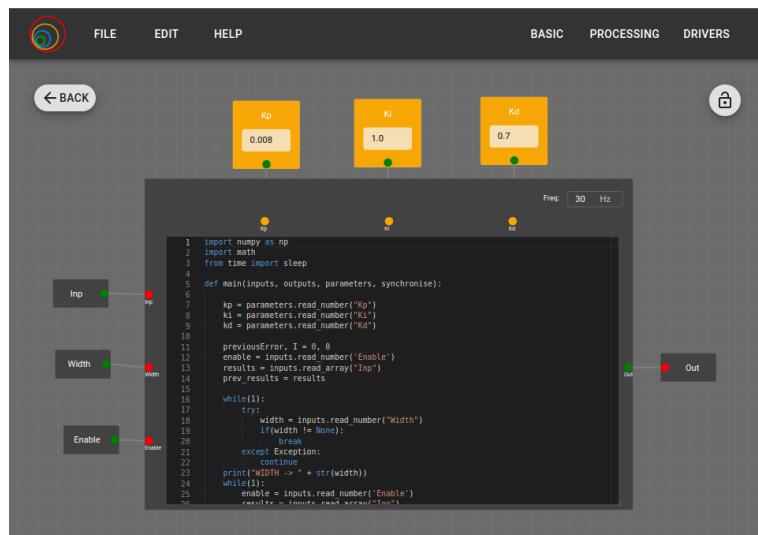


Figura 5.3: Circuito del bloque PID del sigue-persona en VisualCircuit.

Analizando el código del bloque PID (5.4), podemos ver que primero lee los tres parámetros del controlador y entramos en el mismo bucle que hemos mencionado en el código bloque de decisión (5.2), donde esperamos hasta obtener datos para empezar a trabajar. Después, en el bucle principal sólo se analizan y envían los datos del PID en

⁴**PID**: Controlador proporcional, integral y derivativo. Mecanismo de control que, mediante sistema en lazo cerrado (realimentación), permite regular un valor (velocidad, temperatura, presión, etc)

caso de que el bloque haya sido activado.

```

def main(inputs, outputs, parameters, synchronise):
    kp = parameters.read_number("Kp")
    ki = parameters.read_number("Ki")
    kd = parameters.read_number("Kd")
    previousError = 0
    I = 0
    max_rotation = 1
    enable = inputs.read_number('Enable')
    results = inputs.read_array("Inp")
    prev_results = results

    #Wait for values
    while(1):
        try:
            width = inputs.read_number("Width")
            if(width != None):
                break
        except Exception:
            continue
    while(1):
        enable = inputs.read_number('Enable')
        results = inputs.read_array("Inp")
        if(enable != 0):
            try:
                if(enable == 1):
                    error = float(results[0]+results[2]/2) - width/2
                    prev_results = results
                P = error
                D = float(error) - float(previousError)
                PIDvalue = (kp*P) + (kd*D)
                previousError = float(error)

                angular_velocity = -PIDvalue
                if(angular_velocity > max_rotation or angular_velocity
                   < -max_rotation):
                    angular_velocity =
                        max_rotation*angular_velocity/abs(angular_velocity)
                data = [0,0,0,0,0, angular_velocity]
                outputs.share_array("Out", data)

                synchronise()
            except Exception:
                synchronise()
                continue

```

Código 5.4: Código del bloque del PID sigue-persona.

Como podemos ver en el código anterior, el controlador usado finalmente es únicamente PD (sin parte integral). La parte proporcional se consigue mediante la resta del resultado actual y el resultado objetivo, en nuestro caso se trata del centro de la imagen, por lo que usamos la mitad del ancho de la imagen. Para la parte derivativa, se busca reducir los cambios bruscos, por lo que se resta el error actual con el error de la iteración anterior.

Una vez que tenemos la velocidad calculada, se la mandamos al bloque de los motores. Este bloque es igual que el que creamos en el apartado ?? pero modificado para poder tener cuatro *inputs*: *Enable*, *vel1* (rotación), *vel2* (PID) y decisión. En el código del bloque también se ha modificado la función *main* para que lea las velocidades correspondientes al comportamiento actual y que esta sea la velocidad que se comanda a los motores del robot.

```
while auto_enable:
    try:
        decision = inputs.read_number("Decision")
        if(decision == 0):
            velocities = inputs.read_array('Vels1')
        else:
            velocities = inputs.read_array('Vels2')
    except Exception:
        continue
```

Código 5.5: Código del bloque del *MotorDriver* sigue-persona.

Al probarlo todo junto usando el TurtleBot2 real, podemos observar que mientras la persona está quieta, la cámara la mantiene centrada y, cuando empieza a moverse hacia un lado, el robot gira para volver a ponerlo en el centro de la imagen.

En la siguiente secuencia de imágenes se puede observar el movimiento que ha seguido el robot tras la ejecución:

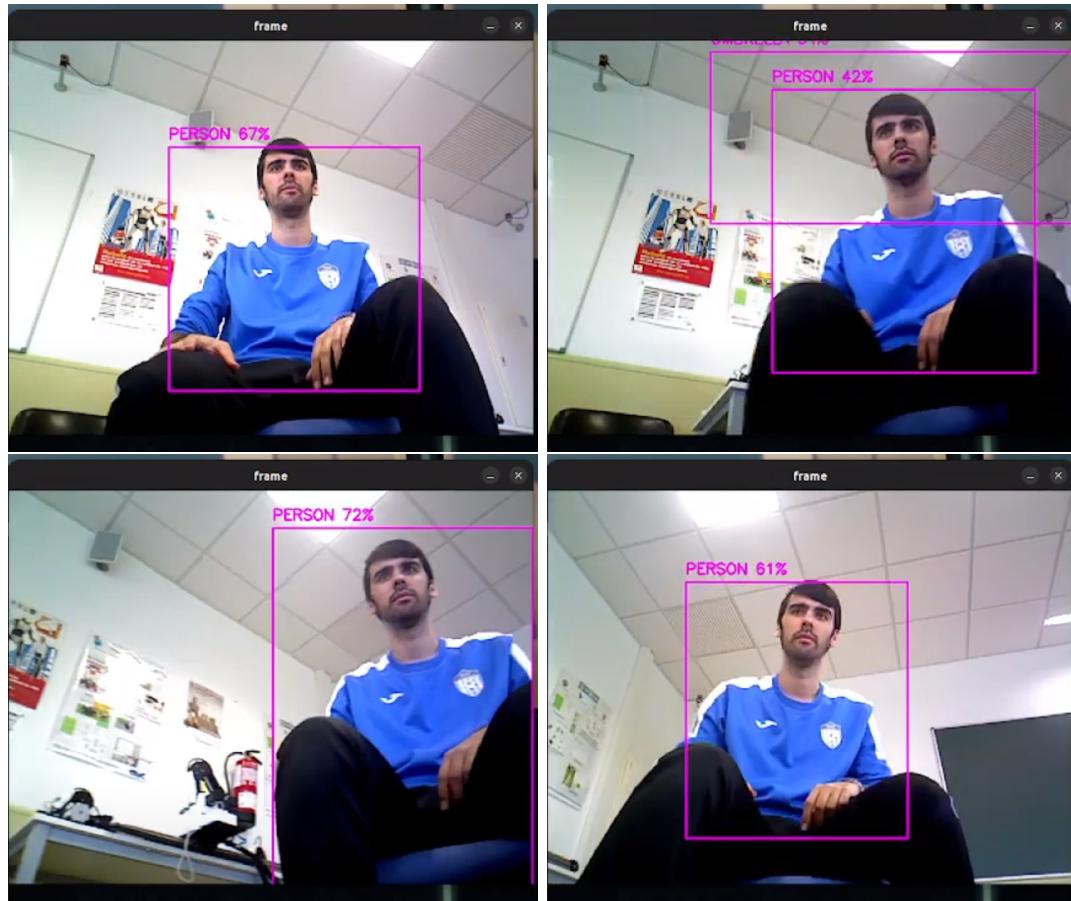


Figura 5.4: Secuencia de imágenes del sigue-personas. Imagenes obtenidas de Youtube⁵.

5.2. Modificaciones al sigue-personas para incluir movimiento lineal.

Para añadir el comportamiento de seguir linealmente a la persona, debemos leer también la información sobre la profundidad que nos da la cámara. Para ello vamos a crear un bloque usando el modelo de bloques de sensores que usamos anteriormente (??). También cambiaremos el funcionamiento general de varios bloques para optimizar y reducir el número de *inputs* y bloques del circuito.

⁵Vídeo: https://www.youtube.com/watch?v=Uir_iqM0plc&ab_channel=Tapii

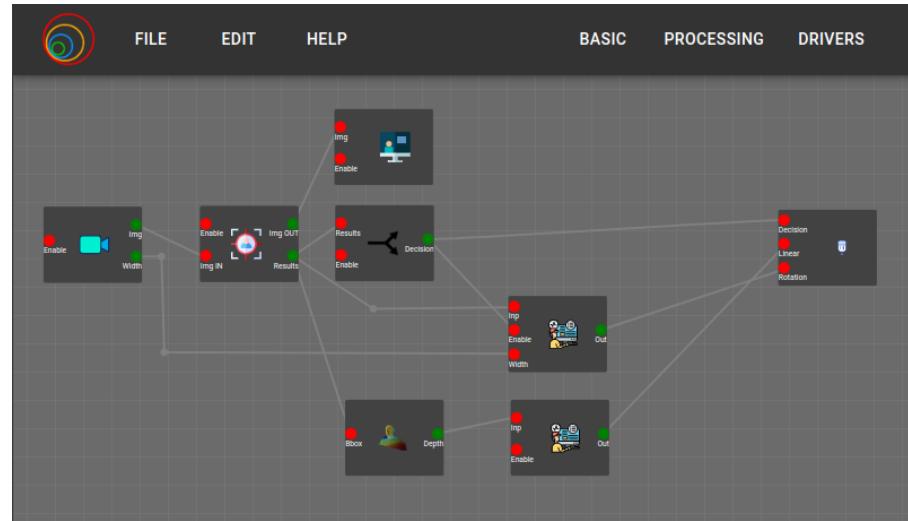


Figura 5.5: Circuito inicial del algoritmo sigue-persona.

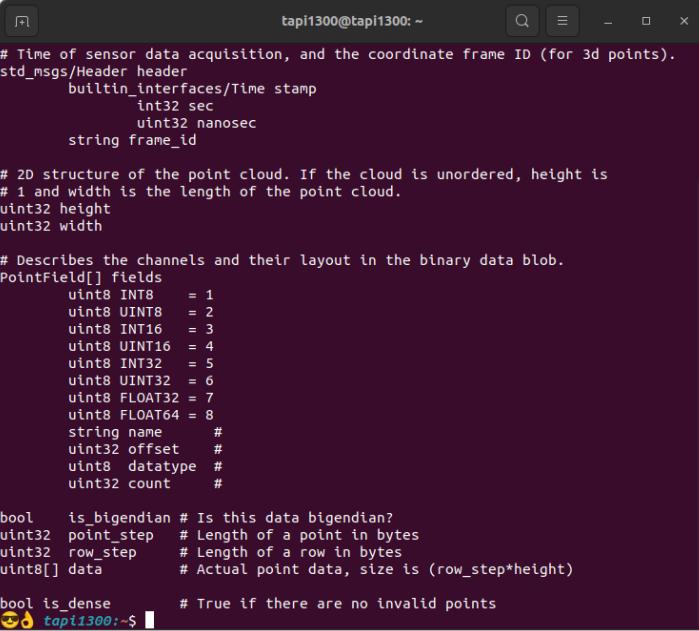
Como podemos ver, toda la rama inferior de bloques es la que se encarga del movimiento lineal, mientras que la superior se encarga del movimiento angular. La única parte de la parte superior que ha cambiado es que ya no existe un bloque que envíe siempre la velocidad de rotación, en cambio está directamente en el bloque que envía la velocidad a los motores y, dependiendo de la decisión, se envía la rotación estática o se envían las velocidades de los bloques PID.

```

while auto_enable:
    try:
        decision = inputs.read_number("Decision")
        if(decision == 0):
            velocities = [0,0,0,0,0,0.05]
        else:
            velocities = inputs.read_array('Vels2')
            velocities[0] = inputs.read_number('Linear')
    except Exception:
        continue

```

Código 5.6: Código del bloque del *MotorDriver* sigue-persona modificado.



```

# Time of sensor data acquisition, and the coordinate frame ID (for 3d points).
std_msgs/Header header
    builtin_interfaces/Time stamp
        int32 sec
        uint32 nanosec
    string frame_id

# 2D structure of the point cloud. If the cloud is unordered, height is
# 1 and width is the length of the point cloud.
uint32 height
uint32 width

# Describes the channels and their layout in the binary data blob.
PointField[] fields
    uint8 INT8      = 1
    uint8 UINT8     = 2
    uint8 INT16     = 3
    uint8 UINT16    = 4
    uint8 INT32     = 5
    uint8 UINT32    = 6
    uint8 FLOAT32   = 7
    uint8 FLOAT64   = 8
    string name     =
    uint32 offset   =
    uint8 datatype  =
    uint32 count    =

bool is_bigendian # Is this data big endian?
uint32 point_step # Length of a point in bytes
uint32 row_step   # Length of a row in bytes
uint8[] data      # Actual point data, size is (row_step*height)

bool is_dense      # True if there are no invalid points

```

Figura 5.6: Estructura del tipo de mensaje *sensor-msgs/msg/PointCloud2*.

En cuanto a la rama inferior, el bloque principal es el que recibe la información de la profundidad. Esta información viene en forma de *PointCloud2* (*sensor-msgs/msg/PointCloud2*) que, como podemos ver en la imagen 5.6, envía los datos del sensor en el campo *data*, por lo que esto será lo que guardemos en la variable global del bloque del sensor y será lo que enviemos por el cable.

La forma en la que se guardan los datos dentro de este campo es en bytes y es la siguiente para cada punto de la cámara ($640 \times 480 = 307200$ puntos): 12 bytes para x,y,z (4 bytes cada coordenada), 4 bytes vacíos, 4 bytes para el color del punto y otros 12 bytes vacíos. Esto hace que en el *array* de datos nos aparezcan 9830400 valores y la búsqueda final sea el número de filas (coordenada Y) por el ancho total de la imagen más la posición actual de nuestra coordenada X multiplicado por 32 (posiciones que ocupa cada pixel en el array) y sumamos 8 para obtener la posición inicial de la coordenada Z. Luego, usando la función “*unpack*” del paquete *struct* podemos transformar los siguientes 4 bytes (correspondientes a la coordenada Z) en un *float* y obtener la distancia entre el robot y el centro de la *BoundingBox* de la persona.

```

# IMPORT
from sensor_msgs.msg import PointCloud2
from struct import unpack

# CALLBACK DENTRO DE LA CLASE
def callback(self, msg):
    global measure
    measure = msg.data

# BUCLE DENTRO DEL MAIN
while(1):
    bbox = inputs.read_array("Bbox")
    if bbox is None:
        continue
    try:
        x = int(bbox[0]+bbox[2]/2)
        y = int(bbox[1]+bbox[3]/2)
    except:
        continue

    rclpy.spin_once(depth_subscriber)
    point = (width*y+x)*32+8
    depth = unpack('f', measure[point:point+4])
    outputs.share_array("Depth",depth)
    synchronise()

```

Código 5.7: Código del bloque del *PointCloud2* del sigue-persona.

Esta información llega a un segundo bloque PID, que es el que se va a encargar de mantener esta distancia con la persona en 1.5 metros. El código de este bloque es similar al del otro bloque PID (5.4), pero ahora no necesitamos una velocidad límite (el mínimo y máximo que conseguiremos no es peligroso en comparación a las velocidades angulares altas).

```

import numpy as np
import math
from time import sleep

def main(inputs, outputs, parameters, synchronise):
    auto_enable = True
    try:
        enable = inputs.read_number("Enable")
    except Exception:
        auto_enable = True
    kp = parameters.read_number("Kp")
    ki = parameters.read_number("Ki")
    kd = parameters.read_number("Kd")
    previousError, I = 0, 0
    while(auto_enable or inputs.read_number('Enable')):
        msg = inputs.read_number("Inp")
        if msg is None:
            continue
        error = float(msg) - 1.5
        sleep(0.01)

        P = error
        D = error - previousError
        PIDvalue = (kp*P) + (kd*D)
        previousError = error

        linear_velocity = PIDvalue
        if msg == 0:
            linear_velocity = 0
        outputs.share_number("Out", linear_velocity)
        synchronise()

```

Código 5.8: Código del bloque del PID de velocidad lineal del sigue-persona.

5.3. Pruebas en el robot real y resultado final.

Ahora que ya está configurado el comportamiento correctamente, tanto la velocidad lineal para mantenerse a 1.5 metros como la velocidad angular para mantener al humano en el centro de la visión del robot, es hora de probarlo en el robot real.

Para configurar correctamente los sensores del robot real debemos tener instalados los paquetes referentes a lanzar el kobuki (3.3.1), al igual que los necesarios para usar la cámara (3.3.3). De aquí usaremos los siguientes comandos para conseguir que todo funcione correctamente:

```
$> ros2 launch asus_xtion asus_xtion.launch.py
```

```
$> ros2 launch ir_kobuki kobuki_rplidar.launch.py
```

Código 5.9: Comandos para activar la cámara con ROS2 y lanzar el kobuki con el láser.

Una vez probado, podemos comprobar los resultados mirando el vídeo en el que se explica el funcionamiento de los bloques (como ya se ha explicado en las secciones 5.1 y 5.2) y se muestra un ejemplo de ejecución con el robot real en los laboratorios:

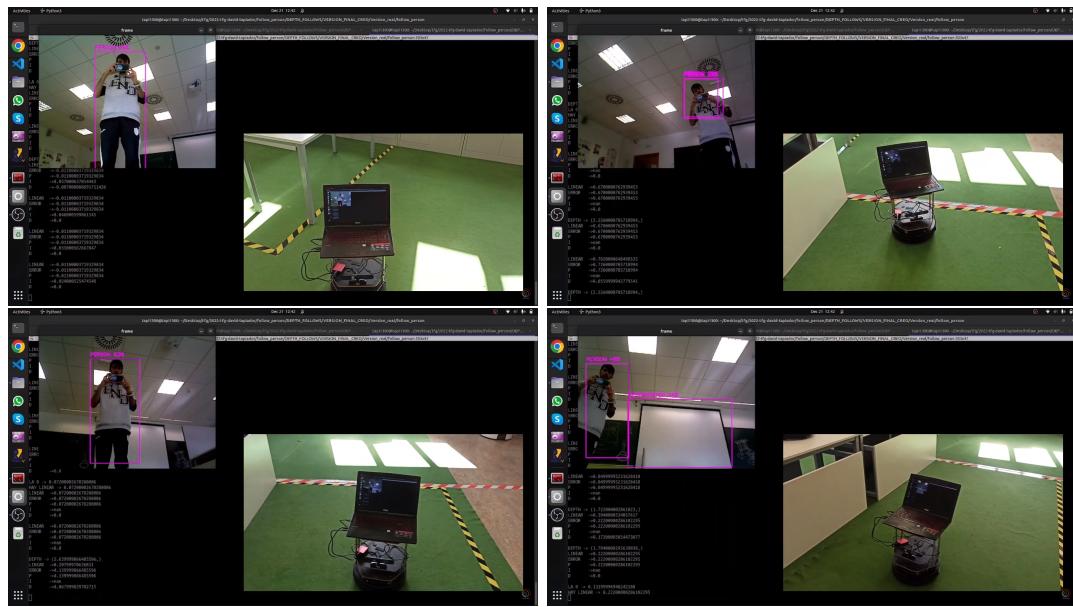


Figura 5.7: Secuencia de imágenes del sigue-personas. Imagenes obtenidas de Youtube⁶.

⁶Vídeo: https://www.youtube.com/watch?v=IknpvAs_jAo&ab_channel=JdeRobot

Capítulo 6

Aplicación Visual Field Force

Como ya se ha explicado, otra de las aplicaciones que se va a desarrollar consiste en navegación usando el algoritmo VFF mediante el uso de máquinas de estados.

Para desarrollar esta aplicación, el proceso se dividirá en dos partes: desarrollo del algoritmo VFF e implementación de dicho comportamiento en una máquina de estados que genere ubicaciones aleatorias.

6.1. Campo de fuerzas virtuales (VFF)

El algoritmo de movimiento mediante campo de fuerzas virtuales o *virtual field force* consiste en permitir el movimiento a través de un lugar avanzando gracias a objetivos temporales (como si fueran balizas) y esquivando los obstáculos entre la posición del robot y el objetivo mediante la fuerza repulsiva generada por las medidas de los distintos sensores al percibir dichos obstáculos.

Este algoritmo se basa en dos partes principales: la fuerza repulsiva (inversamente proporcional a la distancia con los obstáculos) y la fuerza atractiva (dirección al objetivo). Para obtener ambas fuerzas serán necesarios dos sensores: láser (fuerza repulsiva) y odometría (fuerza atractiva).

6.1.1. Diseño del circuito y escenario

El circuito para este comportamiento consistirá de dos ramas principales, una para cada fuerza, y la unión de estas ramas mandando una velocidad final tanto lineal como angular.

En la rama superior esta el sensor láser junto con un bloque para obtener una única medida como resultado de todas las medidas del láser. En la inferior se encuentra el sensor *odom*, que da la posición del robot en las coordenadas del mundo simulado. Esta

medida se pasa a dos bloques: el generador de objetivos (bloque que envía el objetivo actual y, en caso de haber llegado, envía el siguiente dentro de una lista) y el bloque que se encarga de calcular la fuerza atractiva.

Ambas ramas se juntan en un bloque que las suma teniendo en cuenta sus valores de influencia (la repulsiva debe influir más que la atractiva para evitar colisiones por roce) y se envían como velocidades al bloque MotorDriverROS2 (4.2).

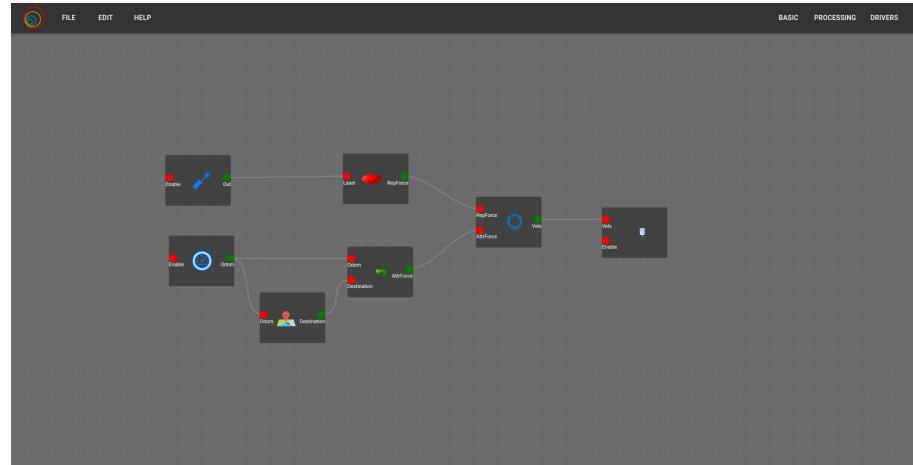


Figura 6.1: Circuito del algoritmo VFF.

El escenario de pruebas consistirá en un circuito creado a base de bloques cúbicos como muros con algunas barras rojas (cubos rectangulares) horizontales que marcan aproximadamente dónde se encuentran los objetivos. Hay una zona en la que no hay prácticamente borde para comprobar que la fuerza atractiva fuese lo suficiente fuerte como para no desviarse al encontrar un hueco en uno de los lados.

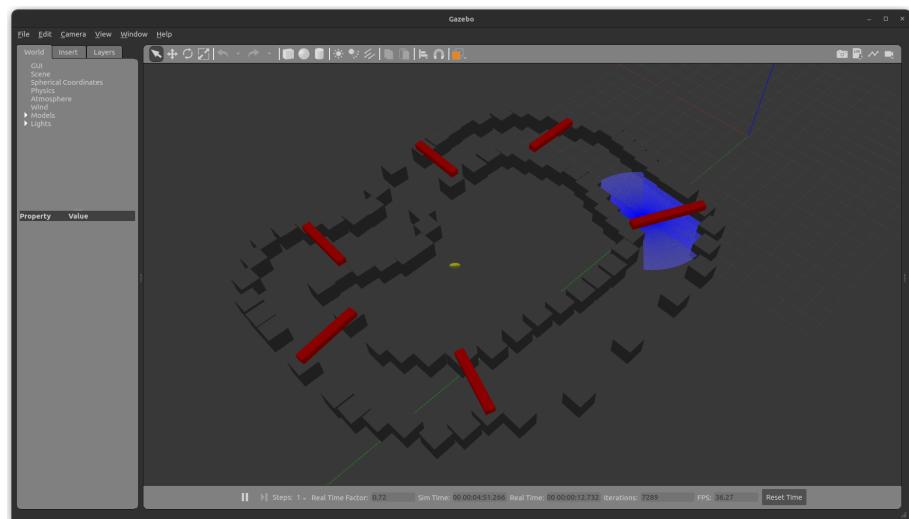


Figura 6.2: Mundo gazebo para probar el algoritmo VFF.

6.1.2. Bloques específicos

El primer bloque nuevo específico de esta aplicación es el que transforma las medidas del láser a un valor único. Para que la información del láser sea más fácilmente manipulable, se ha creado una función `parse_laser_data()` que almacena cada medida del láser en un array de tuplas junto al ángulo que dicha medida representa.

Después se recorre el array de tuplas para obtener vectores que representen el valor del eje X e Y para cada medida del láser. Ésto se hace multiplicando la medida original (hipotenusa) por el seno o coseno del ángulo. Después se guarda en un nuevo array de tuplas para cada par de valores (x,y).

Ahora para calcular la fuerza final se recorre el array de medidas vectoriales sumando los valores inversos, es decir, el valor absoluto de la división de un valor entre la medida, permitiendo que cuanto más cercano (menor sea la medida) mayor influencia tenga en el resultado de fuerza repulsiva final. En el caso del eje X, sólo hay que comprobar que la medida sea distinta de 0, mientras que en el caso del eje Y hay que evitar medidas superiores a 10, ya que es el límite del sensor y sumaría `inf` (infinito).

```

def parse_laser_data (laser_data):
    laser = []
    for i in range(len(laser_data)):
        dist = laser_data[i]
        angle = math.radians (i)
        laser += [(dist, angle)]
    return laser

def getObs_xy(laser):
    laser2 = parse_laser_data(laser)
    laser_vectorized = []
    for d, a in laser2:
        if(a == 0):
            x = 10
            y = 10
        else:
            x = d * math.cos (a) * -1
            y = d * math.sin (a) * -1
        v = (x, y)
        laser_vectorized += [v]
```

```

obsx = 0
obsy = 0
amortiguacion = 1 #Mayor amortiguacion, mas valen los valores lejanos
pico = 1           #Mayor pico, mas valen los valores cercanos a cero
for i in range(int(len(laser_vectorized)/2)):
    if(laser_vectorized[i][0] != 0):
        obsx -=
            pico*abs(math.atan(amortiguacion/(laser_vectorized[i][0])))
    if(i<90):
        if(laser_vectorized[i][1] != 0 and laser_vectorized[i][1] <
           10):
            obsy -=
                pico*abs(math.atan(amortiguacion/(laser_vectorized[i][1])))
    else:
        if(laser_vectorized[180+i][1] != 0 and
           laser_vectorized[180+i][1] < 10):
            obsy +=
                pico*abs(math.atan(amortiguacion/(laser_vectorized[180+i][1])))
return obsx, obsy

def main(inputs, outputs, parameters, synchronise):
    reduction = 1/50
    try:
        while 1:
            measures = inputs.read_array("Laser")
            if measures is not None:
                obsX, obsY = getObs_xy(measures)
                outputs.share_array("RepForce", [obsX/50, obsY/50])

```

Código 6.1: Funciones para obtener la fuerza repulsiva.

En cuanto a la fuerza atractiva, se debe usar la posición actual del robot y la posición que se ha marcado como destino.

Para ello se usará la plantilla para bloques sensores que se planteó en el capítulo 4 (4.2 y 4.3), modificando el tipo de mensaje, ya que el *topic /odom* tiene el tipo de mensaje *nav_msgs/msg/Odometry*, cuyos parámetros podemos comprobar mediante el comando “`ros2 interface show nav_msgs/msg/Odometry`” con el siguiente resultado:

```
tapi1300@tapi1300: ~
tapi1300@tapi1300: ~ 109x42
tapi1300@tapi1300:~$ ros2 interface show nav_msgs/msg/Odometry
# This represents an estimate of a position and velocity in free space.
# The pose in this message should be specified in the coordinate frame given by header.frame_id
# The twist in this message should be specified in the coordinate frame given by the child_frame_id

# Includes the frame id of the pose parent.
std_msgs/Header header
    builtin_interfaces/Time stamp
        int32 sec
        uint32 nanosec
    string frame_id

# Frame id the pose points to. The twist is in this coordinate frame.
string child_frame_id

# Estimated pose that is typically relative to a fixed world frame.
geometry_msgs/PoseWithCovariance pose
    Pose pose
        Point position
            float64 x
            float64 y
            float64 z
        Quaternion orientation
            float64 x_0
            float64 y_0
            float64 z_0
            float64 w_1
        float64[36] covariance

# Estimated linear and angular velocity relative to child_frame_id.
geometry_msgs/TwistWithCovariance twist
    Twist twist
        Vector3 linear
            float64 x
            float64 y
            float64 z
        Vector3 angular
            float64 x
            float64 y
            float64 z
        float64[36] covariance
tapi1300@tapi1300:~$
```

Figura 6.3: Estructura del tipo de mensaje *nav_msgs/msg/Odometry*.

Los datos que se necesitan para la aplicación son la posición *X* e *Y*, al igual que la posición angular del eje *Z*, por lo que esto será lo que se guarde en la función *callback* y lo que se envíe por el cable de salida usando un array formado por estos tres valores numéricos. Los primeros dos valores se pueden sacar directamente del campo *pose.pose.position* del mensaje, pero la rotación viene dada mediante cuaterniones, por lo que hay que transformarlo a ángulos de euler y obtener el valor que se busca. Esto se realiza mediante la función *Rotation* del paquete *scipy.spatial.transform*¹ de *python3*.

```
def callback(self, msg):
    global odom
    odom[0] = msg.pose.pose.position.x
    odom[1] = msg.pose.pose.position.y
    rot = Rotation.from_quat([msg.pose.pose.orientation.x,
                             msg.pose.pose.orientation.y, msg.pose.pose.orientation.z,
                             msg.pose.pose.orientation.w])
    odom[2] = rot.as_euler('xyz', degrees=True)[2]
```

Código 6.2: Funciones para obtener la fuerza repulsiva.

El siguiente bloque es el generador de destinos, que usa la ubicación del robot

¹Librería Spicy:
docs.scipy.org/doc/scipy/reference/generated/scipy.spatial.transform.Rotation.html

para comprobar si ha llegado a la actual (teniendo en cuenta un margen) y mandar la siguiente dentro de la lista de destinos. También se ha implementado un contador de vueltas, aprovechando que se ha creado un circuito cerrado.

```

def main(inputs, outputs, parameters, synchronise):
    dest_arr = [[7,9],
                [10,15],
                [10,23],
                [5,26],
                [-1,23],
                [-1,9]]
    destination = [0,0,0]
    odom = []
    first = True
    margen = 1
    actual = -1
    lap = 0
    while 1:
        odom = inputs.read_array("Odom")
        if odom is not None:
            if(first or
               (odom[0] > dest_arr[actual][0]-margen and
                odom[0] < dest_arr[actual][0]+margen and
                odom[1] > dest_arr[actual][1]-margen and
                odom[1] < dest_arr[actual][1]+margen)):
                actual += 1
            if(first or actual > len(dest_arr)-1):
                lap +=1
                actual = 0
                print("LAP NUMBER " + str(lap))
            destination = dest_arr[actual]
            outputs.share_array("Destination", destination)
            print("NEW DESTINATION! " + str(destination))
            first = False

```

Código 6.3: Bloque generador de ubicaciones.

Estos dos *arrays* (*odom* y destino) los recibe el siguiente bloque, que es el que calcula la fuerza atractiva. Este bloque comprueba que los datos que recibe no estén vacíos (al iniciar la ejecución puede leer *null* de los cables), calculando la posición relativa del objetivo respecto al robot tanto para los ejes *X* e *Y* como para la rotación relativa. En caso de que la rotación relativa supere un valor máximo, se establece dicho máximo como valor de ángulo relativo, enviando éste como único valor, ya que en esta versión sólo se tiene en cuenta la velocidad angular, manteniendo la lineal constante a 1 (en el VFF con máquina de estados sí se tiene en cuenta la velocidad lineal).

```

max_y_rel = 2
def main(inputs, outputs, parameters, synchronise):
    while True:
        x_y_Yaw = inputs.read_array("Odom")
        dest = inputs.read_array("Destination")
        if x_y_Yaw is not None and dest is not None:
            dx = dest[0] - x_y_Yaw[0]
            dy = dest[1] - x_y_Yaw[1]
            # Rotate with current angle
            y_rel = dx * math.sin (math.radians(-x_y_Yaw[2])) + dy *
                math.cos (math.radians(-x_y_Yaw[2]))
            if(y_rel > max_y_rel):
                y_rel = max_y_rel
            elif(y_rel < -max_y_rel):
                y_rel = -max_y_rel
            outputs.share_number("AttrForce", y_rel)

```

Código 6.4: Bloque que calcula la fuerza atractiva.

Por último, ambas fuerzas se combinan en un mismo bloque, que es el encargado de sumar ambas fuerzas y transformarlas en velocidad angular. Para ello aplica un valor a modo de controlador constante para que la fuerza repulsiva sea más significativa que la atractiva y permitir que el robot esquive objetos que puedan quedar muy cercanos. Finalmente se envía el *array* de velocidades manteniendo la velocidad lineal a 1 y siendo la velocidad angular el valor calculado por el algoritmo.

```

import math

def main(inputs, outputs, parameters, synchronise):
    maximo = 3
    while True:
        rep = inputs.read_number("RepForce")
        attr = inputs.read_number("AttrForce")
        if rep is not None and attr is not None:
            final_w = 1.4*rep + 0.4*attr
            if(final_w > maximo):
                final_w = maximo
            elif(final_w < -maximo):
                final_w = -maximo
            outputs.share_array("Vels", [1,0,0,0,0,final_w])

```

Código 6.5: Bloque que transforma las fuerzas en velocidades.

6.1.3. Validación experimental

Para comprobar que el algoritmo funcionase, se han realizado varias pruebas: sólo fuerza repulsiva con obstáculos y sin ellos, y el algoritmo completo con un bloque de depuración (6.2.2).

Para las primeras pruebas el circuito contaría únicamente con 3 bloques: el bloque del sensor láser, el que calcula la fuerza repulsiva y el bloque MotorDriverROS2:

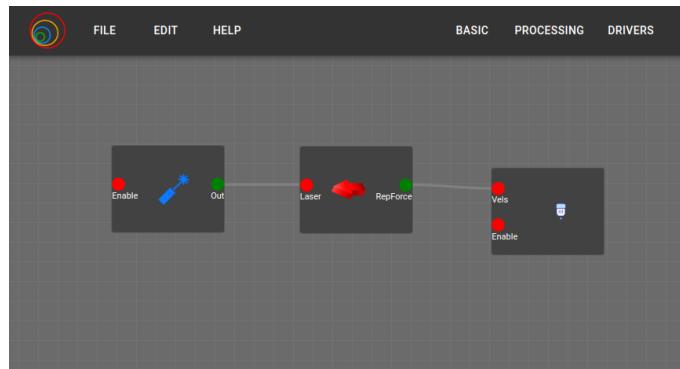


Figura 6.4: Circuito de VFF sólo con la fuerza repulsiva.

Como podemos comprobar en las siguientes secuencias y en sus vídeos correspondientes, la parte de la fuerza repulsiva cumple con el objetivo, ya que mantiene al robot alejado de las paredes del circuito a la vez que evita los obstáculos (en el caso en el que los hay).

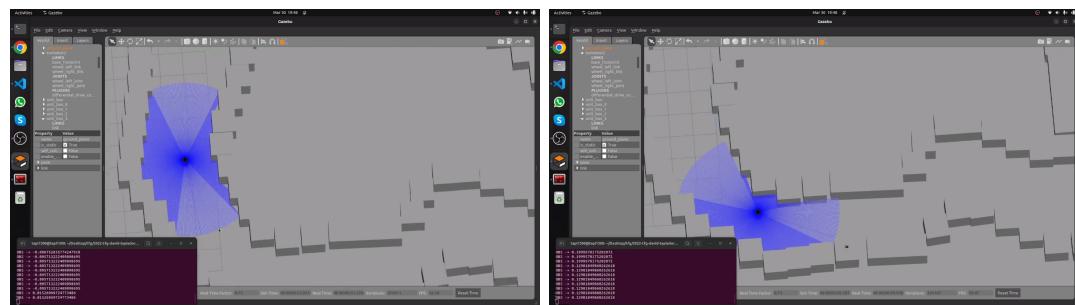


Figura 6.5: VFF usando sólo la fuerza repulsiva sin obstáculos. Imágenes obtenidas de Youtube².

²Vídeo VFF fuerza repulsiva sin obstáculos: https://www.youtube.com/watch?v=uhtBRw96Zl4&ab_channel=Tapii

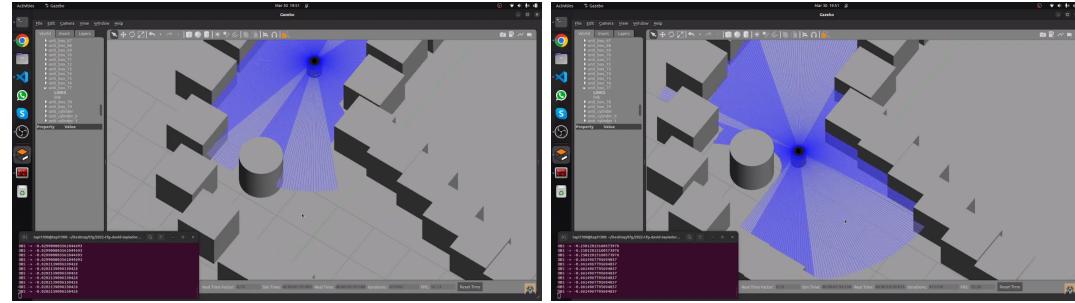


Figura 6.6: VFF usando sólo la fuerza repulsiva con obstáculos. Imagenes obtenidas de Youtube³.

En la siguiente prueba, el circuito usado incluye el bloque display (se describirá su funcionamiento en el siguiente punto, ya que aquí no estaba completamente desarrollado) y un bloque *screen* para mostrar la imagen de este bloque.

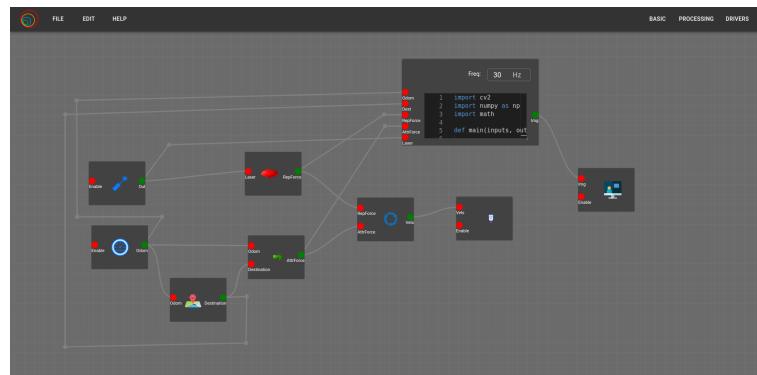


Figura 6.7: Circuito del algoritmo VFF.

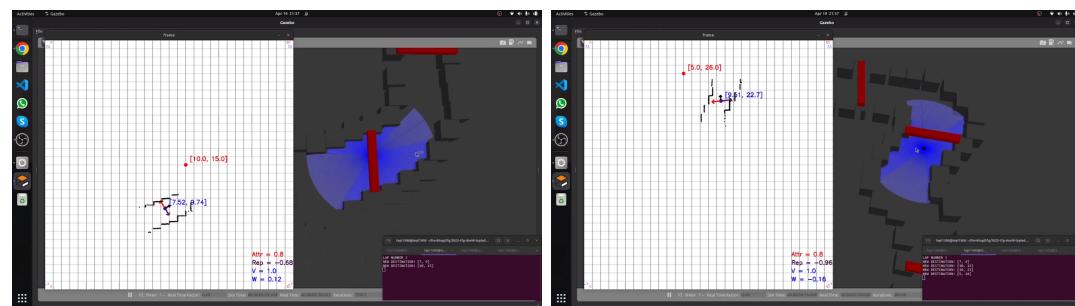


Figura 6.8: Secuencia de pruebas del algoritmo VFF. Imagenes obtenidas de Youtube⁴.

³Vídeo VFF fuerza repulsiva con obstáculos: https://www.youtube.com/watch?v=HeTFum_gTGw&ab_channel=Tapii

4 Vídeo VFF con display: https://www.youtube.com/watch?v=xdGCIRyFu7E&t=108s&ab_channel=Tapii

6.2. VFF mediante máquina de estados

Una vez que el algoritmo está prácticamente desarrollado, se va a implementar una máquina de estados. Esta máquina de estados tendrá 3 estados distintos: generar ubicación aleatoria, ir a la ubicación y volver al punto de salida (después de varias ubicaciones aleatorias).

Como se puede ver en el siguiente diagrama, la ejecución se inicia generando una ubicación aleatoria que se envía al estado VFF. Cuando se ha ido a 4 ubicaciones aleatorias, se pasa al estado *"Return Home"*, que envía la ubicación inicial del robot como nuevo destino para el estado VFF. Una vez que llegue al inicio, se detiene al robot.

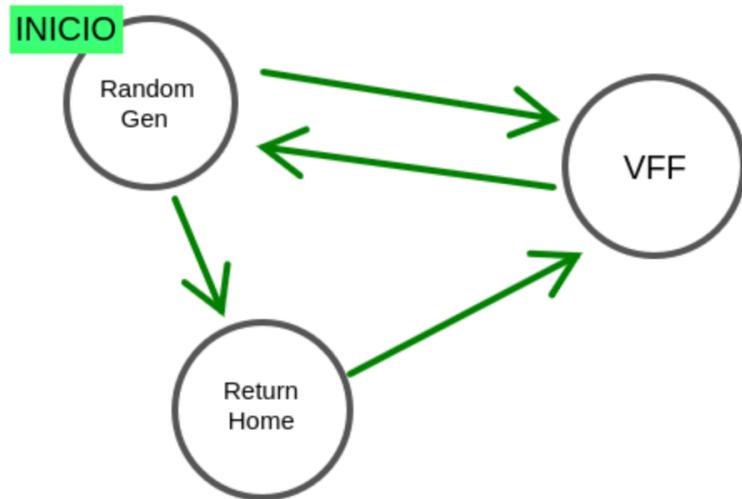


Figura 6.9: Diagrama de la máquinas de estados.

6.2.1. Diseño del circuito y escenario

Para implementar el comportamiento de máquina de estados, se han implementado dos bloques nuevos para los comportamientos nuevos. También se ha colocado el bloque de la odometría como un sensor general, ya que se usa en varios estados, mientras que el estado VFF está separado visualmente para que sea más sencillo de reconocer.

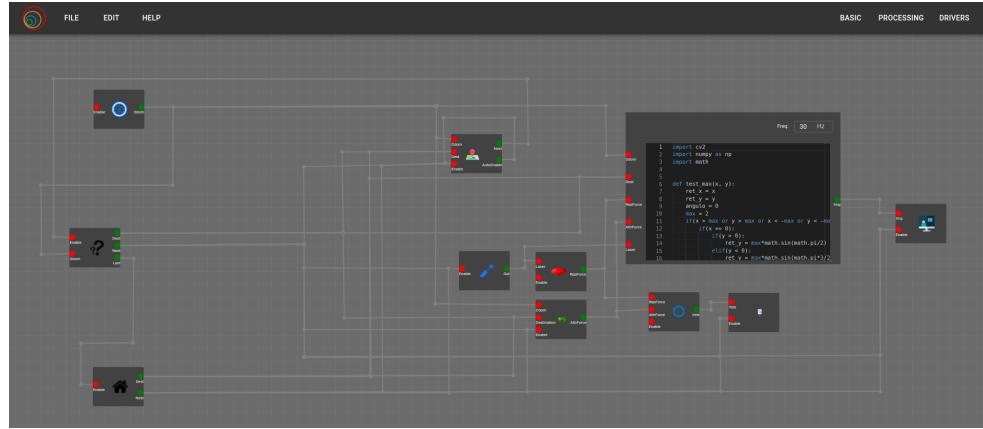


Figura 6.10: Circuito de VFF usando máquinas de estados.

Como los objetivos ahora son aleatorios, no se puede usar el circuito que teníamos en el apartado anterior, por lo que se ha creado un mundo en el que se han repartido varios cilindros por todo el mapa para que el robot tenga que moverse hasta el objetivo esquivándolos.

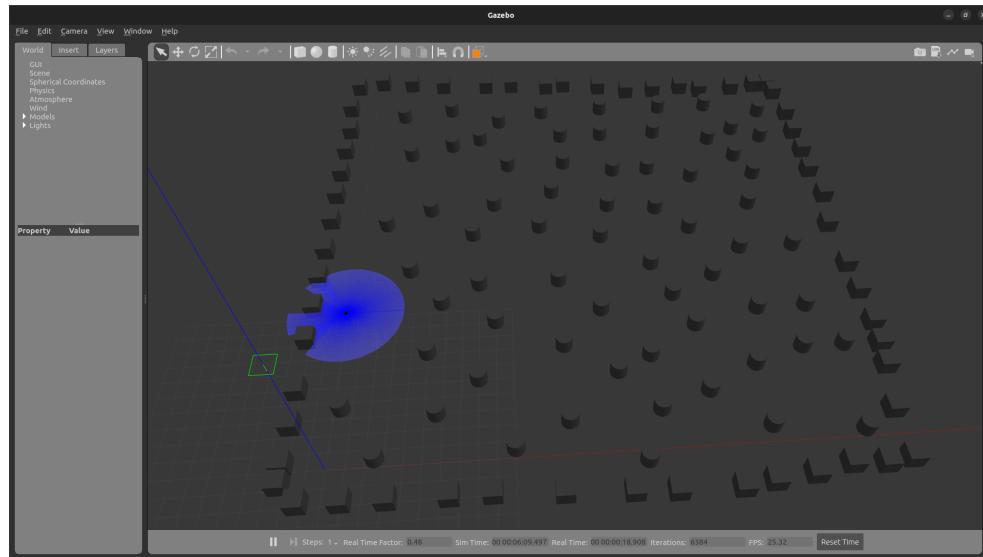


Figura 6.11: Mundo para probar el VFF con máquinas de estados.

6.2.2. Bloques específicos

Para ampliar el funcionamiento del VFF para que también tenga en cuenta las fuerzas para calcular la velocidad lineal, se ha tenido que modificar los bloques que calculan las fuerzas atractiva y repulsiva (6.1 y 6.4), por lo que en vez de mandar una única fuerza, se mandan la fuerza como vector (componente X e Y). También se han añadido lecturas al *input enable*, ya que es el que habilita activar y desactivar

los bloques, únicamente ejecutando los bloques que estén activos en ese momento y manteniendo el resto en estado inactivo.

```
# Cambio en bloque de fuerza repulsiva
outputs.share_array("RepForce", [obsX/50, obsY/50])

# Cambio en bloque de fuerza atractiva
outputs.share_array("AttrForce", [x_rel, y_rel])

# Bucle principal bloque laser
while 1:
    enable = inputs.read_number('Enable')
    if enable == 1:
        measure = None
        rclpy.spin_once(laser_subscriber)
        if measure is not None:
            outputs.share_array("Out", measure)
```

Uno de los bloques nuevos es el generador de ubicaciones aleatorias. Éste también es el que se encarga de decidir cuál será el siguiente estado. Para ello tiene un contador interno que le permite saber cuántas veces se han generado ubicaciones aleatorias y, cuando se llegue al valor de *max_times* se cambia al estado de volver al origen. Para generar la ubicación aleatoria, generamos un número que cumpla que num1 esté entre *x-márgen* y *x+márgen* y otro num2 que esté entre *y-márgen* y *y+márgen*, para evitar ubicaciones que estén demasiado alejadas del robot. También dicho número debe estar entre 0 y 30, ya que estos son los límites del mundo (máximo que se representa dentro del bloque *display*).

```
from random import randint
import time

def main(inputs, outputs, parameters, synchronise):
    first = True
    changed = False
    times = 0
    max_times = 4
    margen = 3
    x = [0,30]
    y = [0,30]
    while 1:
        enable = inputs.read_number("Enable")
        if(enable == 0):
            changed = False
```

```

if (enable == 1 or first) and not changed:
    odom = inputs.read_array("Odom")
    if odom[0] != None:
        changed = True
        first = False
        times += 1
        if(times < max_times):
            print("***ESTADO ACTIVADO -> GENERAR UBICACION
                  ALEATORIA***")
            dest = [randint(x[0]+1, x[1]-1),
                    randint(y[0]+1,y[1]-1)]

            while dest[0] > int(odom[0]-margen) and dest[0] <
                  int(odom[0]+margen):
                dest[0] = randint(x[0]+1, x[1]-1)
            while dest[1] > int(odom[1]-margen) and dest[1] <
                  int(odom[1]+margen):
                dest[1] = randint(y[0]+1,y[1]-1)

            print("NUEVO DESTINO -> " + str(dest))
            time.sleep(2)
            print("***ESTADO ACTIVADO -> VFF***")
            outputs.share_array("Dest", dest)
            outputs.share_number("Next", 1)
            outputs.share_number("Last", 0)
    else:
        print("***ESTADO ACTIVADO -> VUELTA AL ORIGEN***")
        outputs.share_number("Next", 0)
        outputs.share_number("Last", 1)

```

Código 6.6: Bloque que genera destinos aleatorios y decide el siguiente estado.

El siguiente bloque nuevo es el correspondiente al estado de volver al inicio. Cuando este bloque se activa, envía la ubicación del origen al estado VFF e imprime varias trazas para saber en qué punto del comportamiento se encuentra la ejecución.

```

import time
def main(inputs, outputs, parameters, synchronise):
    dest = [2,10]
    first = True
    going = False
    while 1:
        enable = inputs.read_number("Enable")
        if enable == 1 and first:
            print("UBICACION DEL ORIGEN -> " + str(dest))
            time.sleep(2)
            outputs.share_array("Dest", dest)
            outputs.share_number("Next", 1)
            first = False
        if enable == 0 and not first:
            going = True
        if enable == 1 and going:
            print("HEMOS LLEGADO AL ORIGEN!!!")
            outputs.share_number("Next", 0)

```

Código 6.7: Bloque para volver al inicio.

El bloque que transforma las fuerzas a velocidades también cambia, ya que ahora hay que tener en cuenta la componente lineal de las fuerzas y, por lo tanto, calcular nuevas proporciones y evaluar valores máximos y mínimos. También se ha añadido que, si la velocidad lineal es negativa y la angular es cercana a 0 (valor absoluto menor que 0.3), se ponga al robot a girar estático. Esto es para evitar objetivos que se encuentren detrás del robot y que, por lo tanto, intente llegar a ellos marcha atrás.

```

def main(inputs, outputs, parameters, synchronise):
    max_v = 2
    min_v = 0.5
    max_w = 3
    alpha_V = 1
    beta_V = 1
    alpha_W = 0.7
    beta_W = 1.4
    while 1:
        rep = inputs.read_array("RepForce")
        attr = inputs.read_array("AttrForce")
        if rep is not None and attr is not None:
            attr_x, attr_y = test_max(attr[0],attr[1])
            rep_x, rep_y = test_max(rep[0],rep[1])
            final_v = alpha_V*attr_x + beta_V*rep_x
            final_w = alpha_W*attr_y + beta_W*rep_y

```

```

        if(final_v < 0 and final_w < 0.3 and final_w > 0.3):
            # Rotar en el sitio hasta que no sea sentido opuesto
            final_v = 0
            final_w = max_w/3
        elif(final_v > max_v):
            final_v = max_v
        elif(final_v < min_v):
            final_v = min_v

        if(final_w > max_w):
            final_w = max_w
        elif(final_w < -max_w):
            final_w = -max_w

outputs.share_array("Vels", [final_v,0,0,0,0,final_w])

```

Código 6.8: Bloque que pasa de fuerzas a velocidades.

Por último, el bloque *display* consiste en crear una imagen mediante un *array* de *numpy* en el que se muestra aproximadamente una representación del mundo que ve el robot. Aquí se muestran tanto la ubicación del robot y del destino, como las fuerzas atractiva (verde), repulsiva (roja) y total (naranja), sus valores y las lecturas del láser mediante pequeños puntos. También se ha representado cada metro cuadrado del simulador mediante una cuadrícula en la imagen.

Aquí se usa la librería cv2 de opencv-python⁵ para editar la imagen, para insertar líneas (*cv2.line* para líneas y *cv2.arrowedLine* para flechas) y para escribir texto (*cv2.putText*), al igual que la librería *math* (*cos*, *sin*, *radians*, *sqrt*, ...) para calcular la orientación de las flechas de las distintas fuerzas.

Dada la extensión del código, no se va a incluir en esta memoria, por lo que aquí⁶ se puede acceder a él.

6.2.3. Validación experimental

En la siguiente secuencia de imágenes se puede comprobar el resultado de la ejecución del algoritmo usando la máquina de estados. Como se puede ver, el TurtleBot2 es capaz de evitar los obstáculos en tiempo real llegando a los distintos objetivos aleatorios que se han calculado durante esa ejecución.

⁵OpenCV-Python: <https://pypi.org/project/opencv-python/>

⁶Código bloque display: https://github.com/RoboticsLabURJC/2022-tfg-david-tapiador/blob/main/FSM/ZIPS/FSM_final/modules/Code_1.py

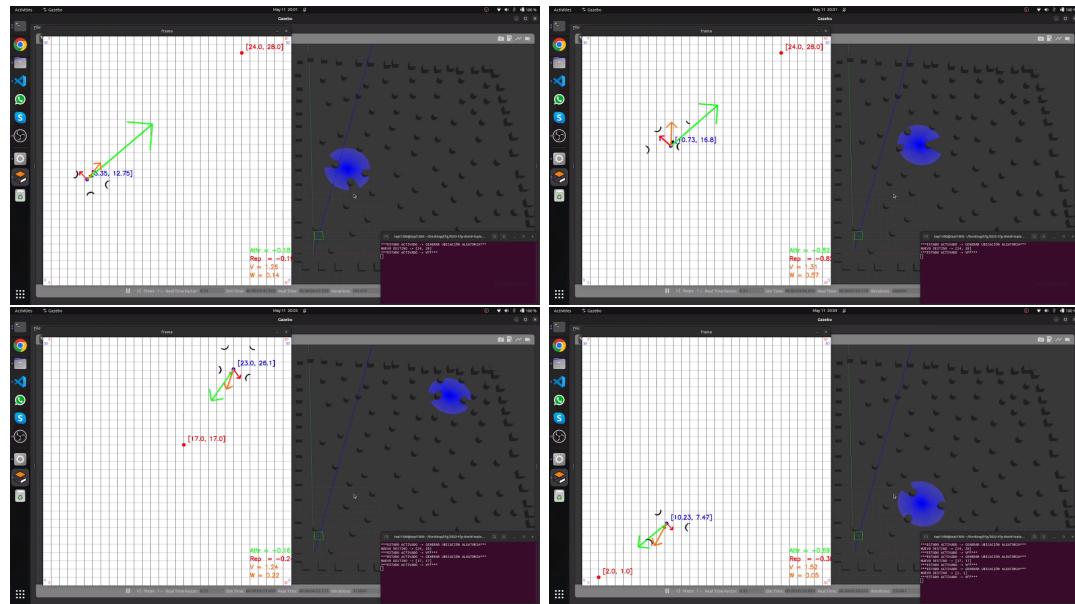


Figura 6.12: Secuencia de imágenes del algoritmo VFF usando FSM. Imagenes obtenidas de Youtube⁷.

⁷Vídeo: https://www.youtube.com/watch?v=EiBT8yqX29Q&ab_channel=Tapii

Capítulo 7

Conclusiones

Finalizamos esta memoria de Trabajo Fin de Grado con un resumen de las metas logradas por este trabajo al igual que por posibles líneas de trabajo para continuar.

7.1. Conclusiones

Bibliografía

- [Amazon.com, 2011] Amazon.com, I. (2011). Imagen de la cámara asusxtion. url: https://www.amazon.es/asus-90iw0122-b01ua-90iw0122-b01ua-webcam-negro/dp/b005uhb8ek/ref=cm_cr_arp_d_product_top?ie=utf8.
- [Castro, 2017] Castro, S. (2017). Comunicación entre nodos intermedios, hardware/software y nodo master. url: <https://blogs.mathworks.com/student-lounge/2017/11/08/matlab-simulink-ros/>.
- [Components, 2011] Components, R. (2011). Imagen del sensor láser rplidar a2. url: <https://www.roscomponents.com/es/lidar-escaner-laser/155-rplidar-a2m8-360-laser-scanner.html>.
- [elDiario.es, 2017] elDiario.es (2017). Imagen del robot shakey. url: https://www.eldiario.es/hojaderouter/tecnologia/shakey-robot-inteligencia-artificial-coche-autonomo_1_3466717.html.
- [Hostalia, 2017] Hostalia (2017). Imagen del ajedrecista de leonardo torres quevedo. url: <https://blog.hostalia.com/doctor-hosting/leonardo-torres-quevedo-1912-primer-juego-ordenador-ajedrecista/>.
- [Open Source Robotics Foundation,] Open Source Robotics Foundation, I. Imagen de un turtlebot2. url: <https://www.turtlebot.com/turtlebot2/>.
- [Robosavvy, 2022] Robosavvy (2022). Base kobuki para turtlebot2. url: <https://robosavvy.co.uk/kobuki-ymr-k01-w1-turtlebot-2-base.html>.
- [Robotics24/7, 2022] Robotics24/7 (2022). Imagen de los robots nao y pepper. url: https://www.robotics247.com/article/united_robots_group_acquires_softbank_-robotics_europe_service_robot_portfolio.
- [RobotsInAction, 2019] RobotsInAction (2019). Imagen del robot industrial unimate. url: <https://robotsinaction.com/el-primer-robot-industrial-unimate/>.