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

La robótica ha experimentado un crecimiento constante en los últimos años, consolidándose como un sector en alza. Esto ha llevado a un aumento significativo en las posibilidades y aplicaciones de la robótica, que cada vez se vuelven más complejas y desafiantes.

La correcta implementación de los distintos algoritmos es fundamental para su puesta en marcha, por ello en la última década se ha puesto mucho interés en la creación y maduración de simuladores y depuradores que ayuden en esta tarea. La herramienta VisualCircuit busca facilitar el desarrollo del propio algoritmo mediante programación visual usando bloques prefabricados o propios.

El objetivo del siguiente Trabajo Fin de Grado es extender la funcionalidad de esta herramienta para permitir la creación de algoritmos y aplicaciones que usen el *middleware* robótico ROS2. Para ello se han desarrollado varios bloques que han sido añadidos a las librerías de bloques, probándolos mediante aplicaciones robóticas simples y complejas, como seguir a una persona mediante reconocimiento visual.

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*

TFG *Trabajo Fin de Grado*

URDF *Unified Robotics Description Format*

USB *Universal Serial Bus*

VFF *Virtual Force Field*

Índice general

Índice de figuras

Listado de códigos

Capítulo 1
Introducción

La robótica es la disciplina 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 un robot 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 (figura 1.1) 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.

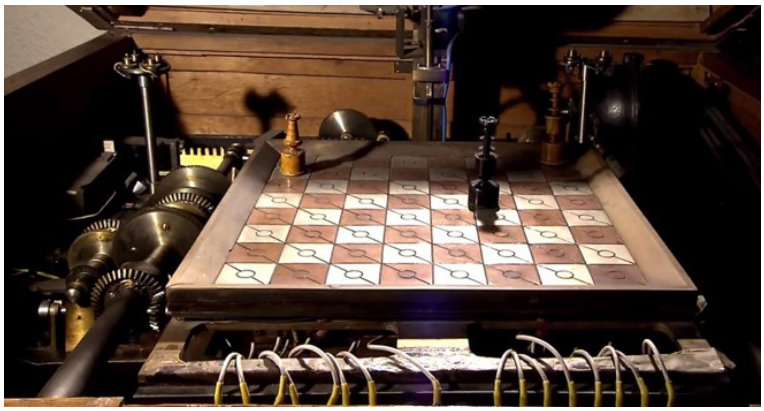


Figura 1.1: El Ajedrecista. Imagen obtenida de [Hostalia, 2017]

En la segunda mitad del siglo XX, con el gran avanze 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 (figura 1.2) , 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.

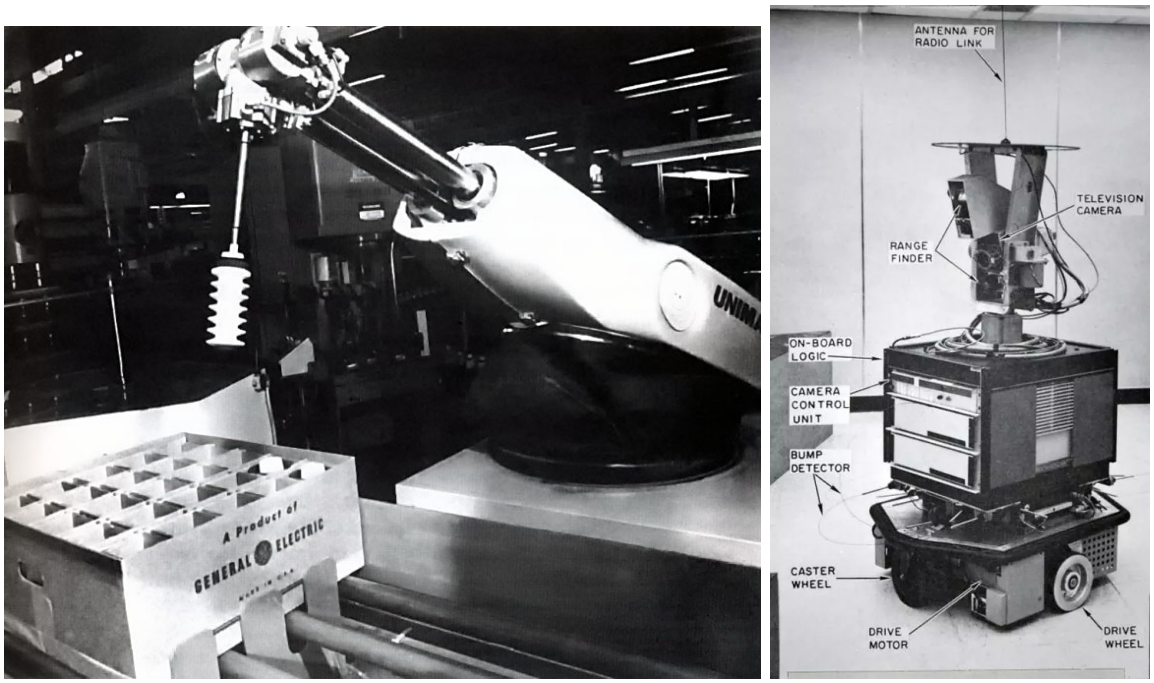


Figura 1.2: Robots Unimate y Shakey. Imágenes obtenidas respectivamente de [RobotsInAction, 2019] y [elDiario.es, 2017]

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 hasta 9km/h. A partir de él surgieron varios robots humanoides con nuevas tecnologías, como *QRIO* de *Sony* en 2004 que era capaz de reconocer caras, el pequeño robot *Nao* en 2008 con su habilidad para interactuar con el ser humano o *Pepper* (figura 1.3), 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. 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 (figura 1.5). Entre ellas, podríamos destacar las aplicaciones domésticas con robots de limpieza como los famosos *Roomba* de iRobot, el sector de la agricultura con vehículos autónomos y monitorizacion 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 Waymo 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 programaciónde robots son algo que hasta hace poco se consideraba algo de nicho y que sólo se enseñaba en algunas universidades, haciendo que su avance y desarrollo fuera lento. Hoy en día se considera algo tan fundamental, que incluso en algunas escuelas primarias se comienzan a enseñar 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 de robots, 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 docente de robótica[1], donde podrán programar usando robots reales como el robot Pepper o el TurtleBot2 (sección 3.3) 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 (sección 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.

Un gran ejemplo de este tipo de programación es *Scratch*[2]. Esta plataforma nos permite programar el comportamiento de avatares conocidos como *sprites* mediante

[1] **Laboratorio de robótica**: `https://labs.eif.urjc.es/index.php/laboratorios/edificio-laboratorio-iii/laboratorios-laboratorio-l3104/`

[2] **Scratch**: `https://scratch.mit.edu/`

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 (educación primaria y secundaria) como introducción a la programación por su simplicidad a la hora de entender conceptos básicos como bucles o condicionales.

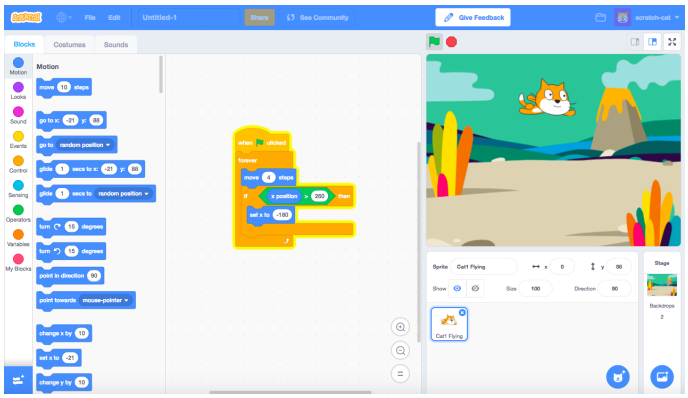


Figura 1.7: Plataforma Scratch.

VisualCircuit[3], la plataforma en la que se basa este Trabajo 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 (figura 1.8). 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 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.

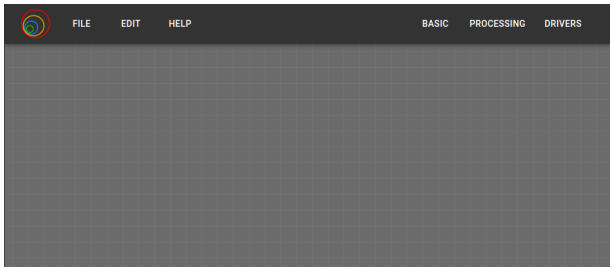


Figura 1.8: Plataforma VisualCircuit.

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

[3]**VisualCircuit**: `https://visualcircuit.org/`

Capítulo 2
Objetivos y metodología de Trabajo

2.1. Objetivos

En la herramienta VisualCircuit, antes de realizar este proyecto, ya existían bloques dedicados específicamente a la robótica para algunos sensores (cámara, odometría e IMU[1]) y para los motores usando ROS *Noetic*[2], pero al tratarse de una versión obsoleta del middleware, decidimos que era momento de actualizar a ROS2 *Humble*[3], ya que es la versión estable más moderna en el momento de realización de este Trabajo Fin de Grado.

Los objetivos concretos de este Trabajo Fin de Grado son los siguientes:

- Desarrollar bloques para sensores y actuadores usando ROS2 *Humble* y probar su correcto funcionamiento mediante circuitos simples de VisualCircuit. Profundizaremos más en el capítulo 4.
- Diseñar y construir aplicaciones complejas que usen estos bloques para comprobar su funcionalidad en situaciones reales:
  - Aplicación sigue-persona: usar reconocimiento visual para seguir a una persona tanto en entorno simulado como real, usando el robot TurtleBot2 (sección 3.3). Veremos más en el capítulo 5.
  - Aplicación *Virtual Force Field*: usar el láser y la odometría para navegar por el entorno evitando obstáculos. Examinaremos con mayor detalle en el capítulo 6.

[1] **IMU**: Inertial Measurement Unit
[2] **ROS Noetic**: `http://wiki.ros.org/noetic`
[3] **ROS Humble**: `https://docs.ros.org/en/humble/index.html`

2.2. Metodología

Este Trabajo Fin de Grado comenzó en abril de 2022 y finalizó en junio de 2023. Durante estos meses se ha seguido el siguiente modelo de trabajo:

- Reuniones cada dos semanas con el tutor del TFG para analizar los avances, recibir retroalimentación y buscar soluciones en caso de bloqueo.
- Uso de un blog[4] donde se iba actualizando el progreso antes de las reuniones, donde se puede comprobar el desarrollo cronológico del trabajo.
- Todo el material usado y desarrollado durante este Trabajo Fin de Grado se ha ido actualizando en un repositorio público de GitHub[5].

2.3. Plan de Trabajo

Como se ha dicho en el anterior punto, el desarrollo de este TFG ha durado algo más de un año. Este periodo se ha dividido en varias etapas:

1. **Pruebas con VisualCircuit**: Realizar circuitos para entender el funcionamiento de la plataforma, desarrollando bloques propios que modifiquen imágenes o compartan información.
2. **Inicio del TFG**: Configuración del entorno de pruebas (instalación de ROS2 *Humble*, diseño de mundos en *Gazebo*...)..
3. **Desarrollo de bloques drivers con ROS2**: Actualización y creación de bloques para sensores y actuadores usando ROS2 *Humble*.
4. **Prueba de los bloques en situaciones reales**: Desarrollo de aplicaciones robóticas avanzadas para probar el correcto funcionamiento de los bloques implementados.
5. **Memoria del Trabajo Fin de Grado**: Redacción de esta memoria.

[4]**Blog**: `https://roboticslaburjc.github.io/2022-tfg-david-tapiador/`
[5]**GitHub del TFG**: `https://github.com/RoboticsLabURJC/2022-tfg-david-tapiador`

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

Python[1] 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 (sección 3.5) se hace en este lenguaje.

3.2. ROS2 (Robot Operating System 2)

ROS[2] 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

[1]**Python ORG**: `https://www.python.org/`
[2]**ROS**: `http://wiki.ros.org/es`
[3]**Middleware**: 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 simultáneaos. 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.

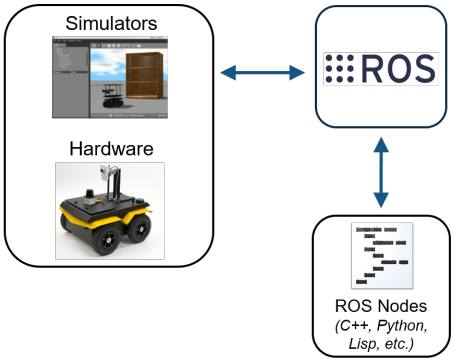


Figura 3.1: Comunicación del nodo Master con los nodos Intermedios y con distintos sensores y actuadores. Imagen obtenida de [Castro, 2017]

ROS cuenta con dos versiones principales, ROS y ROS2[4]. 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 (sección 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).

[4]**Versiones 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 RVIZ2[5], 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...).

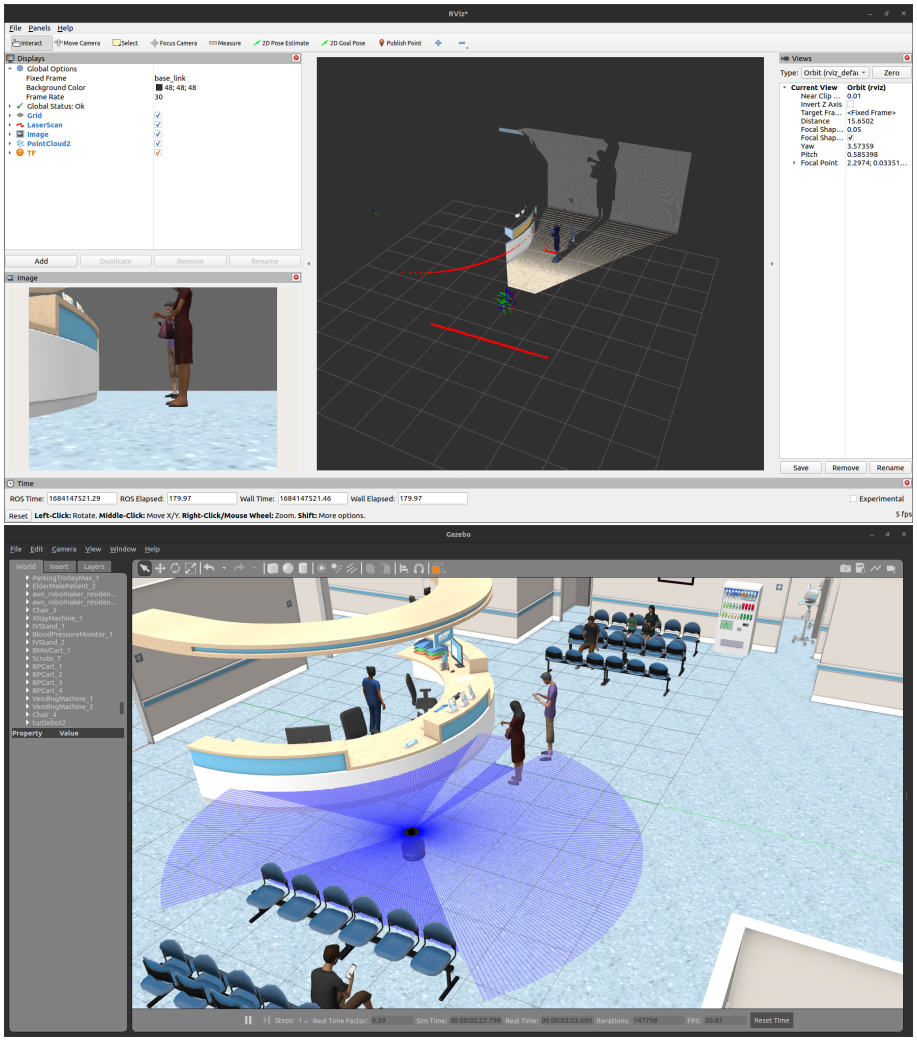


Figura 3.2: Ejemplo de RVIZ2 frente al mundo gazebo.

[5]**RVIZ2**: `https://github.com/ros2/rviz/tree/humble`

3.3. TurtleBot2

La URJC de Fuenlabrada, en sus laboratorios de robótica, cuenta con varios robots TurtleBot2[6] 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-Humble[7] 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]

[6] **TurtleBot2**: `https://www.turtlebot.com/turtlebot2/`

[7] **Drivers 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-D[8], 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 hasya 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 internet[9].

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 (sección 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 ODE[11], aunque se puede configurar con otros motores, como Bullet[12] o DART[13].

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 (sección 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`

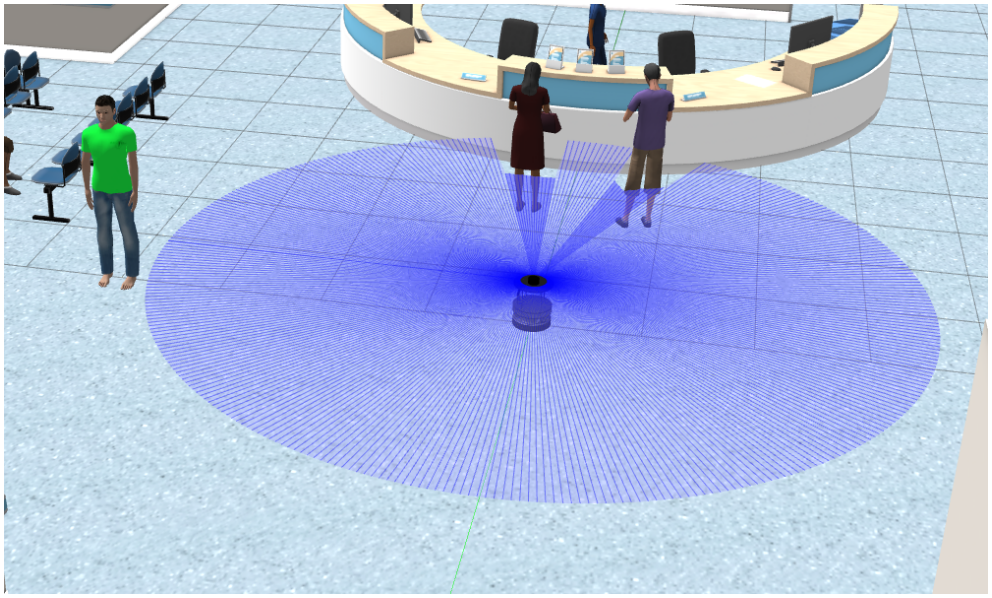


Figura 3.7: Ejemplo de ejecución en gazebo.

3.4.1. TurtleBot2 simulado

Para algunas partes del proyecto, como el desarrollo de *drivers* para ROS2 (capítulo 4) o el comportamiento VFF usando máquinas de estados (capítulo 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 Turtlebot2[19] (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`

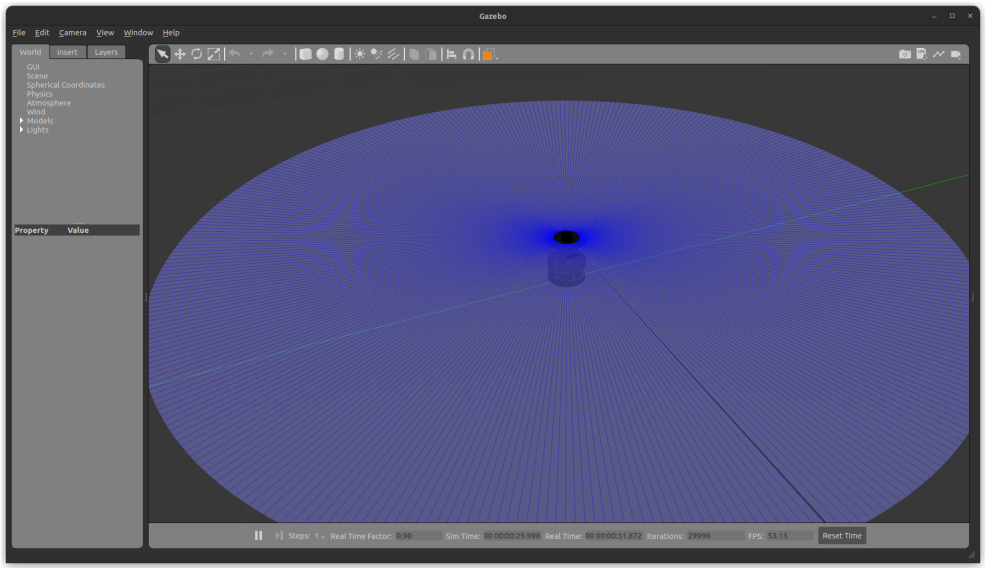


Figura 3.8: TurtleBot2 en gazebo.

3.5. VisualCircuit

VisualCircuit[20] es un editor visual online basado en programación por bloques de código en lenguaje *python* orientado al desarrollo de aplicaciones robóticas. Está desarrollado sobre IceStudio[21].

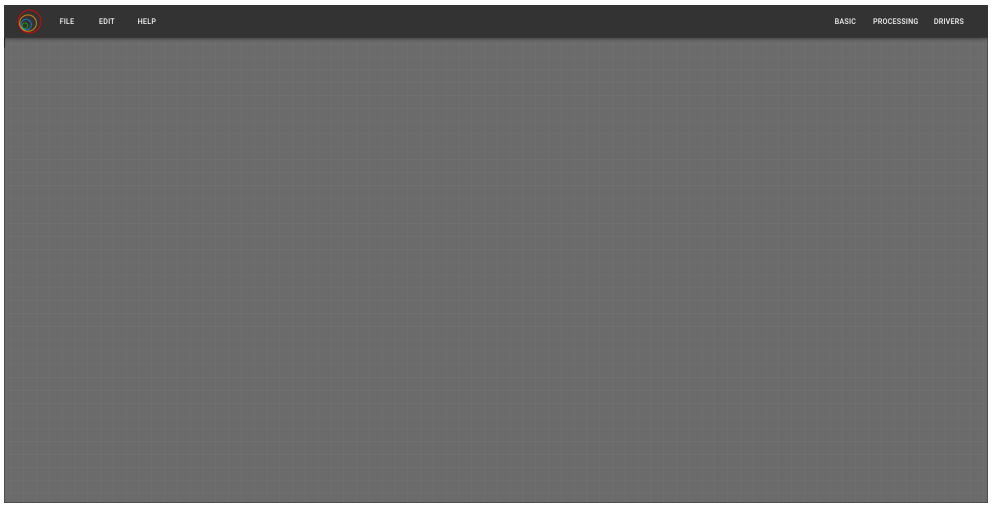


Figura 3.9: Página de VisualCircuit.

[20] **VisualCircuit Docs**: `https://jderobot.github.io/VisualCircuit/`
[21] **IceStudio Project**: `https://github.com/FPGAwars/icestudio`

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

- *Basic:* bloques simples, como inputs y outputs, bloques para definir parámetros y constantes, o bloques para insertar código propio.
- *Processing:*
  - *Control:* bloques de control (PID).
  - *OpenCV:* bloques relacionados con OpenCV[22] 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 imagenes desde archivos y desde cámaras, pantalla para mostrar las imágenes.
  - *ROS-Sensors:* cámara, odometría e IMU[23] 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.

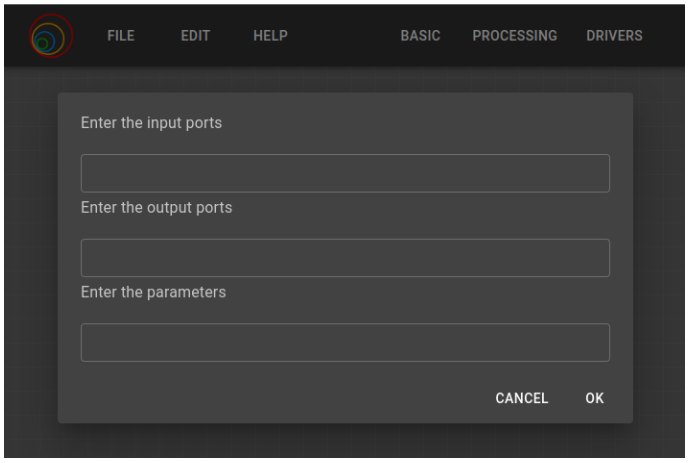


Figura 3.10: Creando un bloque en VisualCircuit.

[22] **OpenCV**: `https://opencv.org/`
[23] **IMU**: Inertial Measurement Unit

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.

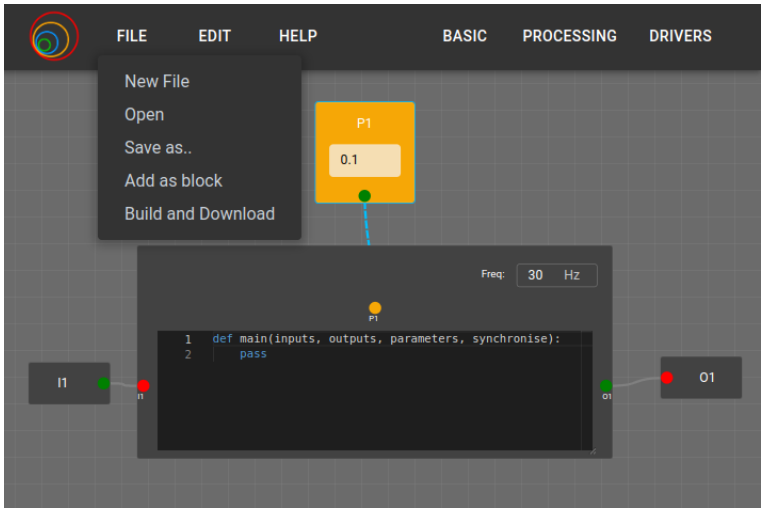


Figura 3.11: Guardando un bloque en VisualCircuit.

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 desee.

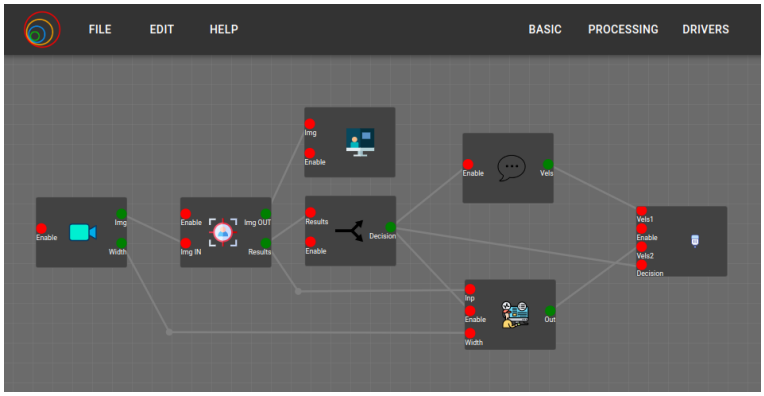


Figura 3.12: Ejemplo de proyecto en VisualCircuit.

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, la odometría 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.

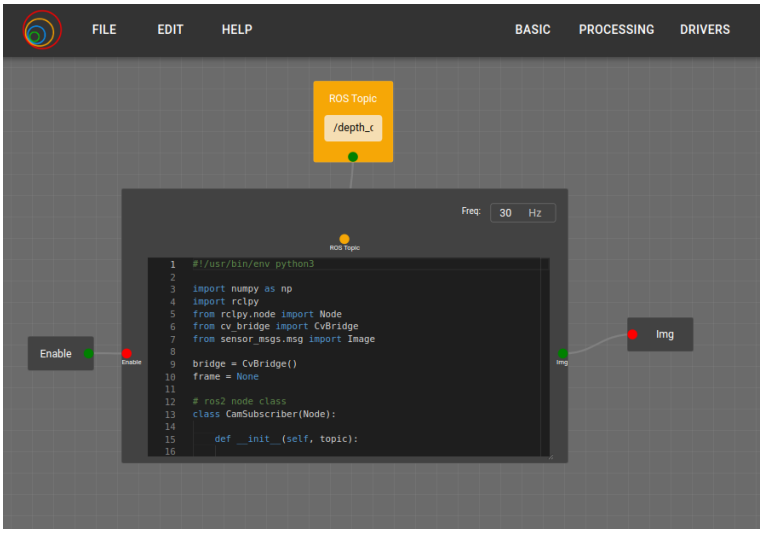


Figura 4.1: Modelo para crear bloques driver de sensores.

En cuanto al código que usaremos, seguiremos las indicaciones de los manuales de ROS2-humble[1] 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 compartirla por el cable *"output"*.

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

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

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

Una vez que tenemos el modelo general para estos bloques, hay que modificarlos para que funcionen con la cámara, el láser y la odometría. 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:

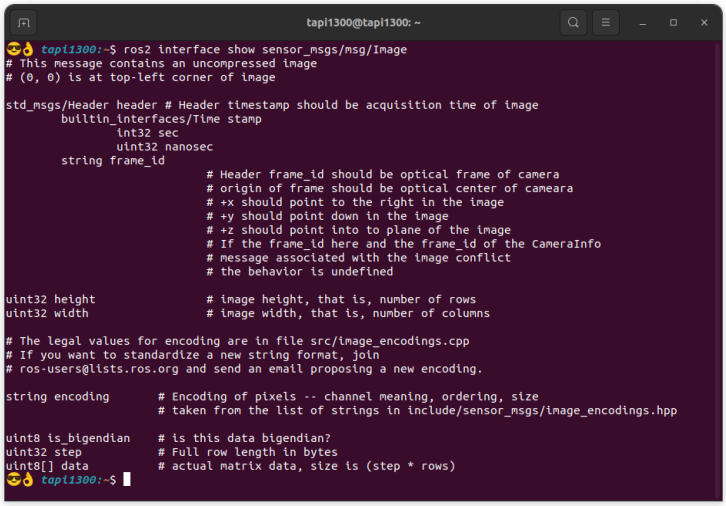


Figura 4.2: Estructura del tipo de mensaje *sensor_msgs/msg/Image*.

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 numpy[3], 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.

[2]**CvBridge**: `http://wiki.ros.org/cv_bridge`
[3]**Numpy**: `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:

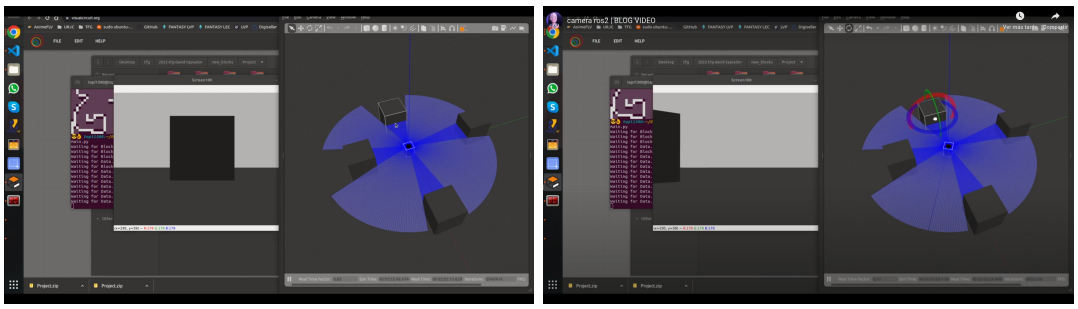


Figura 4.3: Secuencia de imágenes del bloque cámara ROS2 con TurtleBot2 simulado. Imagenes obtenidas de Youtube[4].

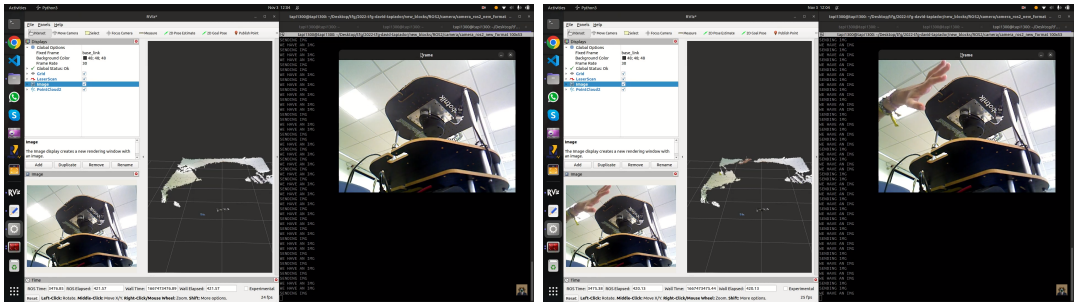


Figura 4.4: Secuencia de imágenes del bloque cámara ROS2 con TurtleBot2 real. Imagenes obtenidas de Youtube[5].

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

[5]**Vídeo 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`":

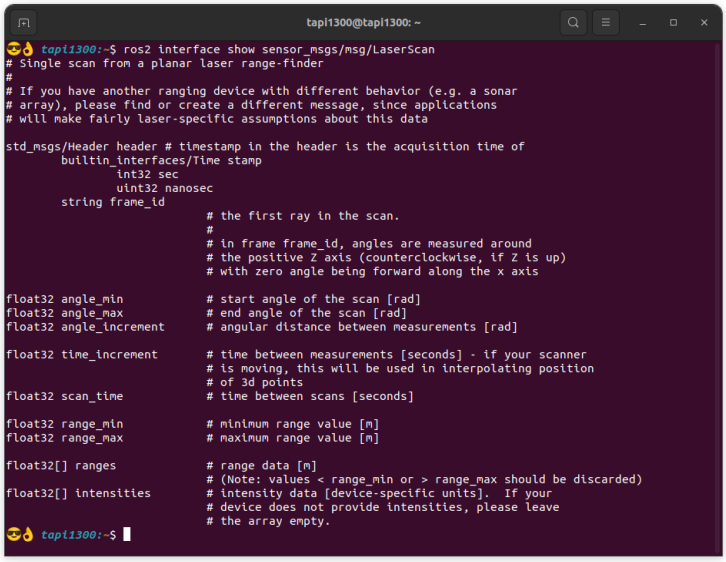


Figura 4.5: Estructura del tipo de mensaje *sensor_msgs/msg/LaserScan*.

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 guardarlo 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.extend((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:

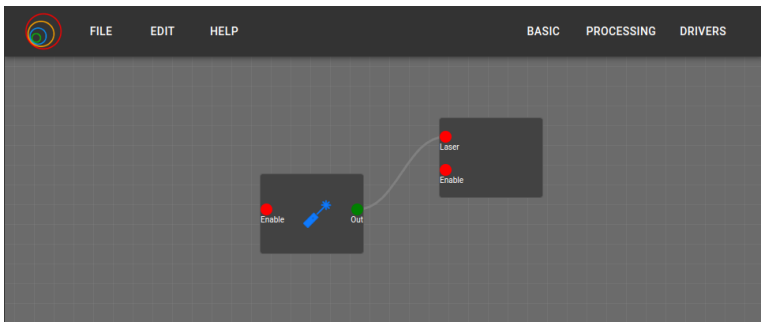


Figura 4.6: Circuito de pruebas del bloque laserROS2.

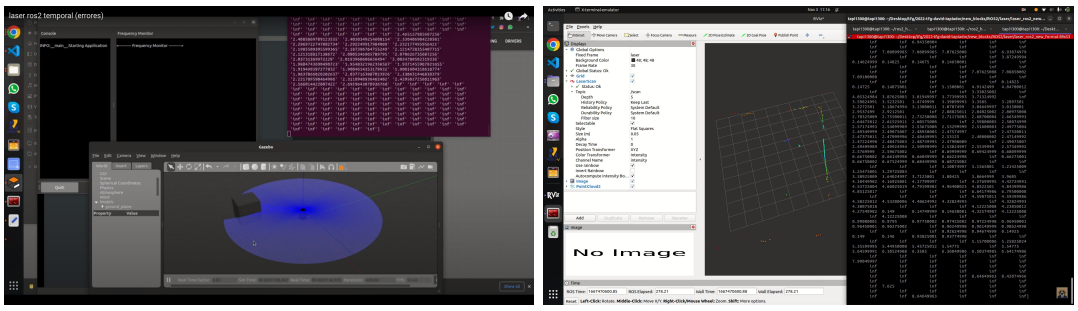


Figura 4.7: Ejemplo del bloque laserROS2 con TurtleBot2 simulado y real. Imagenes obtenidas de Youtube[6].

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

4.1.3. Bloque odometría ROS2

El bloque de la odometría debe devolver la posición del robot en el mundo, esto implica coordenadas X e Y, al igual que la posición angular del eje Z. En este caso, el tipo de mensaje del *topic* */odom* es *nav_msgs/msg/Odometry*, cuyos parámetros podemos comprobar mediante el comando "`ros2 interface show nav_msgs/msg/Odometry`con el siguiente resultado:

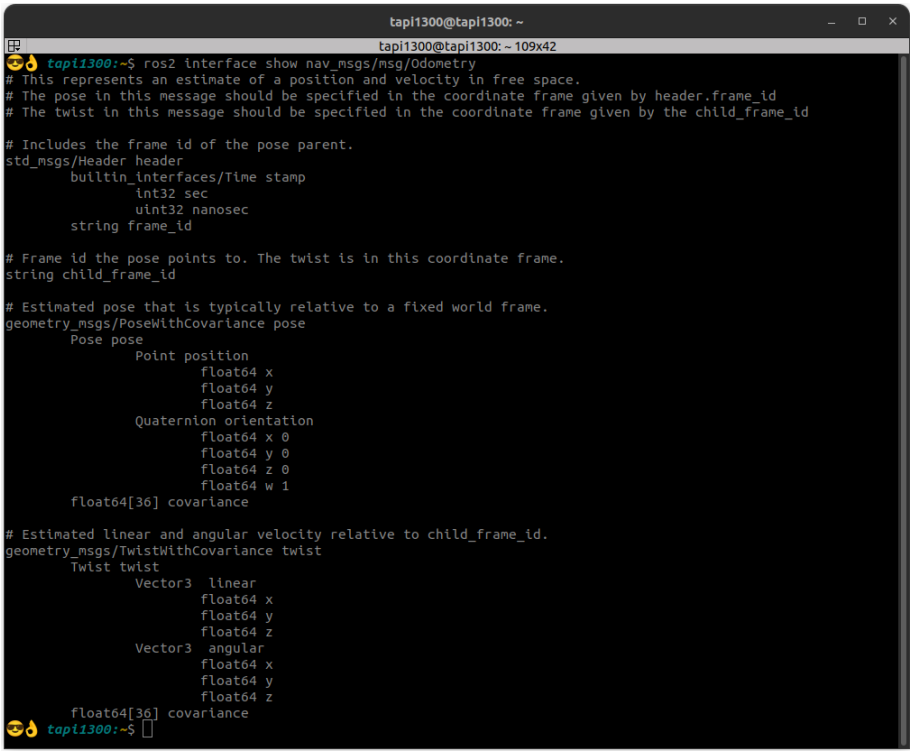


Figura 4.8: Estructura del tipo de mensaje *nav_msgs/msg/Odometry*.

En la función *callback* se debe almacenar estos tres valores y enviarlos por el cable de salida usando un array. 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*[7] de *python3*.

[7] **Librería Spicy**: `docs.scipy.org/doc/scipy/reference/generated/scipy.spatial.transform.Rotation.html`

```
        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 4.8: Funciones para obtener la fuerza repulsiva.

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 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:

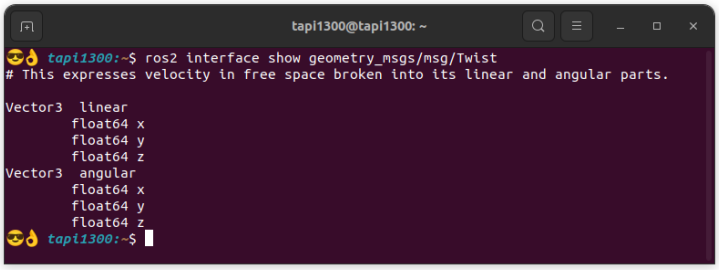


Figura 4.9: Estructura de tipo de mensaje *geometry_msgs/msg/Twist*.

Como se puede ver en la figura 4.9, 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-humble[8] y a partir de ahí crear el nodo publicador necesario para este bloque.

[8] 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'))

    while auto_enable or inputs.read_number('Enable'):
        velocities = inputs.read_array('Vels')
        if velocities != None:
            rclpy.spin_once(vel_publisher)
        synchronise()
```

Código 4.9: 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.

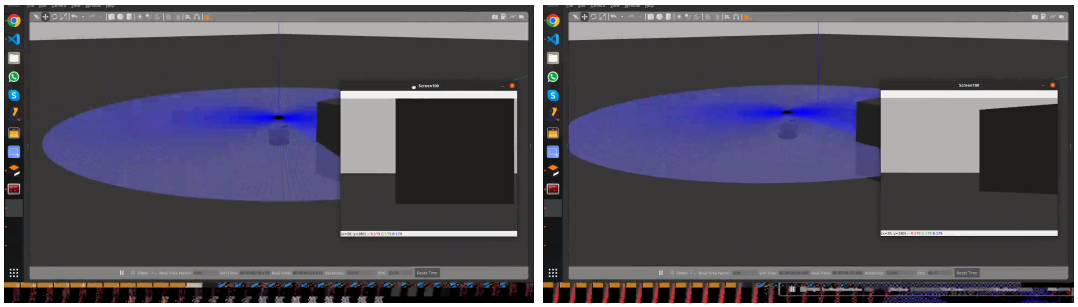


Figura 4.10: Secuencia de imágenes del bloque MotorDriverROS2 con TurtleBot2 simulado. Imagenes obtenidas de Youtube[9].

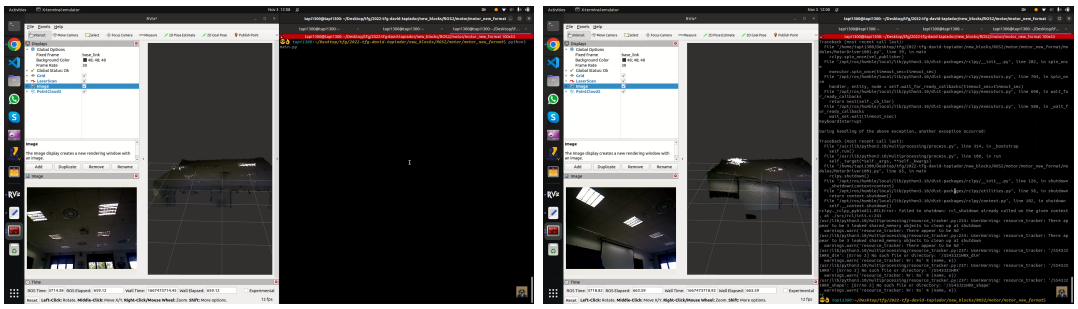


Figura 4.11: Secuencia de imágenes del bloque MotorDriverROS2 con TurtleBot2 real. Imagenes obtenidas de Youtube[10].

[9]**Vídeo bloque MotorDriverROS2 simulación**: `https://www.youtube.com/watch?v=a5J6Qccc5xk&t=73s&ab_channel=Tapi1300`

[10]**Vídeo bloque MotorDriverROS2 real**: `https://www.youtube.com/watch?v=MNaFWD9-ats&ab_channel=Tapii`

Capítulo 5
Aplicación sigue personas

Ahora que ya está integrado ROS2 dentro de la plataforma de VisualCircuit, se han creado varios proyectos usando los bloques drivers. El primero de ellos será un comportamiento de *follow-person* usando reconocimiento visual.

5.1. Descripción del comportamiento y escenario

El comportamiento sigue-personas o *follow-person* que buscamos desarrollar consiste en rotar en el sitio hasta encontrar a una persona mediante algoritmos de detección visual de objetos y mantener a la persona centrada en la imagen, a la vez que se mantiene al robot a una distancia constante de la persona. Este comportamiento se ha dividido en dos etapas: seguimiento estático (sólo velocidad angular) y seguimiento completo (velocidad angular y lineal).

Para esta aplicación se ha usado el robot TurtleBot2 (3.3). Dado que sólo se necesita la visión para seguir a la persona, sólo se usa la cámara RGB-D como sensor, ya que también ofrece la profundidad leída en cada *pixel* y sirve para mantener al robot a una distancia constante de la persona.

Para el entorno de pruebas se ha usado un modelo de persona teleoperada (figura 5.1) que creó en su TFG Carlos Caminero[1], compañero de la carrera.

[1] `https://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 usó Carlos incluía muchos elementos del entorno que no son necesarios en este caso, por lo que se ha modificado el mundo para dejar únicamente al robot y a la persona. El modelo del robot usado en las pruebas simuladas es el que se mencionó en el capítulo 3.4.1.

Las pruebas realizadas con el robot TurtleBot2 real se han hecho en el laboratorio docente de robótica, mencionado en la sección 1.1.1, instalando en el robot la cámara como único sensor (sección 3.3.3) y usando los motores de la base *kobuki* (sección 3.3.1) como único actuador.

Para configurar correctamente los sensores del robot real se deben tener instalados los paquetes para activar el kobuki (3.3.1), al igual que los necesarios para usar la cámara (3.3.3). para conseguir que todo funcione correctamente se han usado los siguientes comandos:

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

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

Código 5.1: Comandos para activar la cámara con ROS2 y lanzar el kobuki con el láser.

5.2. Seguimiento de persona sólo con rotación

5.2.1. Diseño del comportamiento

En esta primera aproximación, buscaremos mantener a la persona centrada en la imagen usando únicamente movimiento angular.

La lógica que seguirá será la siguiente: recibir la imagen del *topic* de la cámara y compartirla con el bloque de detección de objetos, enviar la imagen con las detecciones al bloque *screen* para visualizar en tiempo real lo que está analizando el robot, mandando también los resultados a un bloque encargado de decidir el comportamiento que seguir. 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 se ha desarrollado en la sección 4.2, que recibe tanto las distintas velocidades, como la decisión que se ha tomado, y envía al *topic* la que corresponda.

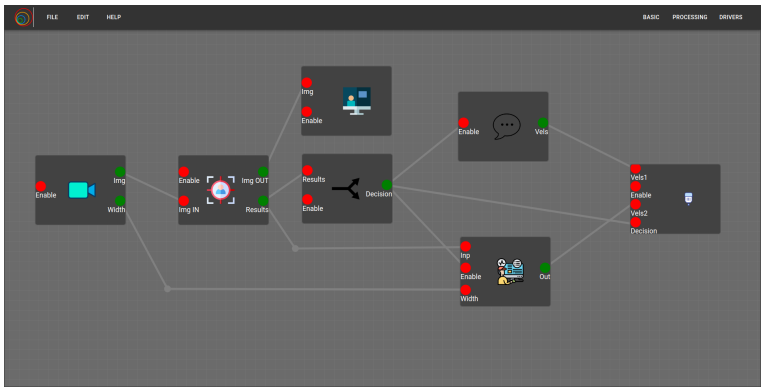


Figura 5.2: Circuito inicial del algoritmo sigue-persona.

5.2.2. Bloques específicos de la lógica de esta aplicación

El primer bloque de esta aplicación es el encargado de la detección de objetos mediante yolov3[2], un algoritmo de detección de objetos a tiempo real que permite identificarlos tanto en video como en imágenes usando redes neuronales (darknet[3]). El bloque ya está integrado en VisualCircuit, pero ha sido modificado para poder extraer también la localización de la *Bounding Box*[4] que corresponde a la persona y compartirla con otros bloques.

Para ello, una vez obtenidos los nombres de los objetos encontrados, se recorre toda la lista comprobando si hay alguna persona, en caso de haberla se envía la *Bounding Box* correspondiente por el cable. En caso contrario se envía un *array* con cuatro valores "-1" para indicar que está vacío.

[2]**YouOnlyLookOnce (YOLO)**: `https://pjreddie.com/darknet/yolo/`
[3]**DarkNet**: `https://pjreddie.com/darknet/`
[4]**Bounding Box**: Delimitación que se coloca alrededor de un objeto detectado por el algoritmo.

```
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)
```

Código 5.2: Modificación al bloque de la detección de objetos.

El siguiente bloque (5.3) es el que toma las decisiones de qué comportamiento seguir. Para ello, primero se espera hasta recibir algún resultado de la visión y así no mover al robot antes de haber podido analizar la situación. Una vez lleguen los resultados, se comprueba si es una *Bounding Box* válida, en caso de serlo, la decisión será seguir lo que indique el bloque PID para mantener a la persona centrada en la imagen. En caso de ser una caja vacía (*array* de "-1") se activa un contador para aplicar un filtro de paso bajo. Este filtro permite evitar cambiar de comportamiento por pequeños errores en la detección de objetos. Está establecido a 10, por lo que al llegar a 10 imágenes seguidas sin una persona en la imagen, se cambia de comportamiento al de la rotación en su búsqueda.

```
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
    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 5.3: 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 la información que le llega desde el cable y en caso de ser un "1"(*True*) envia una velocidad angular de 1rad/s para el eje Z e informa en la terminal que el robot está rotando.

```
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 5.4: Código del bloque de la rotación del sigue-persona.

En casi de encontrar a una persona en la imagen, se activa el bloque PID[5]. 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.

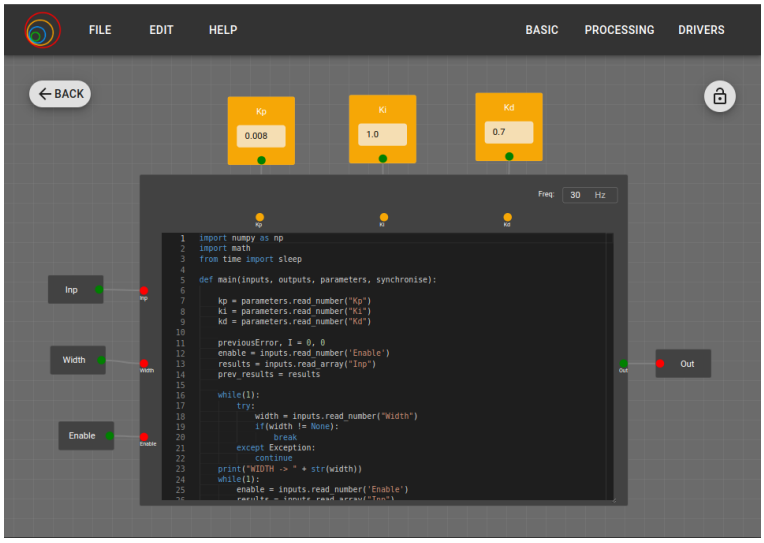


Figura 5.3: Circuito del bloque PID del sigue-persona en VisualCircuit.

Analizando el código del bloque PID (5.5), se puede ver que primero lee los tres parámetros del controlador y entra en el mismo bucle que se ha mencionado en el código bloque de decisión (5.3), donde se espera hasta obtener datos para empezar a trabajar. Después, en el bucle principal sólo se analizan y envían los datos del PID en

[5]**PID**: Controlador proporciona, 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

    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):
            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)
```

Código 5.5: Código del bloque del PID sigue-persona.

Como podemos ver en el código anterior (código 5.5), 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 este caso se trata del centro de la imagen, por lo que se usa 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 obtenida la velocidad angular, se manda al bloque de los motores. Este bloque es similar al creado en el apartado 4.1 pero ahora con 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 5.6: Código del bloque del *MotorDriver* sigue-persona.

5.2.3. Validación experimental

Al probarlo todo junto usando el TurtleBot2 real, se puede 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, obteniendo el resultado esperado.

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

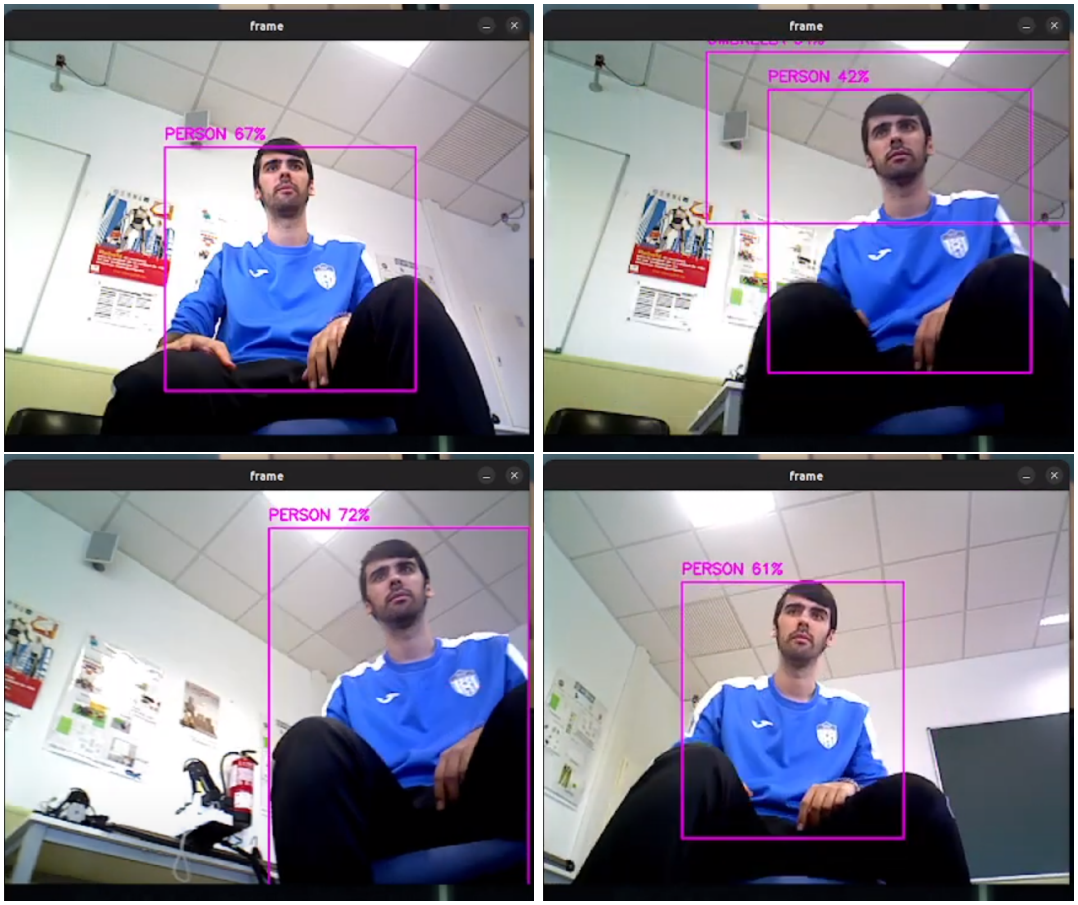


Figura 5.4: Secuencia de imágenes del sigue-personas sólo con rotación. Imagenes obtenidas de Youtube[6].

5.3. Seguimiento completo de persona

5.3.1. Diseño del comportamiento

La segunda etapa combina el comportamiento anterior con seguir linealmente a la persona. Para ello se debe leer también la información sobre la profundidad que nos da la cámara.

Como podemos ver, toda la rama inferior de bloques es la que se encarga del movimiento lineal, obteniendo la distancia con la persona mediante la profundidad captada por la cámara y buscando que el robot se mantenga a una distancia constante de la persona mediante el bloque PID. Por otro lado, la rama superior se encarga del movimiento angular.

[6] **Vídeo**: `https://www.youtube.com/watch?v=Uir_iqMOplc&ab_channel=Tapii`

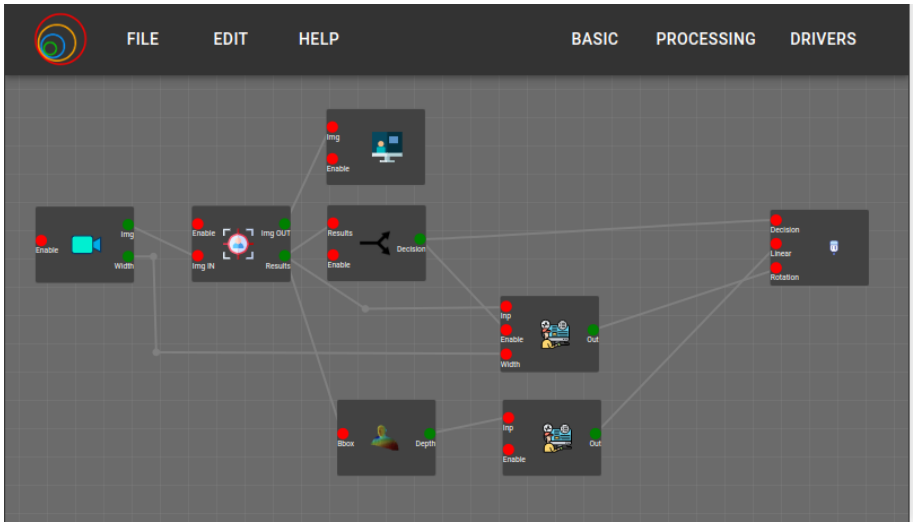


Figura 5.5: Circuito inicial del algoritmo sigue-persona.

5.3.2. Bloques específicos de la lógica de esta aplicación

La única parte de esta rama que ha cambiado frente a la anterior etapa es que en ésta no existe un bloque que envíe la velocidad de rotación, sino que está directamente implementado en el bloque MotorDriverROS2 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 5.7: Código del bloque del *MotorDriver* sigue-persona modificado.

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.

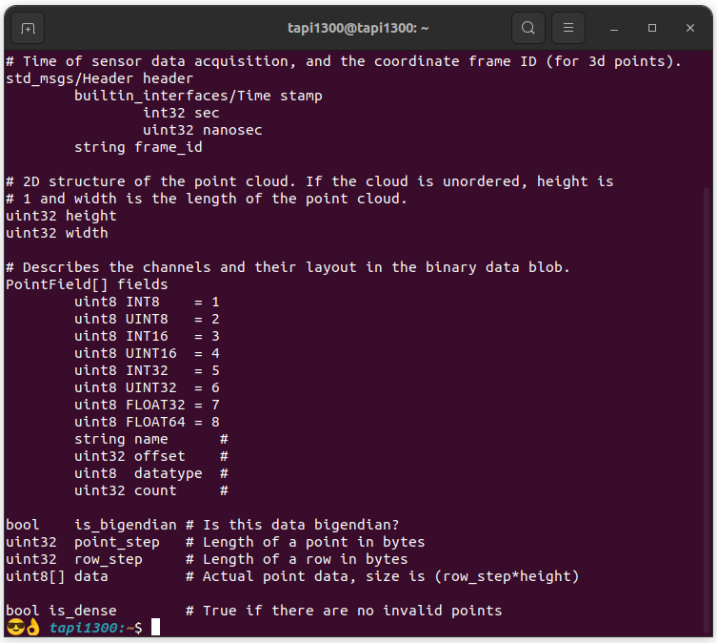


Figura 5.6: Estructura del tipo de mensaje *sensor_msgs/msg/PointCloud2*.

La forma en la que se guardan los datos dentro del campo *data* es en bytes, siguiendo la siguiente estructura para cada punto de la cámara (640x480 = 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 aparezcan 9830400 valores y la búsqueda final sea el número de filas (coordenada Y) multiplicado por el ancho total de la imagen más la posición actual de la 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 (*(width*y+x)*32+8*). Luego, usando la función "*unpack*"del paquete *struct* se transforman los siguiente 4 bytes (correspondientes a la coordenada Z) a un *float* y se obteniene 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 5.8: Código del bloque del *PointCloud2* del sigue-persona.

Esta información llega a un segundo bloque PID, que es el que se encarga de mantener esta distancia con la persona en 1.5 metros. El código de este bloque es similar al del otro bloque PID (5.5), pero ahora no se necesita una velocidad límite (el mínimo y máximo posibles linealmente no son tan peligrosos en comparación a las velocidades angulares altas).

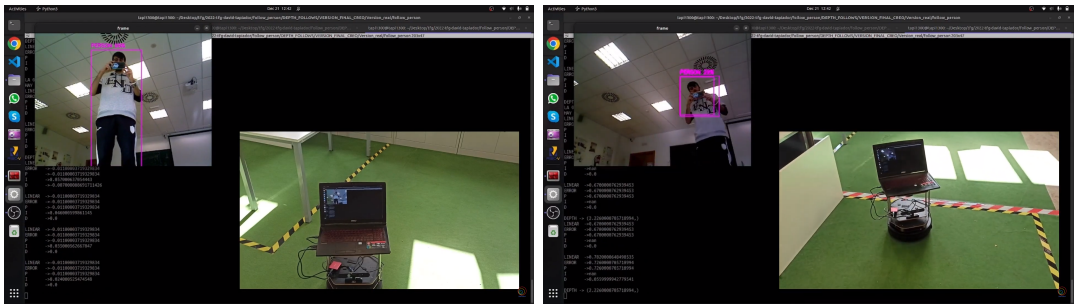
```
    import numpy as np
    import math
    from time import sleep
    def main(inputs, outputs, parameters, synchronise):
        auto_enable = True
        try:
            enable = inputs.read_number("Enable")
        except Exception:
            auto_enable = True
        kp = parameters.read_number("Kp")
        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 5.9: Código del bloque del PID de velocidad lineal del sigue-persona.

5.3.3. Validación experimental

Para comprobar que la aplicación sigue-personas completa funcionase, se puso al robot dentro de la sala del laboratorio y se grabó cómo seguía a una persona desde dos ángulos: la visión del robot y la visión de la persona (para observar qué movimientos hizo el robot). Los resultados que se pueden observar en la siguiente secuencia de imágenes (figura 5.7) extraídas del vídeo oficial demuestran el correcto funcionamiento de la aplicación y de los bloques desarrollados.



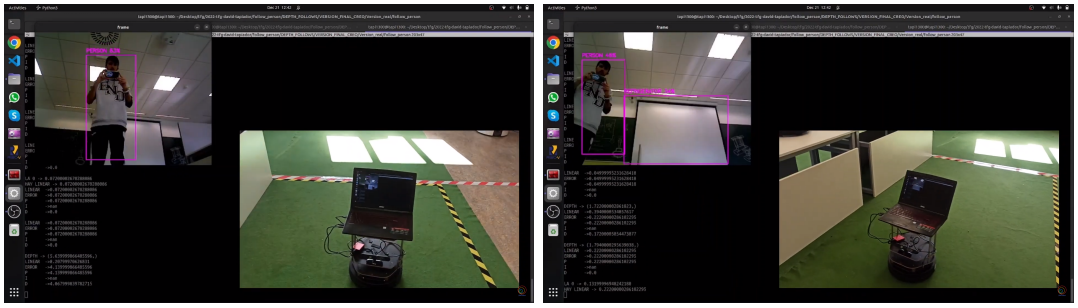


Figura 5.7: Secuencia de imágenes del sigue-personas completo. Imagenes obtenidas de Youtube[7].

[7] **Vídeo**: `https://www.youtube.com/watch?v=IknpvAs_jAo&ab_channel=JdeRobot`

Capítulo 6
Aplicación *Virtual Force Field*

Como ya se ha explicado, otra de las aplicaciones que se ha desarrollado consiste en navegación local del robot usando el algoritmo VFF mediante el uso de máquinas de estados.

Se han hecho dos implementaciones de esta aplicación: una canónica combinando fuerzas para recorrer un circuito y otra como autómata de estado finito, aprovechando los mecanismos que tiene VisualCircuit para activar y desactivar bloques, navegando a destinos aleatorios generados en la propia ejecución.

6.1. Campo de fuerzas virtuales (VFF)

El algoritmo de navegación local mediante campo de fuerzas virtuales o *virtual field force* consiste en programar el movimiento del robot a través de un lugar avanzando gracias a destinos temporales (como si fueran balizas) y esquivando los obstáculos entre la posición del robot y el destino actual 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 destino). 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 consiste en dos ramas principales, una para cada fuerza, y la unión de estas ramas mandando una velocidad final tanto lineal como angular a los motores del robot (figura 6.1).

En la rama superior está el sensor láser junto con un bloque para obtener una única fuerza repulsiva 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 destino 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 ponderadamente 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 (v, w) de traslación y de rotación al bloque MotorDriverROS2 (sección 4.2).

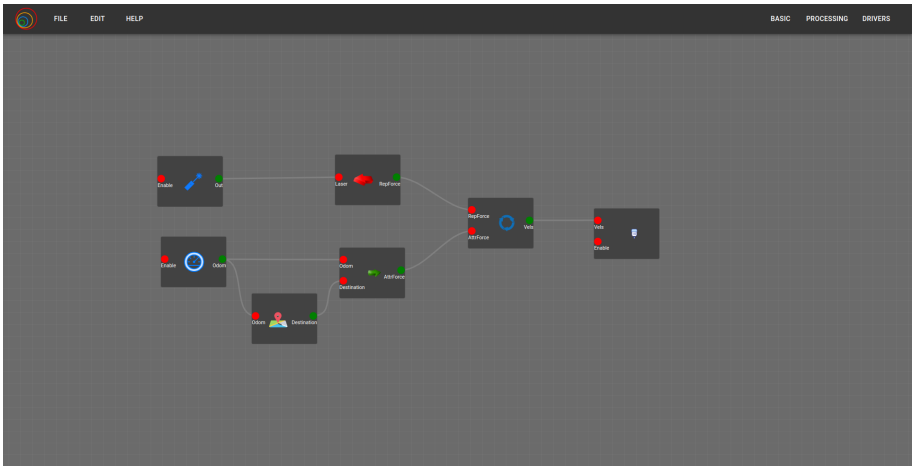


Figura 6.1: Circuito del algoritmo VFF.

El escenario de pruebas consiste en un circuito creado en el simulador Gazebo a base de bloques cúbicos como muros con algunas barras rojas (cubos rectangulares) horizontales que marcan aproximadamente dónde se encuentran los objetivos (figura 6.2). 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.

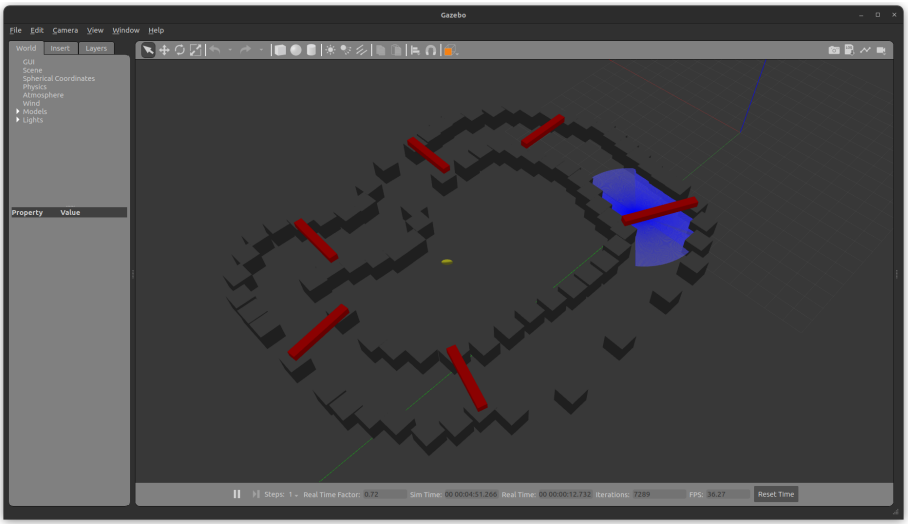


Figura 6.2: Mundo gazebo para probar el algoritmo VFF.

6.1.2. Bloques específicos de la lógica de esta aplicación

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 valore (x,y).

Ahora para calcular la fuerza repulsiva 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
    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 absoluta actual del robot y la posición que se ha marcado como destino actual. Para obtener la posición del robot se ha de utilizar el bloque de odometría planteado en el apartado 4.1.3, mientras que para obtener los destinos se ha creado el bloque generador de destinos, que usa la ubicación del robot para comprobar si el robot ha llegado al destino actual (teniendo en cuenta un margen) y mandar el siguiente dentro de la lista de destinos. También se cuenta con un contador de vueltas, aprovechando que el circuito de pruebas es circular.

```
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 6.2: Bloque generador de ubicaciones.

Estos dos *arrays* (posición y destino) los recibe el siguiente bloque, que es el que calcula la fuerza atractiva instantánea. 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 6.3: 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 los bloques descritos.

```
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 6.4: 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 (figura 6.2.2).

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

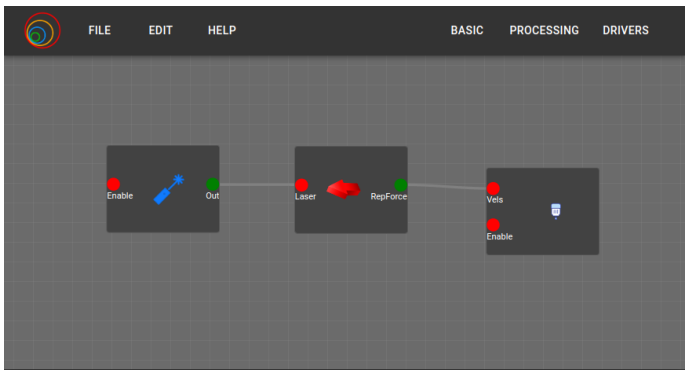


Figura 6.3: Circuito de VFF sólo con la fuerza repulsiva.

Como podemos comprobar en las siguientes secuencias y en sus vídeos correspondientes, la parte de la fuerza repulsiva cumple con su misión, 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 haya).

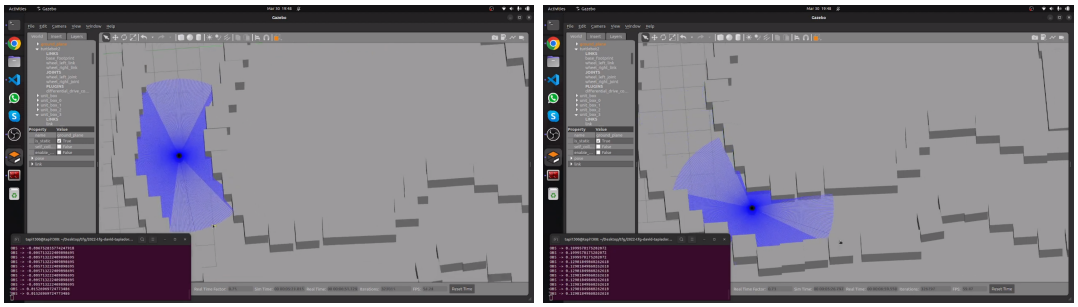


Figura 6.4: VFF usando sólo la fuerza repulsiva sin obstáculos. Imagenes obtenidas de Youtube[1].

[1]**Vídeo VFF fuerza repulsiva sin obstáculos**: `https://www.youtube.com/watch?v=uhtBRw96Zl4&ab_channel=Tapii`

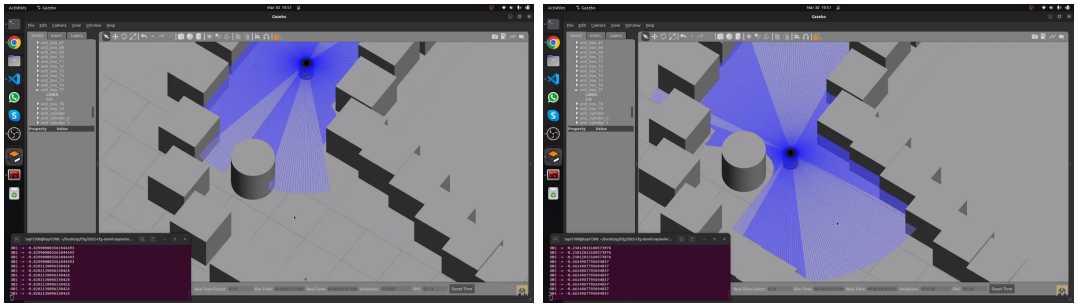


Figura 6.5: VFF usando sólo la fuerza repulsiva con obstáculos. Imagenes obtenidas de Youtube[2].

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

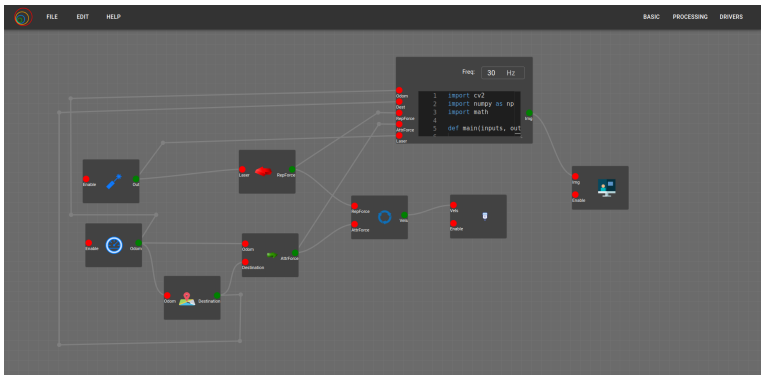


Figura 6.6: Circuito del algoritmo VFF.

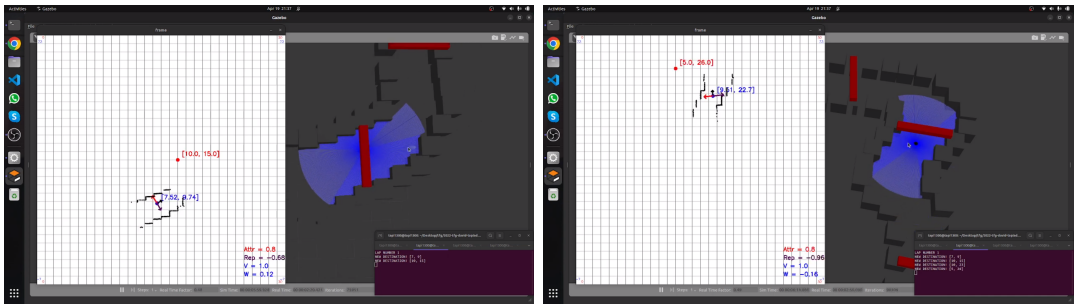


Figura 6.7: Secuencia de pruebas del algoritmo VFF. Imagenes obtenidas de Youtube[3].

[2]**Vídeo VFF fuerza repulsiva con obstáculos**: `https://www.youtube.com/watch?v=HeTFum_gTGw&ab_channel=Tapii`

[3]**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 practicamente desarrollado, se implementó una máquina de estados para completar la aplicación. Esta máquina de estados tiene 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 (figura 6.8), 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.

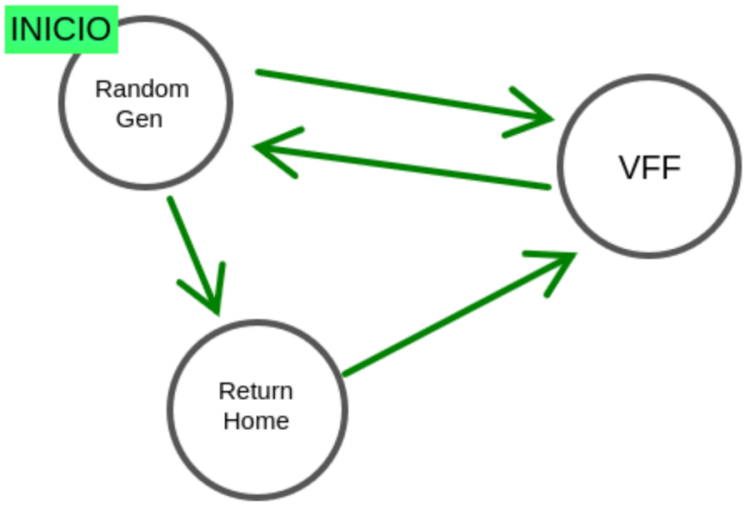


Figura 6.8: Diagrama de la máquinas de estados.

6.2.1. Diseño del circuito y escenario

Para implementar el comportamiento de la máquina de estados, se han implementado dos bloques 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.

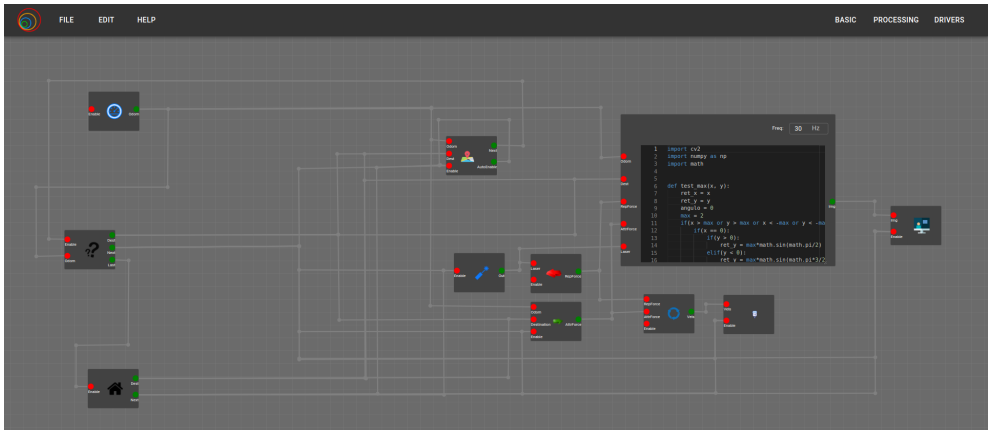


Figura 6.9: Circuito de VFF usando máquinas de estados.

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.

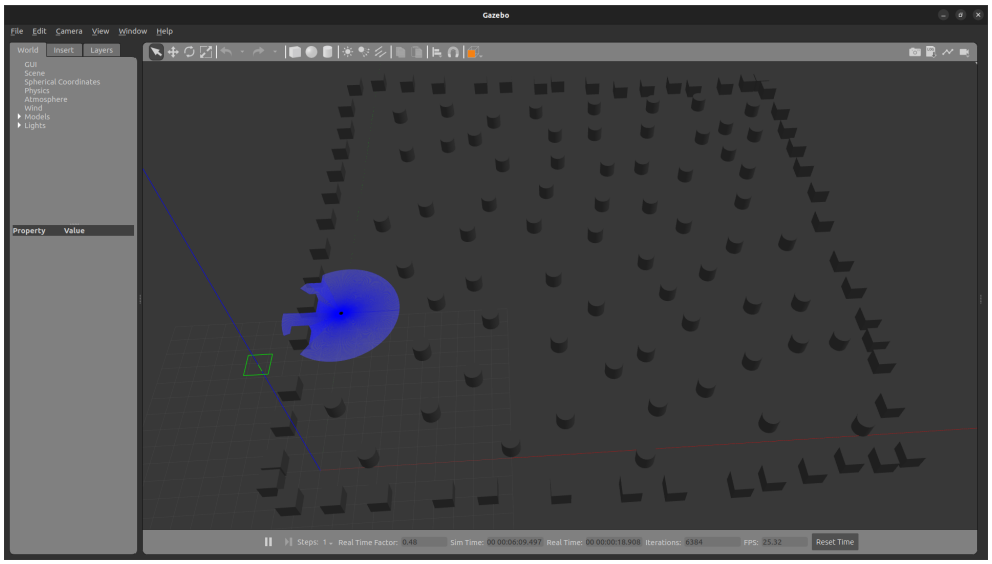


Figura 6.10: Mundo para probar el VFF con máquinas de estados.

6.2.2. Bloques específicos de la lógica de esta aplicación

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 (códigos 6.1 y 6.3), por lo que en vez de mandar una única fuerza escalar, 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 llegua al valor de *max_times* cambia al estado de volver al origen. Para generar la ubicación aleatoria, genera 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 6.5: 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 6.6: Bloque para volver al inicio.

El bloque que transforma las fuerzas a velocidades también cambia, ya que ahora hay que tener en cuenta también 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 dando 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 6.7: 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 vectoriales 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 esa imagen.

Aquí se usa la librería cv2 de opencv-python[4] 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 en el repositorio público[5] se puede acceder a él.

[4] **OpenCV-Python**: `https://pypi.org/project/opencv-python/`

[5] **Código bloque display**: `https://github.com/RoboticsLabURJC/2022-tfg-david-tapiador/blob/main/FSM/ZIPS/FSM_final/modules/Code_1.py`

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.

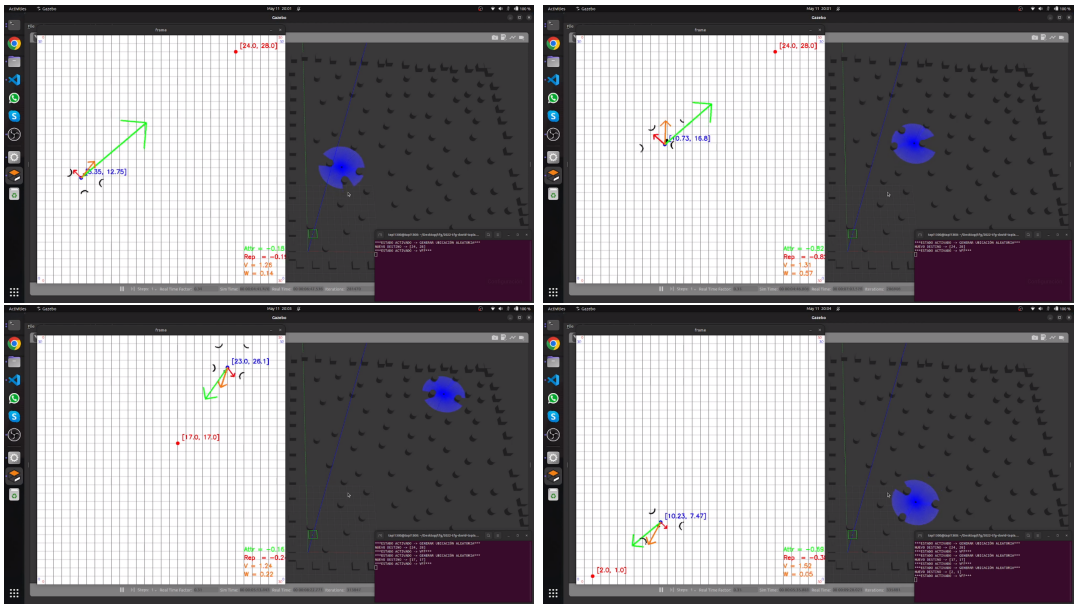


Figura 6.11: Secuencia de imágenes del algoritmo VFF usando FSM. Imagenes obtenidas de Youtube[6].

[6]**Vídeo**: `https://www.youtube.com/watch?v=EiBT8yqX29Q&ab_channel=Tapii`

Capítulo 7
Conclusiones

Tras detallar en profundidad las mejoras aportadas a la herramienta VisualCircuit, en este último capítulo de la memoria del Trabajo Fin de Grado se hará un resumen de las metas logradas.

7.1. Conclusiones

El objetivo principal de este proyecto era actualizar la plataforma VisualCircuit para que incluyera bloques orientados directamente a la robótica usando ROS2 *Humble*. Este objetivo ha sido alcanzado con éxito, añadiendo a las bibliotecas de VisualCircuit los bloques correspondientes a los sensores (láser y cámara) y actuadores (motores) tanto reales como simulados. También se ha desarrollado un bloque *encoders* para obtener la posición del robot en el simulador.

Otro de los objetivos era comprobar el correcto funcionamiento de estos bloques mediante aplicaciones robóticas que lo demostraran. Mediante la aplicación sigue-personas (capítulo 5) y *Virtual Force Field* (capítulo 6) se puede comprobar que este objetivo también se ha cumplido.

Este TFG ha logrado enriquecer la plataforma VisualCircuit permitiendo desarrollar aplicaciones más avanzadas y acercarla cada vez más a ser una herramienta de referencia para programar robots.

7.2. Líneas futuras

Algunas formas en las que se podría continuar con el desarrollo planteado en este Trabajo Fin de Grado podrían ser:

- Desarrollar bloques para nuevos sensores, como podrían ser los *bumpers*, o actuadores como *leds* y sonidos.
- Avanzar hacia el mundo de los drones, incluyendo bloques que activen, desactiven y configuren los modos de vuelo y que permitan despegar y aterrizar, ya que la programación de drones suele ser más tediosa y ver el comportamiento de forma tan visual como permite VisualCircuit ayudaría en el proceso.
- Ampliar VisualCircuit de tal forma que se permita compartir bloques y comportamientos completos entre distintos usuarios mediante un foro comunitario, añadiendo más bloques prefabricados a las bibliotecas de bloques.

Bibliografía

[Amazon.com, 2011] Amazon.com, I. (2011). Imagen de la cámara asus-xtion. 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_robotics_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/.