Universidad de Alcalá
Escuela Politécnica Superior

Máster Universitario en Ingeniería Industrial

Trabajo Fin de Máster

Control remoto de robot P3-DX: Comparación entre robot real y virtualizado con ROS

Autor: Álvaro Salcedo Izquierdo

Director: Felipe Espinosa Zapata

ESCUELA POLITECNICA SUPERIOR

TRIBUNAL:

Presidente: Daniel Pizarro Pérez

Vocal 1º: Ignacio Parra Alonso

Vocal 2º: Felipe Espinosa Zapata

FECHA: …………………………………………………………………..

Por una nueva vida

AGRADECIMIENTOS

A mis padres, por todo su apoyo, su ayuda y su cariño, sin ellos esta etapa no se hubiera cerrado.

A mi abuelo, por preguntarme siempre, su cariño y por creer en mí.

A Raquel y Antonio por su cariño.

A Pilar, porque sé que has estado ayudándome desde donde estés.

A mi familia, por creer en mí.

A Ella, por su paciencia durante toda esta etapa, compañía y cariño.

A mis amigos, a todos aquellos que me han acompañado dentro de la universidad y fuera de ella.

A Felipe Espinosa y Carlos Santos, por su gran ayuda y guía en este proyecto.

Gracias.

Índice de Contenidos

Resumen………………………………………………………………. 15
Abstract…………………………………………………………...…15
Palabras clave (Keywords)……………………………………….........15
Resumen extendido…………………………………………………..17

Capítulo 1: Introducción 19
1.1. Objetivos... 21

Capítulo 2: Herramientas utilizadas 23
2.1. ROS, Robot Operating System... 23
2.1.1 Conceptos básicos de ROS 29
2.1.2 Herramientas/Comandos básicos de ROS. 32
2.1.3 Distribuciones de ROS 35
2.2. Gazebo 37
2.2.1. Distribuciones de Gazebo 39
2.2.2. Relación entre ROS y Gazebo 40
2.2.3. Modelos en Gazebo 41
2.2.3.1 Entornos 42
2.2.3.2. Robots... 43
2.3. RVIZ 45
2.4. Matlab/Simulink 46
2.4.3. RTW: Real Time Workshop.. 48
2.5. LINUX/RTAI 48
2.6. Resumen de las herramientas instaladas en cada ordenador 50

Capítulo 3: Desarrollo 51

3.1. Plan de Trabajo 52

3.2. Robot P3-DX 54

3.3. Virtualización física y funcional del robot P3DX así como del entorno. 54

3.3.1. Programación de ROS/Gazebo en PC de robot virtual 55

3.3.1.1. Creación del Robot P3-DX en Gazebo 55

3.3.1.2. Creación del entorno en Gazebo 64

3.3.1.3. Intercambiando velocidades del robot de Gazebo desde ROS 71

3.4. Descripción del algoritmo de control y generación de referencias a implementar en el PC Centro Remoto 78

3.4.1. Identificación de la planta P3DX 78

3.4.2. Solución de control para el seguimiento de consignas del P3DX, servosistema 80

3.5. Comunicación entre PC Centro Remoto – PC Robot Virtual / Robot Real 87

3.5.1. Socket 87

3.5.2. S-Function 92

3.5.3. Diferencias en la comunicación entre el robot real y el virtual 97

3.5.3.1 Nodo de comunicación en robot virtual 97

3.5.3.2. Nodo de comunicación en robot real 98

3.6. Procedimiento a seguir para ejecutar la solución de control del Centro Remoto en Robot Virtual/Real 105

3.6.1. En Robot Virtual 105

3.6.2. En Robot Real 106

Capítulo 4: Extensión del trabajo a un convoy de robots 109

4.1. Virtualización física y funcional de un convoy de robots P3DX 110

4.2. Descripción del algoritmo de control para el guiado de robots P3DX en convoy 118

4.3. Comunicación entre PC Centro Remoto - Convoy Virtual/Real 119

4.3.1. Modificación en el socket del centro remoto 119
4.3.2. Modificación en nodo de comunicación del Convoy Virtual 121
4.4. Procedimiento a seguir para ejecutar la solución de control del Centro Remoto en Convoy Virtual y Convoy Real 122
4.4.1. En Convoy Virtual 122
4.4.2. En Convoy Real 122

Capítulo 5: Propuesta de evitación de obstáculos 123
5.1. Descripción del algoritmo de control para evitación de obstáculos 123
5.2. Implementación en robot virtual/real 130
5.2.1. En Robot Virtual 132
5.2.2. En Robot Real 136

Capítulo 6: Validación de resultados emulados con robot virtual e implementados en robot real 139
6.1. Servosistema en robot virtual y robot real 139
6.2. Guiado de un conjunto de robots: unidades P3DX y virtualización con ROS 142

Capítulo 7: Conclusiones 147
7.1. Trabajo futuro 148

ANEXO 149
Anexo 0: Implementación de S-Function y generación de ejecutable con RTW 151

Bibliografía 155

Índice de Ilustraciones

Ilustración 1.1: Tareas habituales en el ámbito de la robótica...................................... 19
Ilustración 1.2: Interfaz de las plataformas SAMON y USARsim................................ 20
Ilustración 1.3: Robot virtual y robot real controlados desde un centro remoto con el mismo algoritmo de control.. 21
Ilustración 2.1: Logo de Robot Operating System (ROS).. 24
Ilustración 2.2: Estadísticas de repositorios de ROS.. 24
Ilustración 2.3: Últimos datos oficiales de los paquetes alojados en los repositorios de ROS. 24
Ilustración 2.4: Plataformas robóticas soportadas por ROS... 25
Ilustración 2.5: Comunicación entre plataformas con ROS. 25
Ilustración 2.6: Diferencias entre ROS y un sistema operativo.................................... 26
Ilustración 2.7: Comunicación interna de ROS. 27
Ilustración 2.8: Alternativas de comunicación entre nodos.. 27
Ilustración 2.9: Ejemplo publicación-subscripción de un topic en ROS....................... 28
Ilustración 2.10: Estructura de un paquete de ROS.. 29
Ilustración 2.11: Gráfico de nodos y topics que proporciona Rosgraph. 34
Ilustración 2.12: Árbol de transformadas generado con TF. 35
Ilustración 2.13: Distribuciones de ROS. 36
Ilustración 2.14: Logo de Gazebo... 37
Ilustración 2.15: Robots famosos en entorno Gazebo. 38
Ilustración 2.16: Distribuciones de Gazebo.. 39
Ilustración 2.17: Gazebo como nodo de ROS. 40
Ilustración 2.18: Importación de modelos en Gazebo. 41
Ilustración 2.19: Interfaz building editor de Gazebo... 42
Ilustración 2.20: Estructura general de los componentes/modelos en Gazebo.............. 44
Ilustración 2.21: Logo de RVIZ. 45
Ilustración 2.22: Interfaz RVIZ. 45
Ilustración 2.23: Logo de Matlab. 46
Ilustración 2.24: Diagrama de bloques en Simulink. 47
Ilustración 2.25: Logo de Linux. 48
Ilustración 2.26: Herramientas instaladas en cada ordenador. 50
Ilustración 3.1: Etapas del trabajo con Centro remoto, robot real y robot virtual.......... 51
Ilustración 3.2: Enlaces entre las herramientas utilizadas.. 53
Ilustración 3.3: Montaje interior de la caja y plataforma P3-DX utilizada. 54
Ilustración 3.4: Componentes en 3D del robot virtual P3-DX. 56
Ilustración 3.5: Robot virtual inicial.. 56
Ilustración 3.6: Robot virtual final utilizado. 57
Ilustración 3.7: Elemento link con sus componentes principales.................................. 59
Ilustración 3.8: Elemento joint. 60
Ilustración 3.9: Modelo P3-DX final en un mundo vacío. 63

Ilustración 3.10: Pasillo Oeste Escuela Politécnica en formato CAD. 64
Ilustración 3.11: Medidas del entorno pasillo Oeste Escuela Politécnica. 64
Ilustración 3.12: Paleta para añadir paredes, ventanas, puertas y escaleras en Building Editor de Gazebo. 65
Ilustración 3.13: Añadir paredes a partir de sus puntos inicial y final en Buildin Editor de Gazebo. 66
Ilustración 3.14: Añadir puertas a partir de su pose, ancho y alto en Building Editor de Gazebo. 67
Ilustración 3.15: Pasillo Oeste (primera planta de la EPS) creado con building Editor en Gazebo. 68
Ilustración 3.16: Robot virtual P3DX dentro del entorno virtual pasillo Oeste. 71
Ilustración 3.17: Gráfico de nodos y topics para enviar velocidades al robot virtual. ... 72
Ilustración 3.18: Topics activos una vez lanzado el robot virtual. 73
Ilustración 3.19: Gráfico de nodos y topics para recibir velocidades del robot virtual. 75
Ilustración 3.20: Nodo publicando velocidades y subscribiéndose a los datos del robot. 77
Ilustración 3.21: Modelo del P3DX en VVEE en Simulink. 80
Ilustración 3.22: Diagrama de bloques de un servosistema en Simulink. 81
Ilustración 3.23: Error y salida del servosistema diseñado mediante Lyapunov............ 85
Ilustración 3.24: Diagrama de bloques de un Servosistema con observador en Simulink. 86
Ilustración 3.25: Diagrama de bloques del observador en Simulink. 86
Ilustración 3.26: Protocolo de comunicación entre el centro remoto y el robot virtual/real. 88
Ilustración 3.27: Librería de Simulink de la S-Function. 92
Ilustración 3.28: Diagrama de bloques de un Servosistema-estimador con S.Function en Simulink. 93
Ilustración 3.29: Comunicación con S-Function. 96
Ilustración 3.30: Comunicación entre centro remoto y robot real/virtual. 97
Ilustración 3.31: Comunicación completa entre Centro Remoto y Robot Virtual. 97
Ilustración 3.32: Interfaz entre un programa de control y el robot P3DX utilizando Player. 99
Ilustración 3.33: Interfaz entre un programa de control y el robot P3DX utilizando ARCOS. 100
Ilustración 3.34: Detalle del funcionamiento interno de la interfaz ARCOS. 101
Ilustración 3.35: Análisis de la incertidumbre utilizando el driver Player (izquierda) y ARCOS (derecha) con un Ts=100ms. 102
Ilustración 3.36: Comparación de la respuesta temporal de la velociddad lineal del robot real utilizando el driver Player y ARCOS. 103
Ilustración 3.37: S-Function con el driver creado mediante la interfaz ARCOS. 103
Ilustración 3.38: Comunicación completa entre Centro Remoto y Robot Real. 104
Ilustración 4.1: Paquetes de robots P3DX para la generación de un convoy. 110
Ilustración 4.2: Topics pertenecientes a los diferentes robots virtuales del convoy..... 112
Ilustración 4.3: Estructura .launch para lanzar un convoy de 4 robots en Gazebo....... 113

Ilustración 4.4: Convoy de 4 robots P3DX en el entorno Gazebo. 114
Ilustración 4.5: Convoy de 4 robots con cajas identificativas. 114
Ilustración 4.6: Nodo publicando velocidades y subscribiéndose a los datos de un convoy de 4 robots. 117
Ilustración 4.7: Propuesta de control a un conjunto de robots.................................... 118
Ilustración 4.8: Sockets dependiendo de si se comunica con convoy virtual o real..... 120
Ilustración 4.9: Comunicación entre Centro Remoto y convoy de 4 robots ROS/Gazebo. 121
Ilustración 5.1: Diagrama de bloques FLC con evitación de obstáculos y modelo cinemático del robot. 124
Ilustración 5.2: Partes del Fuzzy Logic Controller. 125
Ilustración 5.3: Conjuntos borrosos de las entradas. 127
Ilustración 5.4: Conjuntos borrosos de las salidas.. 128
Ilustración 5.5: Trayectoria seguida por el modelo cinemático del robot con el control FLC de evitación de obstáculos... 129
Ilustración 5.6: Evitación del obstáculo situado en (1,0). 129
Ilustración 5.7: Ganancias para controlar la distancia de seguridad con respecto al obstáculo.. 130
Ilustración 5.8: Obstáculo: Azul representa la caja a esquivar. Los puntos rojos representan los obstáculos a esquivar por parte del control. 130
Ilustración 5.9: Trayectoria seguida por el robot esquivando una caja como obstáculo. 131
Ilustración 5.10: Subsistema "robot", con S-Function y modelo cinemático.............. 132
Ilustración 5.11: Comunicación entre centro remoto y robot virtual calculando la pose en el centro remoto.. 132
Ilustración 5.12: Comunicación entre centro remoto y robot virtual obteniendo la pose del topic odom.. 133
Ilustración 5.13: Insertar obstáculos del repositorio en Gazebo................................. 134
Ilustración 5.14: Modificación de pose y tamaño de los objetos introducidos en Gazebo. 134
Ilustración 5.15: Caja obstáculo junto con unidad móvil P3DX en Gazebo. 136
Ilustración 5.16: Comunicación entre centro remoto y robot real obteniendo la pose y orientación del mismo. 137
Ilustración 6.1: Captura del movimiento del robot virtual (izquierda) y real (derecha) a lo largo del pasillo Oeste en el punto de partida OL3. 139
Ilustración 6.2: Captura del movimiento del robot virtual (izquierda) y real (derecha) a lo largo del pasillo Oeste algunos segundos despues de partir del OL3.......................... 140
Ilustración 6.3: Captura del movimiento del robot virtual (izquierda) y real (derecha) a lo largo del pasillo Oeste cerca del punto objetivo OL5. 140
Ilustración 6.4: Velocidad lineal (arriba) y velocidad angular (abajo) reportadas tanto por el robot virtual(rojo) y real(verde) respecto a la consigna(azul). 141
Ilustración 6.5: Diagrama de bloques de Simulink del control aplicado al conjunto de 4 robots. 142

Ilustración 6.6: Captura del movimiento del conjunto virtual (izquierda) y real (derecha) a lo largo del pasillo Oeste en el punto de partida OL3. 143
Ilustración 6.7: Captura del movimiento del conjunto virtual (izquierda) y real (derecha) a lo largo del pasillo Oeste cerca del punto objetivo OL5... 143
Ilustración 6.8: Consigna de velocidad lineal aplicada a cada robot. 144
Ilustración 6.9: Consignas de velocidad angular aplicadas a los robots del conjunto.. 144
Ilustración 0.1: Configuración del bloque S-Function. 151
Ilustración 0.2: Configuración Simulink. 152
Ilustración 0.3: Configuración de RTW en entorno Simulink. 152
Ilustración 0.4: Resultado exitoso en el proceso de generación de código. 153
Ilustración 0.5: Lanzamiento de ejecutable creado con Matlab/Simulink/RTW.......... 154

RESUMEN

Este trabajo fin de máster se centra en el diseño de los interfaces necesarios para facilitar la evaluación de controladores remotos de robots, permitiendo ejecutar la solución de control primero en un robot virtual reduciendo así el tiempo, los riesgos y los costes asociados a la experimentación real, para posteriormente ejecutar la solución en un robot real haciendo mínimas modificaciones en el código.

Se utiliza la herramienta Matlab/Simulink/RTW para el diseño e implementación de las soluciones de control debido a su alto potencial en este ámbito, y la herramienta ROS (Robot Operating System)/Gazebo para la emulación del robot.

ABSTRACT

This TFM focuses on the necessary interfaces design to facilitate the evaluation of remote controllers of robots, allowing to execute the control solution first in a virtual robot in order to reduce risks, time and costs associated with the real experimentation. Then, to execute the solution in a real robot doing minimal modifications in the code.

Matlab/SimulinkRTW tool is used for the design and implementation of control solutions due to its potential in this area, and ROS tool (Robot Operating System)/Gazebo for the emulation robot.

PALABRAS CLAVE / KEYWORDS

ROS, Gazebo, virtualización de robot P3-DX, virtualización de convoy, evitación de obstáculos.

RESUMEN EXTENDIDO

Para el diseñador de sistemas de navegación y control en robótica, la disponibilidad de herramientas software facilita tanto el análisis como la síntesis de soluciones. Actualmente existen múltiples herramientas, comerciales y open-source, con sus correspondientes interfaces de integración que proponen una alta variedad de soluciones para problemas de robótica enfocados a facilitar al diseñador de este ámbito su trabajo.

Entre estas herramientas destaca el entorno Matlab/Simulink/RTW para el diseño e implementación de soluciones de control y Robot Operating System (ROS), con aplicaciones asociadas como Gazebo, para el modelado de robots en entornos dinámicos. Con ellos se facilita la evaluación con garantías de la solución definitiva de aplicaciones en robótica antes de ser implementada sobre el propio robot real, por lo que nos ahorra tiempo, espacio y dinero implementando las soluciones en un robot virtual abstrayéndonos del real, y una vez que la solución convezca al desarrollador, poder implementarla en el robot real haciendo mínimas modificaciones en el código/aplicación.

Este trabajo fin de máster se centra en ofrecer al diseñador una solución en la que pueda verificar sus soluciones de control en un robot virtual sin tener que utilizar solo una herramienta específica tanto para el diseño del controlador como para la implementación de este diseño en un robot virtual, es decir, que no tenga que enfocar su solución de diseño a la herramienta que le permitirá implementarla en el robot virtual. Por lo que se propone que el diseñador pueda realizar la solución de control en la herramienta Matlab/Simulink/RTW, la cual destaca por su gran enfoque en el diseño e implementación de soluciones de control, abstrayéndose de la herramienta donde ejecutará esta solución en un robot virtual. Y utilizar la herramienta Robot Operating System (ROS) y la aplicación asociada Gazebo, las cuales destacan por su gran potencial en la simulación de robots en entornos dinámicos, para implementar la solución de control en un robot virtual.

Con esto expuesto, el objetivo principal del trabajo será realizar una herramienta que se encargue de dicha comunicación y que la traspolación al robot real sea lo menos tediosa posible, por lo que el aporte principal que aporta este trabajo fin de máster es permitir que la misma solución de control, ejecutada en un centro remoto (el cual tendrá Matlab/Simulink/RTW), se evalúe primero sobre un robot virtual (PC ejecutando ROS/Gazebo) y después, una vez realizados los ajustes necesarios y mínimos posibles, sobre el robot real, aportando con ello una reducción en el tiempo, los riesgos y costes asociados a la experimentación real.

El enlace centro remoto - robot (real y virtual) está basado en aplicaciones cliente-servidor mediante sockets. El enlace entre ROS y Gazebo se desarrolla con topics, buses específicos de ROS.

A modo de ejemplo se recurrirá a un robot P3DX que realiza el seguimiento de una trayectoria no lineal en un entorno interior de la escuela politécnica superior de Álcala. En este trabajo se detallará la creación de un entorno cualquiera en la herramienta Gazebo, y como se dijo antes, se realizará como entorno de ejemplo un pasillo de la escuela politécnica superior, concretamente de la primera planta del sector Oeste.

También se detallará la creación de cualquier modelo robot en la herramienta Gazebo, y más concretamente se realizará el robot P3-DX con las dimensiones, peso y apariencia del robot real P3-DX que se encuentra en el laboratorio OL3 de la Escuela Politécnica superior de Álcala, perteneciente al Geintra.

La solución conseguida se extenderá al caso de múltiples robots, en ambos casos es fundamental la validación por emulación de la solución real de control aplicada a un entorno virtualizado.

También se propondrá un algoritmo tipo Mamdani, para evitación de obstáculos, de cualquier tipo de forma y tamaño, con ubicación conocida en el entorno de trabajo.

Capítulo 1: Introducción

Actualmente la implementación de los sistemas robóticos se hace cada vez más compleja debido a que las aplicaciones requieren un mayor grado de abstracción, innovación y conocimiento en consecuencia de cómo evoluciona el mundo de la robótica y las necesidades reales que se plantean.

Por tanto, los simuladores robóticos siguen desempeñando un rol importantísimo no solo en el ámbito académico (docencia e investigación), sino también en el ámbito profesional. Por ejemplo, tareas habituales en el ámbito de la robótica como el diseño de controladores, generación de trayectorias, evitación de obstáculos, cooperación entre unidades, localización y mapeado simultáneo (SLAM), etc, serían más complejas tanto de diseñar como de testear si no contáramos con simuladores, por lo que a lo largo de los últimos años se han desarrollado diferentes aplicaciones como Matlab/Simulink y ROS, que se utilizarán en este trabajo, y permiten testear de forma eficiente y rápida nuevos conceptos, estrategias y algoritmos que mejoren el desempeño de las tareas mencionadas.

En la siguiente figura podemos ver algunas tareas habituales en la robótica mencionadas:

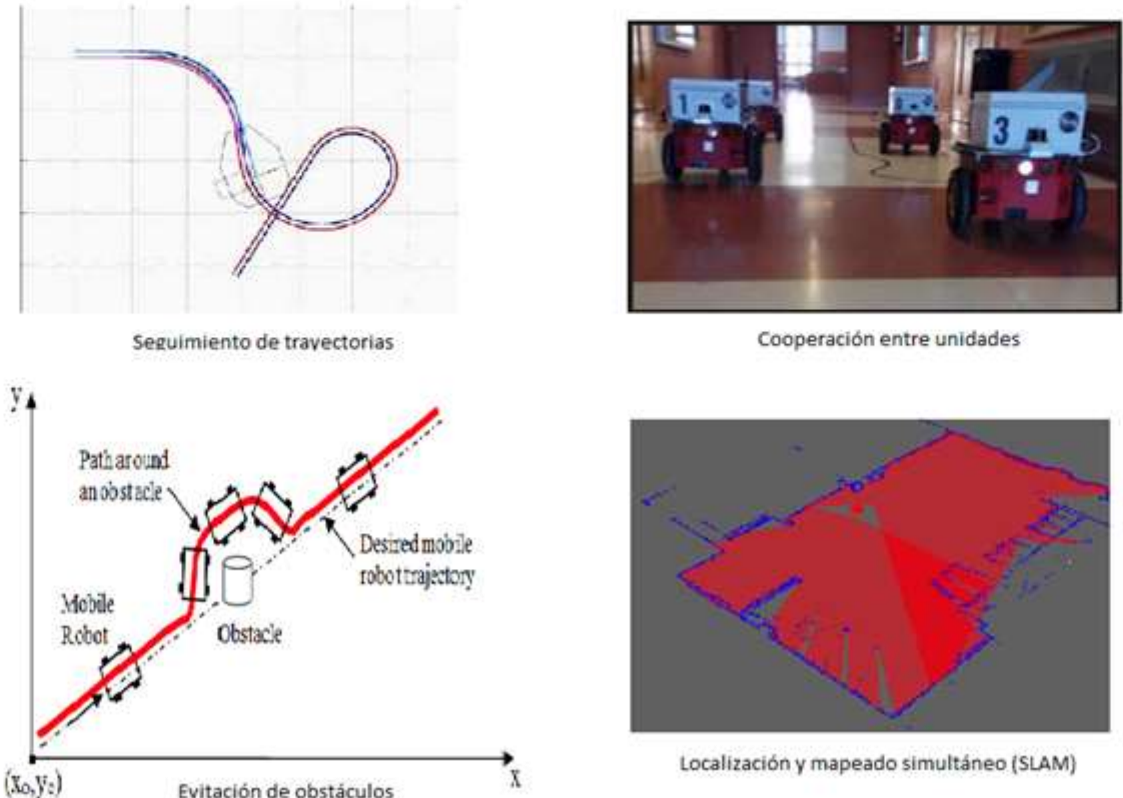
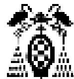


Ilustración 1.1: Tareas habituales en el ámbito de la robótica.



Estas tareas son transferidas al modelo físico (o robot real) una vez se hayan validado previamente en simulación a partir de modelos, tanto del robot como del entorno.

Sin herramientas de simulación es arriesgado y, en muchos casos, de coste elevado, la puesta a punto de las soluciones complejas que los diseñadores proponen como, cadenas cinemáticas de robots, la programación de sistemas robóticos, evitación de obstáculos con múltiples parámetros, y un largo etcétera.

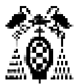
Como resultado de los esfuerzos de la investigación académica e industrial en eliminar estas barreras se han desarrollado múltiples plataformas completas donde se pueden emular e implementar cualquier algoritmo diseñado. Son múltiples los desafíos que esto conlleva, como la adaptación de sensores, actuadores, comunicación, inercias…etc, con el objetivo de que las herramientas resultantes sean robustas, modulares y confiables.

Algunas de estas plataformas comentadas con dedicación a cierta clase de tarea o robot son: MARTe centrado en el desarrollo de aplicaciones colaborativas en tiempo real [1]; SAMON para aplicaciones enfocadas a AUV (autonomus underwater vehicles) [2]; USARSim para el estudio de tareas colaborativas en USAR (urban search and recue) [3], entre otras. Podemos ver como luce el interfaz de algunas plataformas mencionadas en la siguiente figura:

Ilustración 1.2: Interfaz de las plataformas SAMON y USARsim.

De entre las soluciones existentes, en este trabajo fin de máster se utiliza Matlab/Simulink y ROS junto con su aplicación asociada Gazebo.

En el grupo Geintra de la UAH se tiene experiencia en el siguiente proceso de implementación de controladores para robots P3-DX: modelado del robot real, diseño de la solución de control, generación de código ejecutable en tiempo real (tanto en el centro remoto como en los robots), implementación del lazo de control utilizando la red wifi como enlace entre el centro remoto y la unidad móvil controlada. En este TFM se aprovecha esta experiencia y se completa con la idea de disponer de un banco de pruebas virtual para testear los algoritmos de control antes de su implementación en robots y



entornos reales. Para ello se recurre a las utilidades proporcionadas por el entorno ROS. En definitiva, se pretende trabajar con herramientas de manera totalmente independiente, e incluso simultáneamente, a fin de evaluar la cadena: modelado-diseño-simulación-implementación-validación.

El centro remoto es un ordenador donde, además del diseño y simulación de la solución de control (Matlab/Simulink), se implementa el algoritmo de control, comunicándose bien con el robot virtual (PC con ROS/Gazebo) y/o con el robot real.

En la siguiente figura se muestra la idea básica a desarrollar en este trabajo:

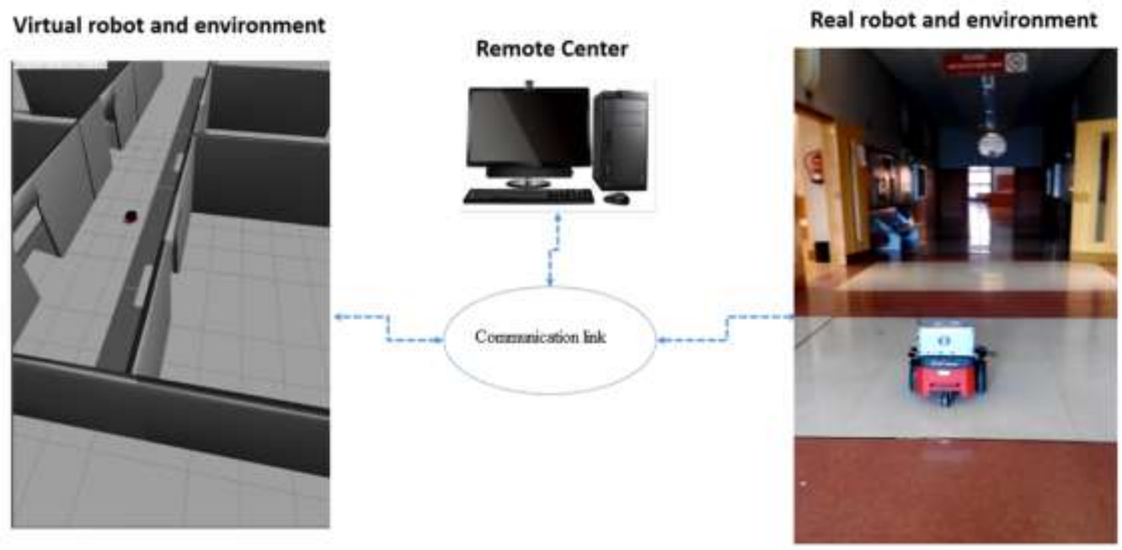


Ilustración 1.3: Robot virtual y robot real controlados desde un centro remoto con el mismo algoritmo de control.

1.1. Objetivos

El objetivo principal de este TFM es disponer de un demostrador que permita evaluar en un escenario virtual, diferentes propuestas de control remoto de un conjunto de unidades robóticas, de forma independiente o en conjunto, con y sin obstáculos en la trayectoria prevista. El algoritmo de control se diseña con la herramienta Matlab/Simulink a partir del modelo en el espacio de estados del robot. Para la validación del algoritmo (ejecutable en tiempo real) se pretende contar con una fase de emulación en la que la planta controlada es un robot virtual en un entorno virtual. La herramienta elegida para esta tarea es ROS/Gazebo.

El PC ejecutando ROS se conecta en red ethernet con el centro remoto que ejecuta la solución de control obtenida con la toolbox Real-Time Workshop de Matlab. Como prototipo de robot se elige el Pioneer P3-DX.

Se ha de desarrollar la integración ROS-Matlab para que la solución de control actúe en tiempo real sobre las unidades robóticas virtualizadas con ROS en el correspondiente PC.

El mismo algoritmo de control del centro remoto, una vez validado, se aplica sobre unidades P3-DX reales

En definitiva, los objetivos concretos planteados para este TFM son:

- Integración de aplicaciones ROS-Matlab para emulación de aplicaciones remotas en tiempo real.

- Virtualización tanto del entorno como de la unidad P3-DX, de forma que reciba los comandos de velocidad lineal y angular desde el centro remoto y devuelva a este los valores de velocidades proporcionados por la odometría (virtual).

- Evaluación de algoritmos de control periódico sobre un conjunto de unidades robóticas virtualizadas.

- Resolver la evitación de obstáculos.

- Comparación de resultados tras aplicar los mismos algoritmos de control a un conjunto de robots reales.

En definitiva, la motivación de este trabajo consiste en describir y mostrar resultados de aplicación de la solución adoptada, construyendo los interfaces necesarios que permitan la integración de herramientas software para resolver problemas de control remoto de robots reales o virtuales, de forma que la única variante sea el nodo de la red inalámbrica a la que se conecta el centro remoto de control.

Capítulo 2: Herramientas utilizadas

En este apartado se van a describir las herramientas que se han utilizado para la realización de este trabajo fin de máster. Se detallará la arquitectura y funcionamiento de la herramienta Robot Operating System (ROS) así como de la herramienta ligada Gazebo. También se expondrá la herramienta RVIZ que es un simulador 3D para visualizar sensores, robots, etc; aunque no ha sido utilizada significativamente en este trabajo, el autor cree que es importante mencionarla debido a su gran capacidad de simulación y posible ayuda para el lector.

Por otro lado, se expondrá igualmente la herramienta Matlab/Simulink (esta en menor medida por ser habitual en la formación en ingenierías), su toolbox Real Time Workshop la cual permite generar código C y ejecutables para diferentes targets así como el sistema operativo de tiempo real Linux/RTAI donde se correrá la aplicación. RTAI es una modificación del Kernel original de Linux para dar más prioridad a las interrupciones de periféricos que al propio Linux, que lo trata como una tarea de tiempo real de menor prioridad.

2.1. ROS, Robot Operating System

ROS (en inglés Robot Operating System, o en español Sistema Operativo Robótico) es un entorno de desarrollo de software robótico que proporciona al usuario o diseñador una serie de herramientas de visualización, librerías, monitorización y análisis, siendo todas ellas en código abierto (Open-Source), lo que hace que tenga infinidad de paquetes de aplicaciones desarrolladas por múltiples investigadores, empresas y amantes de la robótica para que una inmensa comunidad las puedan usar o, lo que es mejor, modificarlas y crear nuevas y mejoradas propuestas.

Este framework se desarrolló originalmente en 2007 bajo el nombre de Switchyard por el Laboratorio de Inteligencia Artificial de Stanford para dar soporte al proyecto del Robot con Inteligencia Artificial de Stanford (STAIR2), pero es en 2008 cuando se desarrolla principalmente en el instituto de investigación Willow Garage. A partir de entonces se mantiene por Open Source Robotis Foundation bajo licencia BSD, la cual permite su uso para desarrollos tanto comerciales como no comerciales, reduciendo la inversión y fomentando el uso tanto por desarrolladores independientes, como empresas e investigadores, dándole un enfoque científico pero aplicado.

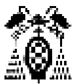


Ilustración 2.1: Logo de Robot Operating System (ROS).

Dado que es Open-Source, ROS se convierte entonces en un sistema realimentado continuamente, que de acuerdo a los datos que nos proporcionan crece de manera exponencial en prestaciones y popularidad, ya que en cinco años ha pasado a tener más de 3000 paquetes de aplicaciones creados por la comunidad así como de tener más de 150 repositorios repartidos por todo el mundo. En las siguientes figuras podemos ver de forma gráfica como ha ido evolucionando ROS según los últimos datos oficiales que nos proporciona su propia web [4]:

Ilustración 2.2: Estadísticas de repositorios de ROS.

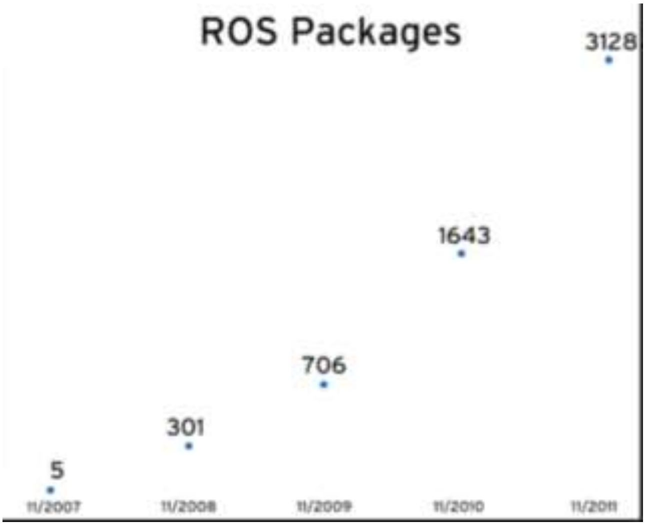


Ilustración 2.3: Últimos datos oficiales de los paquetes alojados en los repositorios de ROS.

También cabe destacar como ha aumentado las posibles plataformas de robots donde ROS funciona, siendo en 2012 de 30 plataformas y llegando a 106 plataformas actualmente en 2016. Se pueden ver las plataformas soportadas por ROS en [5].

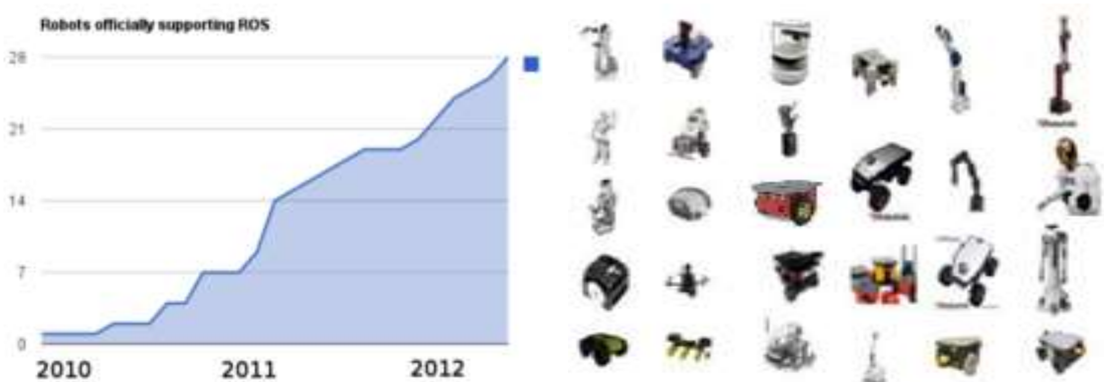


Ilustración 2.4: Plataformas robóticas soportadas por ROS.

Expuesta una pequeña introducción de ROS y ver que es una plataforma robótica en constante crecimiento según datos oficiales de uso, vamos a describir brevemente la estructura de este framework.

En cuanto a la estructura, Robot Operating System permite una implementación distribuida de elementos tan sencillos o complejos como el desarrollador desee. Los proyectos se crean en paquetes los cuales contienen los códigos necesarios para desarrollar aplicaciones robóticas según las necesidades de cada usuario. Debido a que es una plataforma de código libre (Open-Source), ROS acopla componentes externos de otras plataformas también Open-Souce como Gazebo, OpenCV, Player… en una red completamente funcional, y por tanto reduciendo así los problemas de interacción de herramientas a los desarrolladores, permitiéndoles concentrarse en la verdadera innovación. Por tanto, se acoplan componentes de otras plataformas siendo ROS el núcleo central de comunicación como se puede ver en la siguiente figura:

Ilustración 2.5: Comunicación entre plataformas con ROS.

Entre los objetivos de este trabajo fin de máster está el comunicar Matlab/Simulink con la plataforma ROS, la cual no tiene aún ningún proyecto que lo implemente. La propuesta es utilizar un socket pero sin ser ROS el núcleo de la comunicación, teniendo bien diferenciados: el centro remoto o centro de decisiones (PC con Matlab), el robot real y entorno virtual (PC con ROS).

Siguiendo con la estructura de ROS, cada módulo o plataforma posee total autonomía, esto quiere decir que todo se hace mediante mensajes enviados de acuerdo a un protocolo que en este caso es el protocolo XML-RCP, el cual es un sistema basado en la publicación y registro. Este protocolo permite que ROS sea una plataforma multilenguaje de programación, siendo principalmente C++ y Python, pero actualmente se ha extendido a Octave, LISP y Java. No obstante, ROS va más allá haciendo uso del protocolo TCP/IP para generar un esquema cliente servidor, cuyo principal servidor como se dijo anteriormente es el núcleo de ROS, donde cada nodo tiene una dirección y existe la posibilidad de ofrecer y solicitar servicios como veremos en el apartado siguiente 2.1.1 sobre los conceptos básicos de ROS.

Por tanto se podría describir que es la interacción de múltiples hilos de comportamiento dinámico y que ROS los comunica entre sí siendo el núcleo central. Aunque no se trata de un sistema operativo pues no maneja asignación de recursos como memoria o utilización de CPU, sí proporciona servicios estándares como abstracción del hardware, comunicación a través de mensajes entre procesos, mantenimiento de paquetes, control de dispositivos de bajo nivel e implementación de funcionalidad que son propios de los sistemas operativos. En la siguiente imagen se puede ver las principales diferencias entre un sistema operativo y ROS:

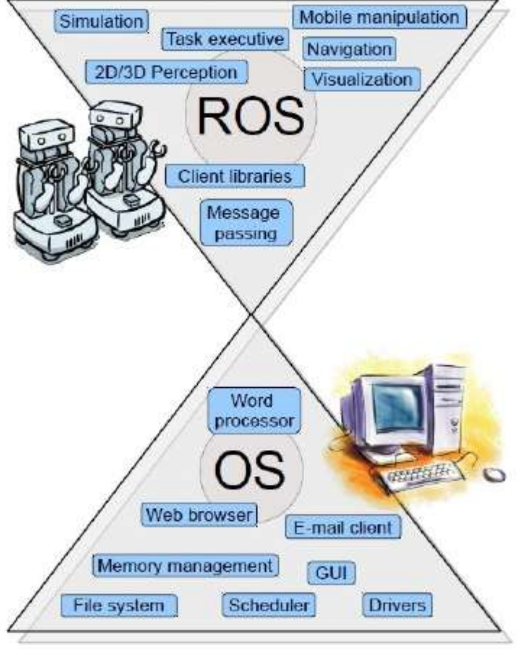


Ilustración 2.6: Diferencias entre ROS y un sistema operativo.

Descrita la estructura de ROS, se va a explicar brevemente como se realiza la comunicación de estos núcleos que hemos comentado anteriormente e introduciremos algunos conceptos del framework para luego después describirlos con detalle en la siguiente sección.

Existe un nodo "maestro" que actúa como un servicio de nombres en la computación gráfica de Ros, esto es, la red de procesos de ROS donde se están procesando los datos y almacenando temas y servicios de información de registro para cada uno de los nodos en ejecución. Los nodos se comunican con este nodo "maestro" reportando su información de registro. A la vez que los nodos se comunican con el "maestro" también se pueden comunicar con otros nodos de dos formas diferentes: de forma síncrona se comunican mediante "servicios", y por el contrario de forma asíncrona se comunican mediante "topics", por lo que los nodos se conectan a otros nodos directamente, y el nodo "maestro" solo proporciona información de búsqueda como si de un servidor DNS se tratara, controlando todos los servicios, mensajes, nodos y topics.

En la siguiente figura se puede ver gráficamente de una forma resumida la comunicación en el entorno ROS:

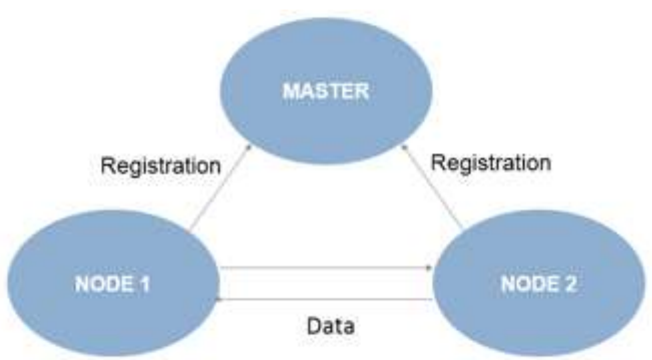


Ilustración 2.7: Comunicación interna de ROS.

Y en la siguiente figura la comunicación entre dos nodos mediante las dos formas distintas descritas, de forma síncrona mediante servicios y de forma asíncrona mediante topics:

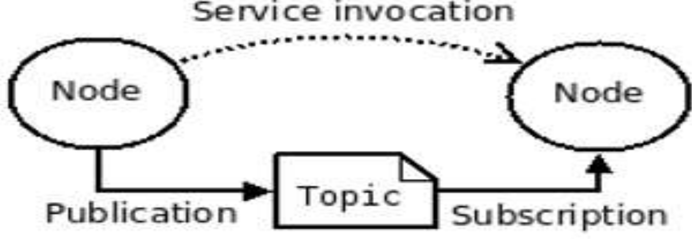


Ilustración 2.8: Alternativas de comunicación entre nodos.

De forma asíncrona el nodo de la izquierda publica mensajes en el topic constantemente, mientras que el nodo de la derecha se subscribe a este topic para obtener los mensajes publicados. De forma síncrona el nodo de la derecha se comunica mediante servicios con el nodo de la izquierda invocando un mensaje y obteniendo una repuesta de este último, o lo que es lo mismo, se comunican mediante servicios.

A continuación se expondrá un ejemplo sencillo sobre cómo nos podríamos comunicar con un sensor láser para saber los datos que está proporcionando.

Imaginemos que queremos saber los datos que está transfiriendo un láser Hokuyo. Para empezar podemos iniciar un nodo de nombre "nodo_hokuyo", el cual habla con el láser y publica en un topic (donde se transmiten los datos) el mensaje "sensor_msgs/LaserScan" donde están los datos recogidos por el láser. Para poder obtener esos datos que están en un topic, lanzaríamos otro nodo de nombre "subscripcion_hokuyo" el cual se subscribe al topic o, lo que es lo mismo, obtiene los datos que contiene el topic, los cuales son los datos generados por el láser.

Cabe destacar que el nodo "nodo_hokuyo" está publicando constantemente los datos en el topic sin importar si alguien está subscrito a él, por tanto los nodos se pueden lanzar en cualquier orden y cuando el usuario quiera sin inducir a error. Se puede ver de forma gráfica el ejemplo en la siguiente figura:

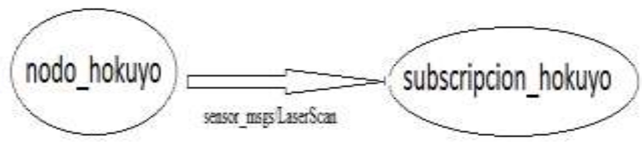


Ilustración 2.9: Ejemplo publicación-subscripción de un topic en ROS.

Con todo esto expuesto podemos decir que se ha elegido esta plataforma para realizar la simulación del algoritmo de control en un robot y entorno virtuales debido a que es un proyecto de código abierto que permite a toda la comunidad acceder a él sin necesidad de ningún coste, permite programar diversos drivers, abordar problemas de alto nivel, abstracción del hardware como se comentó, lo que hace que nos permita ahorrar en tiempo y en coste de infraestructura, centrándonos más en la investigación y diseño del algoritmo.

Pero todo no son ventajas, entre las deficiencias de ROS que deben ser mencionadas está la incompleta documentación de ROS debido a que se trata de un proyecto en continuo desarrollo y que la crea la comunidad. Además, integrar aplicaciones y usar todas las herramientas necesita de un periodo de aprendizaje bastante amplio. Por último, ROS no es multiplataforma pudiéndose instalar en Linux con soporte en Ubuntu y Debian y de forma experimental en OS X(Homebrew) y Gentoo.

A continuación se detallarán los conceptos básicos de ROS que deben saberse antes de poder utilizar la plataforma.

2.1.1 Conceptos básicos de ROS

Robot Operating System tiene tres niveles conceptuales totalmente diferenciados entre sí, que son:

- **Nivel de sistema de archivos:**

Este nivel es la organización de los diferentes archivos en el ordenador. Esta organización se divide principalmente en las siguientes unidades:

- Paquetes: Los paquetes son la unidad principal de la organización del software creado en ROS. Un único paquete puede contener procesos (nodos), bibliotecas dependientes de ROS, bases de datos, archivos de configuración, archivos de creación de entornos y robots virtuales… en definitiva cualquier archivo útil en ROS se almacena en un paquete. Por tanto, lo más importante que podemos crear y eliminar en ROS es un paquete. La estructura de un paquete dentro del ordenador es la siguiente:

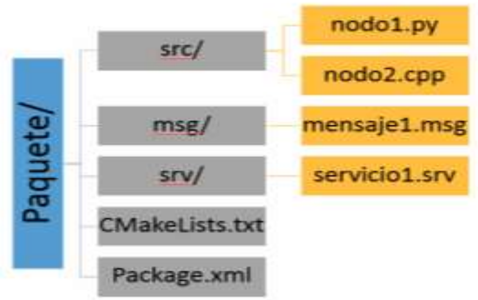


Ilustración 2.10: Estructura de un paquete de ROS.

- Metapaquetes: Estos son paquetes especializados que únicamente sirven para representar un grupo de paquetes relacionados entre sí.

- Repositorios: Estos son una colección de paquetes que comparten un sistema VCS común, esto es, un sistema de control de versiones.

- NVI: Son descripciones de los mensajes que definen las estructuras de datos para los mensajes enviados en ROS. Están almacenados en "my_package/msg/MyMessageType.msg".

- Pilas: Conjunto de paquetes que tienen como objetivo la organización de un proyecto o varios para llevar a cabo funciones de alto nivel.

- Manifiestos del paquete (Manifiest): Son archivos que pueden pertenecer a un paquete o a una pila. Describen información general sobre un paquete o pila específico, así como, una breve descripción de su funcionamiento, su tipo de licencia y las dependencias con otros paquetes. Estas dependencias son importantes porque generalmente los paquetes dependen de otros paquetes y estos deben estar instalados para que el conjunto funcione correctamente.

- **Nivel de computación gráfica:**

Este nivel describe los componentes que conforman la red de procesos de ROS que procesan los datos ejecutados mediante peer-to-peer (P2P). Este nivel se divide en los siguientes conceptos:

- Nodo Maestro: Este nodo es el principal en la comunicación. Proporciona la función de registrar todos los nodos que se están ejecutando y los existentes topics y servicios. Sin el nodo maestro, los nodos no serían capaces de subscribirse a topics o invocar servicios y por tanto no habría comunicación.

- Parameter Server: Los nodos utilizan este servidor para almacenar y recuperar parámetros en tiempo de ejecución. Este servidor pertenece al nodo maestro.

- Nodos: Los nodos se encargan de realizar la parte computacional del proyecto y son escritos utilizando una biblioteca específica de cliente como son roscpp (C++) o Rospy (Python). ROS está diseñado para ser modular de modo que múltiples nodos comprendan un único sistema de control. Por ejemplo en nuestro caso de aplicación con el P3-DX, un nodo se utiliza para controlar los datos del láser Hokuyo, otro nodo para controlar las ruedas del robot (velocidades), también podría haber otro nodo que lleve a cabo la localización del robot… y así todas las funcionalidades que se nos ocurran en nuestro sistema pueden ser llevadas a cabo mediante nodos.

- Topics: Como dijimos anteriormente, los nodos comparten información entre sí mandando y recibiendo mensajes. Estos mensajes son almacenados en los topics, por lo que si un nodo quiere mandar mensajes, se publicará a un topic, y si quiere recibir mensajes se subscribirá a un topic. Cabe destacar que puede haber varios nodos publicando y subscribiéndose a un mismo topic.

- Servicios: Proporcionan una comunicación síncrona entre nodos. Estos se componen por un par de mensajes, uno de request y otro de reply, o lo que es lo mismo, cuando un nodo quiere llamar a un servicio, tiene que llamarle y esperar la respuesta proveniente del otro nodo.

- Mensajes: Estos están compuestos de datos, los cuales pueden ser de diferentes tipos como por ejemplo, enteros, booleanos, arrays… Estos datos son los que se transmiten entre nodos.

- Bags: Los bags (o bolsas) son un formato especial que permite guardar y reproducir datos de los mensajes de ROS. Por tanto tienen una gran importancia a la hora de hacer pruebas y comprobar algoritmos.

- **Nivel de la comunidad:**

Este nivel proporciona recursos que permiten a la comunidad de usuarios intercambiar todo tipo de software y conocimiento. Estos recursos incluyen:

- Distribuciones: Son una recopilación de las pilas (o proyectos) que se pueden instalar fácilmente utilizando el comando *apt-get install*.

- Repositorios: La existencia de diferentes repositorios distribuidos por todo el mundo en internet facilita la obtención y difusión de todo tipo de proyectos, paquetes y documentación creada por la comunidad. Por ejemplo cabe destacar *github*, el cual nos proporciona diversos paquetes con múltiples opciones de descarga y modificación. Estos repositorios son equivalentes a los repositorios de Linux permitiéndonos que se pueden descargar directamente por línea de comandos o mediante el administrador de descargas como si fuera un mero repositorio de Linux.

- ROS Wiki: Esta es la principal fuente de documentación e información de todo lo que a ROS envuelve (tutoriales, información de paquetes, información de uso….). Cualquier persona puede crearse una cuenta en esta Wiki y aportar su propia documentación, correcciones, actualizaciones, escribir tutoriales…etc.

- ROS Answer: Se trata del foro de ROS por excelencia. Toda la comunidad puede realizar y responder preguntas sobre cualquier tema o proyecto relacionado con ROS.

- ROS Blog: Aquí se publicarán todo tipo de noticias e incluso actualizaciones regulares de ROS.

Como vemos en el nivel de la comunidad, es una plataforma que le da mucha importancia a que todos los usuarios estén conectados y puedan darse feedback entre sí teniendo una Wiki, un blog y un foro. Este es un de los aspectos relevantes por los que ROS sigue creciendo día a día gracias a la comunidad.

2.1.2 Herramientas/Comandos básicos de ROS.

Existen numerosos comandos que se pueden ejecutar desde el terminal de Linux para establecer acciones en ROS. En este apartado se nombran los comandos/herramientas más importantes para el autor de este trabajo fin de máster y los que todo usuario deberá aprender para poder manejar con soltura la plataforma:

- Roscore: Son un conjunto de nodos y programas vitales para un sistema basado en ROS. Se debe tener Roscore corriendo para que exista comunicación entre los nodos, servicios, parámetros, etc. Para lanzar esta herramienta basta con ejecutar "roscore" en el terminal de Linux.

- Roscreate-pkg: Con este comando el usuario puede crear paquetes y especificar las dependencias, si las tiene, con otros paquetes existentes. El paquete se crea con unos archivos comunes que son: manifest.xml, CMakeLists.txt, mainpage.dox y Makefile. A continuación se presenta la sintaxis para lanzar esta herramienta por línea de comandos:

```
$ roscreate-pkg pkgname depend1 depend2 depend3
```

- Rosnode: Esta herramienta de línea de comandos sirve para mostrar información de depuración acerca de los nodos, incluyendo publicaciones, suscripciones y conexiones. A su vez, contiene una biblioteca experimental para recuperar información de los nodos. En la siguiente imagen se pueden ver las funcionalidades de esta herramienta y su invocación por línea de comandos.

```
rosnode info    print information about node
rosnode kill    kill a running node
rosnode list    list active nodes
rosnode machine list nodes running on a particular machine or list machines
rosnode ping    test connectivity to node
rosnode cleanup purge registration information of unreachable nodes
```

- Rosrun; Este comando es el utilizado para ejecutar nodos en ROS. No hace falta darle la ruta completa de donde está el nodo en el ordenador, ROS es capaz de saber dónde está dándole tan sola el nombre del nodo que queremos ejecutar y el paquete al que corresponde tal como:

```
rosrun <package> <executable>
```

- Rostopic: Esta herramienta de línea de comandos sirve para mostrar información de depuración acerca de los topics de ROS, incluyendo editores, suscriptores, tasa de publicación, y mensajes de ROS. Esta herramienta tiene múltiples comandos que se pueden ver a continuación:

```
rostopic bw     display bandwidth used by topic
rostopic delay display delay for topic which has header
rostopic echo   print messages to screen
rostopic find   find topics by type
rostopic hz     display publishing rate of topic
rostopic info   print information about active topic
rostopic list   print information about active topics
rostopic pub    publish data to topic
rostopic type   print topic type
```

- Rosservice: Al igual que con los topics, existe una herramienta para los servicios. También tiene diferentes comandos que son los siguientes:

```
rosservice call call the service with the provided args
rosservice find find services by service type
rosservice info print information about service
rosservice list list active services
rosservice type print service type
rosservice uri  print service ROSRPC uri
```

- Rosmsg: Esta herramienta contiene dos tipos de comandos que son: rosmsg que sirve para mostrar información acerca de los tipos de mensajes, y rossrv que sirve para mostrar información acerca de los tipos de servicios.

- Roscd: Permite acceder a un paquete en concreto.

```
roscd <package-or-stack>[/subdir]
```

- Roslaunch: Como se dijo anteriormente, ROS permite ejecutar varios nodos a la vez y esta herramienta es la utilizada para ello. Lo que hace es ejecutar archivos con formato .launch donde se especifican que nodos se quieren lanzar a la vez junto con la configuración de parámetros de cada uno de ellos, facilitando al usuario el uso del sistema global de ROS. Para lanzar esta herramienta basta con:

```
$ roslaunch package_name file.launch
```

- Rosdep: Esta herramienta sirve para descargar e instalar dependencias.

- Rosgraph: Esta herramienta imprime información gráfica sobre todas las comunicaciones que hay a la hora de ejecutarla como son la comunicación entre nodos, topics… A continuación se muestra un ejemplo gráfico de lo que imprime rosgraph:

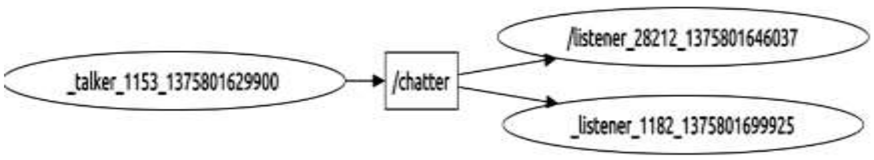


Ilustración 2.11: Gráfico de nodos y topics que proporciona Rosgraph.

Por último en este apartado vamos a mencionar la herramienta TF. En ROS, como en cualquier sistema robótico que se plantee, son muy importantes los sistemas de coordenadas. TF de Ros nos permite coordinar múltiples sistemas de referencias y mantener la relación entre ellos siguiendo una estructura de árbol. TF se distribuye de manera que la información acerca de la coordinación de todos los sistemas de referencia del sistema están disponibles para todos los nodos de la red de ROS. Además, nos permite recuperar transformaciones, puntos, vectores y otras entidades definidas en distintos sistemas de referencia.

Algunos comandos de la herramienta TF, que se pueden ejecutar en el terminal, son los siguientes:

- tf_monitor: Imprime información sobre el actual árbol de coordenadas por consola.
- tf_echo: Imprime información sobre la transformación relativa entre dos ejes de referencia.
- static_transform_publisher: Publica un nuevo eje de referencia a través de una transformación estática de un eje ya existente.

- view_frames: Genera un PDF con nuestro árbol de TF. A continuación se puede ver como es un árbol de transformadas generado a través de este comando:

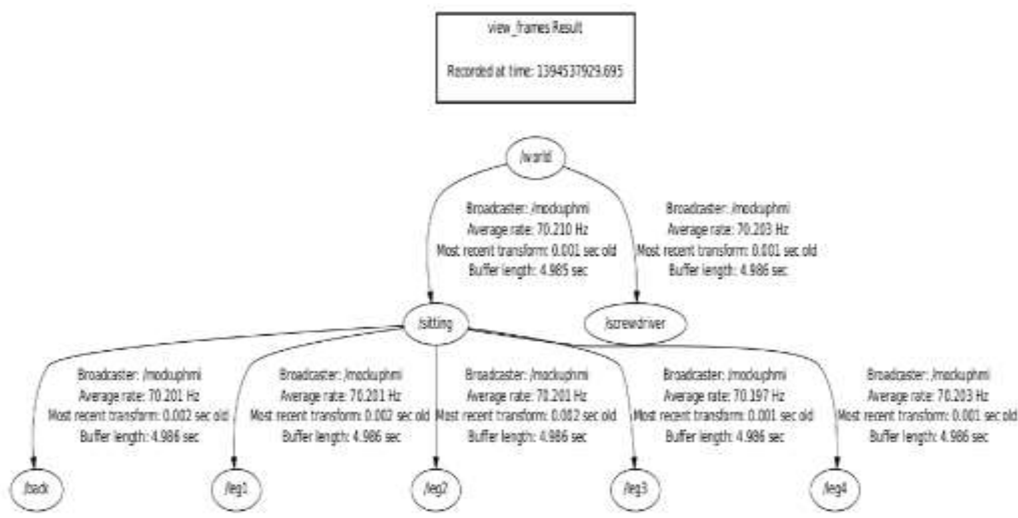


Ilustración 2.12: Árbol de transformadas generado con TF.

2.1.3 Distribuciones de ROS

Una distribución de ROS no es más que un conjunto de paquetes con control de versiones. Estas son similares a las distribuciones o versiones de Linux (Ubuntu por ejemplo, el cual es el que se utilizará en este trabajo fin de máster).

El objetivo que se intenta conseguir con las distribuciones es que los desarrolladores trabajen a partir de una base de código relativamente estable hasta que estén listos y realicen las modificaciones necesarias para ir a la siguiente distribución o versión.

En este trabajo fin de máster se ha decidido utilizar la versión lanzada en julio de 2014 denominada ROS Indigo Igloo debido a que era la versión más estable del momento, estando también ROS Jade, pero esta sin estar lo suficientemente estable en los paquetes utilizados en este trabajo. Por tanto utilizando la versión Indigo no ha sido necesario instalar dependencias externas para que todo funcione correctamente. Esta versión de ROS ha sido instalada sobre el sistema operativo Ubuntu 14.0 LTS.

Como trabajo futuro sería aconsejable trasladar la aplicación a la versión estable más reciente, pero cabe destacar que ROS nos proporciona soporte en la versión Indigo hasta abril de 2019.

En la Ilustración 2.13 se pueden ver las distintas distribuciones de ROS, así como su fecha de lanzamiento, icono y su fecha límite de soporte.

Una vez descrito ROS en su totalidad, y habiendo apuntado que en este trabajo se utiliza la distribución ROS Indigo, se va a proceder a explicar los simuladores que se han utilizado , principalmente Gazebo y por último RVIZ, este último no siendo relevante para la realización del mismo.

Distro	Release date	Poster	EOL date
ROS Kinetic Kame (Recommended)	May 23rd, 2016		May, 2021
ROS Jade Turtle	May 23rd, 2015		May, 2017
ROS Indigo Igloo	July 22nd, 2014		April, 2019 (Trusty EOL)
ROS Hydro Medusa	September 4th, 2013		May, 2015
ROS Groovy Galapagos	December 31, 2012		July, 2014
ROS Fuerte Turtle	April 23, 2012		–
ROS Electric Emys	August 30, 2011		–
ROS Diamondback	March 2, 2011		–
ROS C Turtle	August 2, 2010		–
ROS Box Turtle	March 2, 2010		–

Ilustración 2.13: Distribuciones de ROS.

2.2. Gazebo

Gazebo es un simulador multi-robot de entornos 3D que posibilita evaluar el comportamiento de un robot en un mundo virtual, ofreciendo la posibilidad de simular "físicas", es decir, permite simular fuerzas gravitatorias, aceleraciones, velocidades, etc. Sabiendo que tiene una relación nativa con ROS, es el indicado para utilizar en este TFM.

El desarrollo de Gazebo comenzó en el año 2002 en la Universidad del Sur de California. El concepto que se quería conseguir era un simulador de alta fidelidad para abordar las necesidades de simular robots en ambientes exteriores en diversas condiciones. Pero ha tenido tanto éxito que la mayoría de usuarios también lo utiliza en interiores como es el caso de este trabajo fin de máster. Fue en 2009 cuando entró en el proyecto John Hsu, ingeniero de investigación en Willow Garage, el cual hizo que ROS y Gazebo fueran de la mano. En 2013 fue tanto el éxito de Gazebo y ROS que la Open Source Robotics Foundation (OSRF) los utilizó para su proyecto DARPA Robotics Challenge.

Ilustración 2.14: Logo de Gazebo.

Gazebo ofrece un interfaz excelente para desarrollar y probar los sistemas multirobot en ambientes tanto interiores como exteriores, de formas innovadoras e interesantes, y además, de forma rápida. Se caracteriza porque todos los modelos tienen "físicas", lo que permite una simulación que se acerca a la realidad todo lo que el diseñador considere.

Es una herramienta eficaz, escalable y al igual que ROS es Open-Source, por lo que tiene también gran potencial para abrir el campo de la investigación en la robótica a una comunidad cada vez más grande.

Existen numerosos robots famosos que se encuentran en Gazebo como los son el PR2, Care-O-bot, Turtlebot, P3-DX, etc. En la siguiente ilustración los podemos ver en el entorno Gazebo:

Ilustración 2.15: Robots famosos en entorno Gazebo.

Esta breve introducción justifica porqué se ha escogido Gazebo como simulador para este trabajo, pues se pretende simular un robot virtual y un entorno dinámico que se asemeje al entorno real. La posibilidad de simular físicas facilita la incorporación de paredes, pasillos objetos, robots, etc; así como aceleraciones, velocidades, inercias, masas… de cada robot.

Sin embargo hay que mencionar que Gazebo, al igual que ROS, no son herramientas intuitivas, por lo que el aprendizaje inicial es arduo, es decir, la curva de aprendizaje es algo costosa. También, al igual que ROS, es una herramienta que solo corre en el sistema operativo Linux, pero ya se está trabajando para que en un futuro corra bajo Windows.

Para extender el conocimiento en esta herramienta consultar la referencia [6].

2.2.1. Distribuciones de Gazebo

Al igual que ROS, Gazebo también tiene distribuciones específicas con el mismo objetivo de conseguir que los desarrolladores trabajen a partir de un código base, modelos y mundos relativamente estables hasta que estén listos y realicen las modificaciones necesarias para ir a la siguiente distribución o versión.

Dependiendo de la distribución de ROS que se vaya a usar, Gazebo recomienda instalar una versión u otra del mismo, dado que ROS/Gazebo van a la par, se lanzan distribuciones a la par y para un mismo fin. En este trabajo, con la versión Indigo de ROS, se recomienda la versión 2.2 de Gazebo.

A continuación se presentan las versiones que existen actualmente de Gazebo así como la versión que se lanzará en 2017, la cual contendrá importantes novedades también descritas:

Ilustración 2.16: Distribuciones de Gazebo.

2.2.2. Relación entre ROS y Gazebo

Gazebo está implementado en ROS mediante la utilización de paquetes. Al instalar cualquier versión de ROS se instala por defecto la versión de Gazebo recomendable. Lo que se instala es un paquete con todos los contenidos necesarios para hacer correr Gazebo. En el caso de las versiones de ROS Electric, Fuerte y Groove, el paquete se denomina "gazebo", pero es en las versiones Hydro, Indigo y Jade en las que se denomina "gazebo_ros_pkgs", que es el paquete utilizado en este trabajo. Este paquete también se puede instalar a través del terminal de Linux o descargándolo a través de la página oficial de Gazebo.

Como se dijo, Gazebo es el simulador por defecto utilizado en ROS aunque ambos sean proyectos separados. El paquete de Gazebo para ROS mencionado anteriormente contiene plugins que interactúan con cualquier objeto de la escena del simulador, y proporcionan métodos fáciles de comunicación con ROS, tales como los ya mencionados topics (Gazebo se subscribe a estos topics) y servicios. Gazebo es considerado como un nodo para ROS y permite de una manera muy sencilla lanzarlo con sistemas complejos y simples a través de unos ficheros denominados launchfiles, en los cuales se pueden lanzar múltiples nodos y uno de ellos puede ser Gazebo, haciendo así que se habrá sin más que ejecutar launchfiles. Estos plugins pueden ser creados como el usuario quiera para el fin que necesite, como por ejemplo, crear un plugin que se encargue de escuchar los mensajes que ROS está publicando de velocidades y que se encargue de aplicar estas velocidades junto con los comandos apropiados a los motores del robot virtual.

Por tanto a través de plug-ins se comunica Gazebo con cualquier objeto de la escena, y este, es visto como un nodo para ROS, lo que facilita la comunicación y la interacción entre ellos utilizando topics y servicios. A continuación se puede ver de forma gráfica la relación entre ROS y Gazebo.

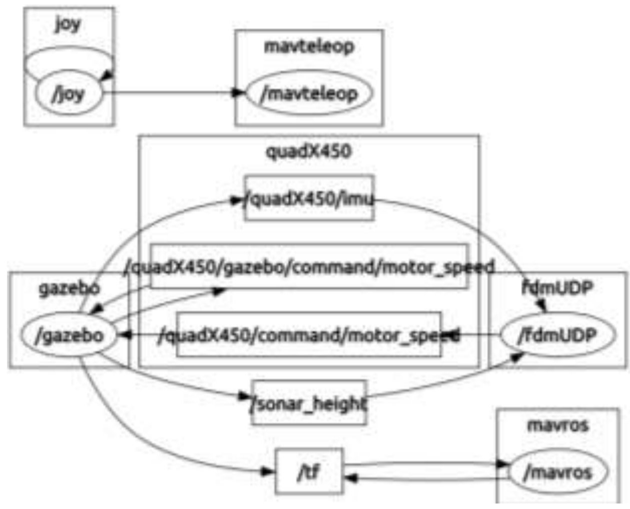


Ilustración 2.17: Gazebo como nodo de ROS.

En este sistema existen varios nodos los cuales están publicando y subscribiéndose en topics y uno de ellos es Gazebo.

2.2.3. Modelos en Gazebo

Como ya se ha comentado, se puede diseñar como se desee modelos tanto de robots como de entornos en Gazebo. Cuando se instala Gazebo existen una gran variedad de entornos y robots ya creados que se pueden importar directamente desde el simulador: oficinas de Willow Garage, modelos de casas, objetos como una lata de refresco, robots como el PR2… Sin embargo estos modelos importados directamente desde el simulador, en el servidor propio de gazebo, no son controlables y solo son utilizables como parte del entorno, es decir, que no cuentan con los topics y servicios necesarios para su manejo siendo tan solo un modelo en 3D. Sin embargo pueden descargarse desde el servidor y entonces si ser controlables. A continuación se muestra desde donde se pueden importar junto con una oficina de Willow Garage:

Ilustración 2.18: Importación de modelos en Gazebo.

Los modelos que explicaremos serán tanto los entornos como los robots.

2.2.3.1 Entornos

Los entornos en gazebo pueden ser creados de dos formas distintas, directamente desarrollando un archivo .sdf con una estructura específica que nos detalla Gazebo, o a través directamente desde un editor de entornos que Gazebo proporciona.

El editor de entornos es una manera fácil y rápida de crear entornos simples y también complejos pudiendo incluir paredes, ventanas, puertas, escaleras, e incluso añadir texturas y colores sin escribir código, únicamente con el ratón. Pudiendo incluso hasta importar planos en formato imagen que Gazebo se encargará de transformarlos en un entorno.

A continuación se presenta la interfaz del building editor de Gazebo:

Ilustración 2.19: Interfaz building editor de Gazebo.

Una vez que se haya creado el entorno con esta interfaz, se guarda y automáticamente Gazebo crea un archivo .sdf con formato XML que representa los componentes creados.

Por tanto también se puede escribir desde cero directamente el archivo .sdf incorporando todos los componentes del entorno.

Los componentes del entorno en el archivo .sdf tienen la siguiente estructura. Por ejemplo una pared:

```
<link name='Pared'>
  <pose>0 2 1 0 0 0</pose>
  <collision name='collision'>
    <geometry>
      <box>
        <size>2 2 2</size>
      </box>
    </geometry>
  </collision>
  <visual name='visual'>
    <geometry>
      <box>
        <size>2 2 2</size>
      </box>
    </geometry>
  </visual>
</link>
```

La "pose" es donde está el centro geométrico de la pared y el "size" el tamaño de la misma. Como vemos tienen dos apartados que son "Collision" (si queremos que el elemento tenga colisión) y "Visual" para indicar lo que se muestra en el simulador.

Los componentes en Gazebo se lanzan a través de archivos .world, los cuales cargan estos archivos .sdf, haciendo así que varios archivos .sdf compongan un entorno.

2.2.3.2. Robots

Para la creación de robots no existe un editor como tal, en este caso tenemos que crear un archivo en formato *Unified Robot Description Format (URDF)* con el cual trabaja ROS, con los componentes del robot al igual que hacíamos con el entorno, pero esta vez aquí es donde poder añadir las propiedades cinemáticas y dinámicas del robot, las uniones (joints) entre los distintos componentes, etc.

Para poder añadir los robots a Gazebo necesitamos que estén en formato SDF con la etiqueta <Gazebo> (explicada más adelante) y siguiendo unas pautas específicas, las cuales convierten automáticamente este formato URDF a SDF.

Para los componentes del robot se puede utilizar una herramienta de gazebo que permite añadir cubos, esferas… imitando a la base de los robots o a las ruedas, pero esta herramienta es muy limitada.

A continuación se puede ver gráficamente una estructura general de los componentes en Gazebo:

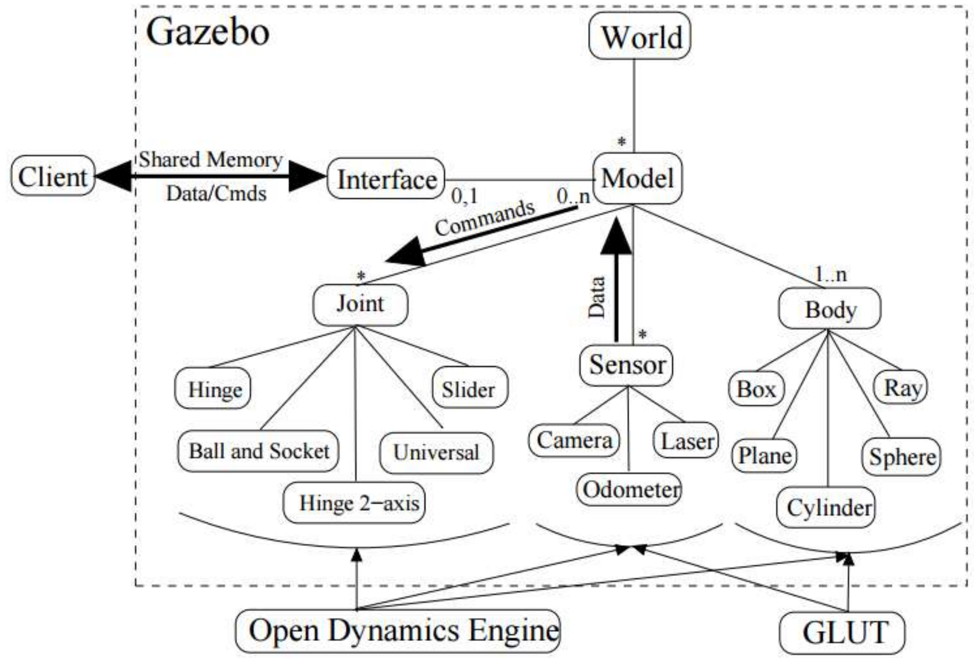


Ilustración 2.20: Estructura general de los componentes/modelos en Gazebo.

En el capítulo de desarrollo de la aplicación se detallará la creación tanto del entorno y robot utilizados para este trabajo.

2.3. RVIZ

RVIZ es un visualizador 3D que permite mostrar datos de diferentes sensores y los distintos estados de información de ROS. Al contrario que Gazebo, RVIZ pertenece a ROS y no es un proyecto aparte. RVIZ también es visto como un nodo en ROS.

Ilustración 2.21: Logo de RVIZ.

Mediante esta herramienta se pueden visualizar modelos virtuales robóticos y mostrar de manera "on-line" los datos recibidos por los diferentes sensores de robots reales, como la visualización de nube de puntos de sensores láser, mapas, imágenes obtenidas a partir de una cámara, etc.

A continuación se muestra el mapa visto por un láser, con la interfaz RVIZ

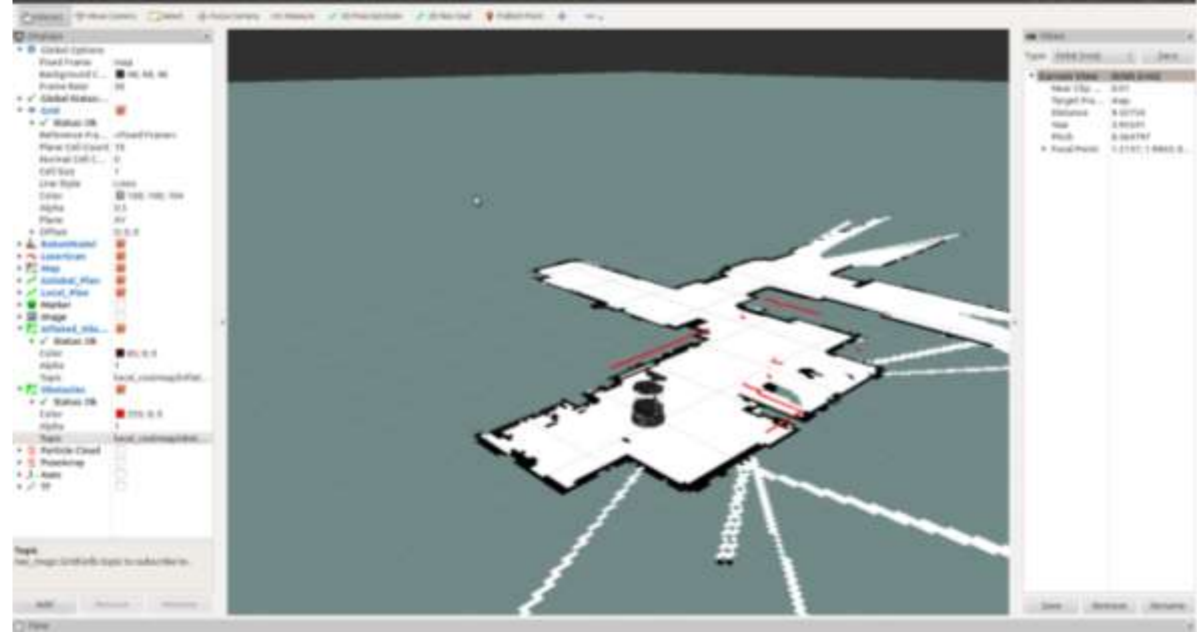


Ilustración 2.22: Interfaz RVIZ.

Para la utilización de esta herramienta, que se encuentra en ROS, basta con lanzar el comando "rosrun rviz rviz" a través del terminal. Al ser un nodo, también puede ser lanzado a través de un archivo .launch.

Existe una gran diferencia entre RVIZ y Gazebo. RVIZ únicamente se utiliza para mostrar y visualizar información de cualquier elemento de nuestro sistema, mientras que Gazebo se puede considerar como una copia virtual del entorno y robot reales. Por esto, en este trabajo se ha utilizado Gazebo en vez de RVIZ.

2.4. Matlab/Simulink

En este apartado se expondrá la herramienta Matlab junto con Simulink, elegidos para realizar los algoritmos de control a utilizar.

Matlab (abreviatura de MATrix LABoratory, laboratorio de matrices) es una herramienta de software matemático que ofrece un entorno de desarrollo integrado con un lenguaje de programación propio denominado lenguaje M.

Entre sus prestaciones básicas se hallan: la manipulación de matrices, la representación de datos y funciones, la implementación de algoritmos, la creación de interfaces de usuario y la comunicación con programas en otros lenguajes y con otros dispositivos hardware. En cuanto a las ventajas de Matlab para este trabajo están:

- Una gran database de algoritmos de control, comunicación y procesado de señales organizados en toolboxes (descritas en los siguientes párrafos).
- Permite testear algoritmos rápidamente sin recompilar, lo cual facilita el desarrollo y diseño del algoritmo.
- La habilidad de auto-generar código C usando MATLAB Coder para un gran subconjunto de funciones matemáticas que se pueden usar en otros entornos como los sistemas embebidos.

Es un software muy utilizado en universidades y centros de investigación y desarrollo. En los últimos años ha aumentado el número de prestaciones, como la de programar redes neuronales, visión artificial y un largo etcétera.

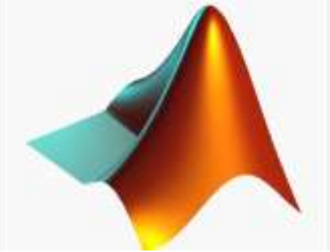


Ilustración 2.23: Logo de Matlab.

MATLAB dispone de una de las herramientas más importantes en el ámbito del control electrónico: Simulink. Esta es una plataforma de simulación multidominio que ofrece un entorno de programación gráfico de más alto nivel de abstracción de los fenómenos físicos involucrados en los sistemas que el lenguaje M de Matlab. Está basado en bloques que realizan acciones determinadas, teniendo así una amplia librería de bloques.

Simulink es muy utilizado en los campos de procesamiento digital de señales (DSP), ingeniería biomédica, telecomunicaciones, y sobretodo en el ámbito de la robótica e ingeniería de control.

En la siguiente ilustración se puede ver un diagrama de bloques de Simulink en el que se simula un controlador sobre una función de transferencia (planta). Además permite simular tanto en tiempo continuo como en tiempo discreto.

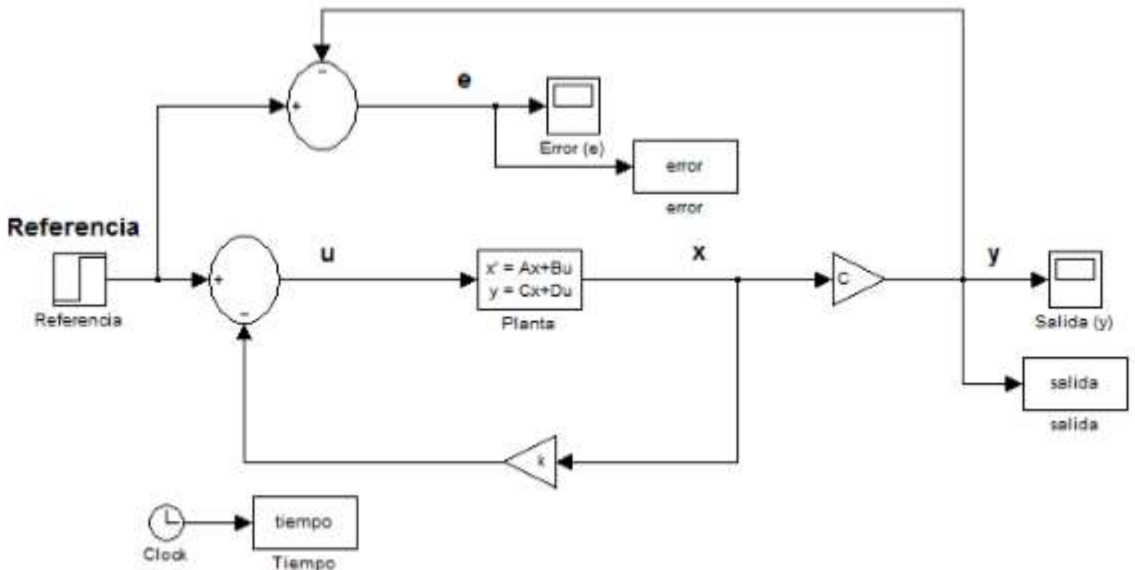


Ilustración 2.24: Diagrama de bloques en Simulink.

Matlab/Simulink se centra en el problema de control, sin considerar el cumplimiento de plazos temporales ni políticas de planificación de tiempo real de la CPU.

Además, Matlab cuenta con toolboxes que amplían las capacidades de la herramienta que son librerías específicas. Por ejemplo: "Fuzzy Logic" que proporciona funciones, aplicaciones y bloques de Simulink para el análisis, diseño y sistemas basados en lógica difusa. También, en el ámbito de la robótica, es muy útil "Robotic Toolbox" de Peter Corke.

Aparte de las toolbox de robótica y de control que se usan en este trabajo, es importante dedicar el siguiente apartado 2.4.3 a la toolbox Real Time Wokshop.

2.4.1. RTW: Real Time Workshop

Desde el punto de visto del autor de este TFM, una de las toolboxes de mayor utilidad para el ingeniero de control en general, y de sistemas robóticos en particular, es Real Time Workshop (RTW), la cual permite, a partir del diagrama de bloques Simulink de una aplicación, generar código para su ejecución en tiempo real sobre una determinada plataforma electrónica (target).

El código C que se genera para sistemas embebidos abarca un conjunto más reducido de bloques de Simulink, pero es un código claro, eficiente y compacto.

Por tanto, utilizaremos esta toolbox para generar código C y por ende un ejecutable, partiendo del diagrama de bloques de Simulink con la solución de control remoto, y este ejecutable se aplicará tanto al robot real como al robot virtual.

2.5. LINUX/RTAI

Linux es un sistema operativo de código abierto desarrollado por Linus Torvalds. Su desarrollo es uno de los ejemplos más prominentes de software libre, todo su código fuente puede ser utilizado, modificado y redistribuido libremente por cualquier usuario bajo los términos de la licencia Pública General de GNU, y otra serie de licencias libres.

Ilustración 2.25: Logo de Linux.

La gran mayoría de usuarios piensa que Linux es el sistema operativo en sí, pero cuando hablamos de él realmente nos referimos al kernel del sistema, el cual se encarga principalmente de gestionar el hardware (periféricos, memoria y procesador), las tareas y la comunicación existente entre estos. Al sistema operativo en su totalidad (Kernel y aplicaciones) se le denomina GNU/Linux.

A las variantes de esta unión de aplicaciones se las denomina distribuciones o versiones, y tienen como objetico ofrecer ediciones que satisfagan las necesidades de un determinado grupo de usuarios. Algunas son Ubuntu, Debian, Kubuntu, Lubuntu…

Dado que Linux es un Kernel de código abierto, puede haber modificaciones en el mismo como es el caso de RTAI (Real Time Applicatioin Interface).

RTAI es una modificación del Kernel original de Linux para tratarlo como una tarea de tiempo real de menor prioridad haciendo que el kernel de Linux se ejecute solo cuando no hay ninguna tarea de mayor prioridad ejecutándose. Además proporciona una amplia selección de mecanismos de comunicación entre procesos y otros servicios de tiempo real.

Esta modificación del kernel maneja en primer lugar las interrupciones de periféricos y se atienden las posibles acciones de tiempo real que hayan sido lanzadas por la interrupción. Además, los desarrolladores de RTAI elaboraron drivers que permiten la comunicación en tiempo real con puertos serie y Ethernet.

Además RTAI se complementa de forma ideal con la toolbox RTW de Matlab mencionada anteriormente ya que los ficheros proporcionados por RTAI (c, .tlc, .tfm) se integran con la herramienta RTW que junto con la generación de código, permiten crear aplicaciones de tiempo real para ser ejecutadas en robot reales. Cabe destacar que se han realizado varios estudios [7] [8] [9] que demuestran la robustez y fiabilidad de los sistemas de tiempo real diseñados mediante esta metodología.

Esta aplicación se instalará, completando la de Linux, en todos los elementos (robots y centro remoto) que han de soportar ejecución de algoritmos de control en tiempo real. Se ha optado por el parche RTAI 3.8.1 corriendo en la distribución GNU/Linux Ubuntu 10.04 LTS, ya que de esta manera podemos asegurar el correcto funcionamiento del sistema.

Sin embargo, para el otro PC (el cual simula el robot virtual) donde está instalado ROS/Gazebo, se instalará la distribución GNU/Linux Ubuntu 14.04 LTS que es la recomendada para la utilización de ROS Indigo, el cual es el que utilizaremos.

2.6. Resumen de las herramientas instaladas en cada ordenador

Una vez descritas las herramientas que se utilizaran para la realización de este TFM, es conveniente dejar bien claro en qué ordenador se instalarán y cuál será el cometido de cada ordenador.

Como ya se comentó en el capítulo 1, tendremos dos ordenadores que actuarán uno como controlador remoto y otro como robot virtual, comunicados ambos mediante un socket. En la siguiente ilustración se pueden observar las herramientas instaladas necesarias en cada uno.

Ilustración 2.26: Herramientas instaladas en cada ordenador.

Capítulo 3: Desarrollo

El planteamiento principal de este trabajo consiste en describir y mostrar resultados de aplicación de la solución adoptada por el autor construyendo los interfaces necesarios que permiten la integración de herramientas software para resolver problemas de control remoto de robots, de forma que la única variante sea el nodo de la red inalámbrica a la que se conecta el centro remoto (controlador).

En el diseño de sistemas de control, es sumamente conocido el ciclo de trabajo modelado-diseño-simulación-validación tanto para sistemas lineales como para sistemas no lineales. En este caso el sistema bajo estudio (robot P3-DX) se considera que es complejo por lo que la etapa de validación en estos casos se suele llevar a cabo en dos etapas.

- En la primera etapa (emulación) se aplica la solución real de control, ejecutada en el centro remoto, a un PC (ROS/Gazebo) que virtualiza el comportamiento dinámico y cinemático del robot: “robot virtual”.

- La segunda y última etapa (implementación), a partir de los resultados obtenidos en la primera y haciendo los ajustes apropiados en la solución de control, se valida finalmente en el robot real: P3-DX.

En la siguiente figura se puede ver de forma gráfica la relación entre estas dos etapas: emulación e implementación.

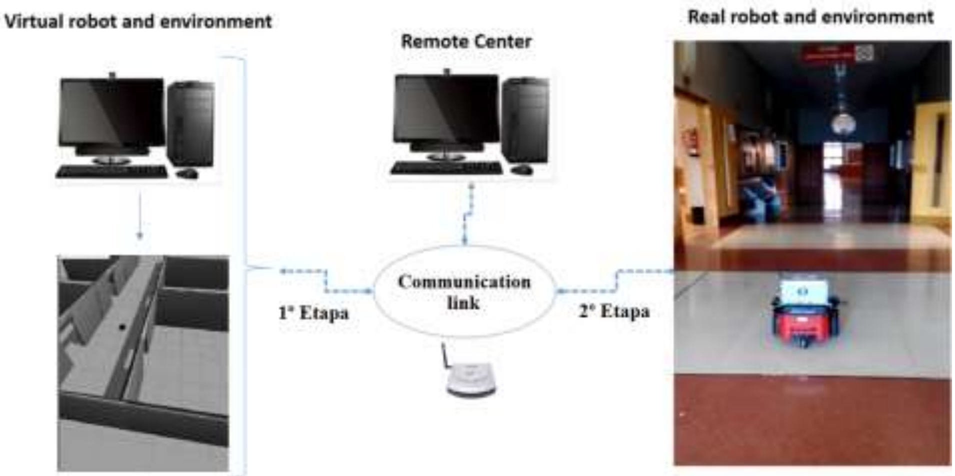


Ilustración 3.1: Etapas del trabajo con Centro remoto, robot real y robot virtual.

Por tanto las soluciones propuestas de control remoto en este trabajo se verificarán primero en robots virtuales, mediante la plataforma creada de ROS/Matlab/Simulink, y después en robots reales. Desde el punto de vista del controlador, la única variante será el nodo de la red inalámbrica a la que se conecta: o PC con el robot virtualizado o PC empotrado en el robot real.

Las comunicaciones entre PCs, en la primera y segunda etapa, se desarrollarán mediante aplicaciones cliente-servidor basadas en sockets. Además, se diseñará el enlace entre ROS y Gazebo mediante topics que son buses específicos de la herramienta ROS. Esto se describirá en los siguientes apartados.

A continuación se detalla el plan de trabajo seguido.

3.1. Plan de Trabajo

El plan de trabajo para la realización de este TFM ha sido el siguiente:

1. A partir del modelo identificado en variables de estado disponible del robot real P3DX, se ha diseñado la solución de control en el entorno Matlab/Simulink. Solución que incluye el seguimiento de una trayectoria no lineal de un único robot, extendiéndola también al caso de un conjunto de robots, así como la evitación de obstáculos que se encuentran en el camino de una trayectoria no lineal.

2. Los diagramas de bloques Simulink propuestos de la soluciones de control, se han convertido a código ejecutable en tiempo real (Linux CNC) mediante la aplicación RTW (Real Time Workshop).

3. Como paso previo a la aplicación de estos códigos sobre la plataforma robótica real P3-DX, se han validado en un robot virtual junto con un entorno dinámico mediante la herramienta Gazebo corriendo sobre la plataforma ROS en un PC independiente, conectado en red con el PC de control (centro remoto). Se han aprovechado las capacidades de Gazebo para someter al sistema bajo estudio a situaciones cambiantes (condiciones iniciales, obstáculos, diferentes destinos, etc), analizando y ajustando cuando ha sido necesario, los parámetros de los algoritmos diseñados en la herramienta Matlab/Simulink.

4. Por último, se ha reorganizado la solución de control en red, de forma que el enlace entre el centro remoto de control y el PC que emula la unidad robótica, se sustituye por un enlace entre el centro remoto y el PC embarcado en el propio robot real.

En la siguiente figura podemos ver los enlaces entre las herramientas utilizadas para la consecución de este proyecto:

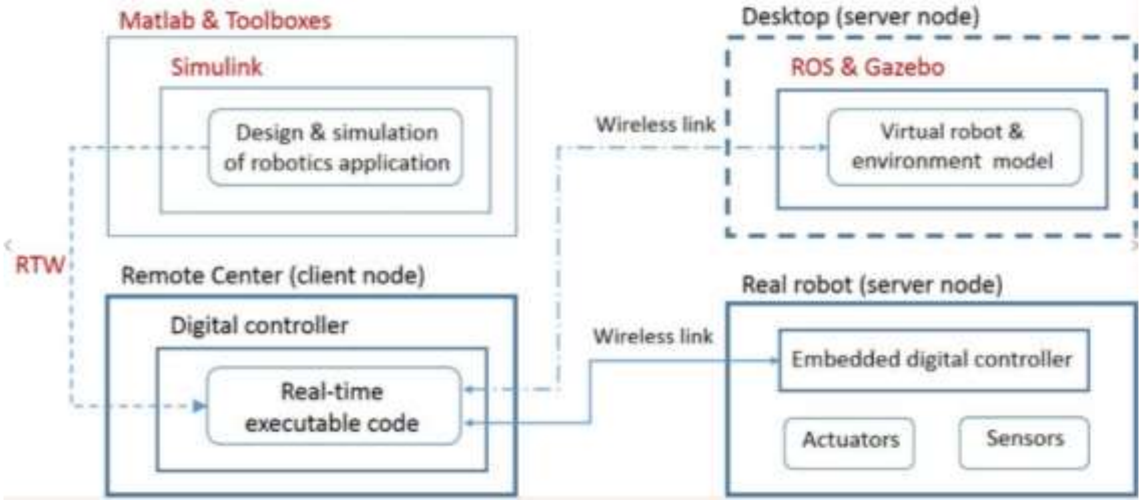


Ilustración 3.2: Enlaces entre las herramientas utilizadas.

La descripción del trabajo se estructurará en los siguientes apartados como sigue:

- Virtualización física 3D y funcional en ROS/Gazebo del robot P3DX utilizado por el grupo GEINTRA así como del entorno del pasillo Oeste de la primera planta de la EPS.

- Identificación de la planta P3DX en VVEE y descripción del algoritmo de control y generación de referencias a implementar en el centro remoto.

- Comunicación entre PC-control y robot virtual/real, comentando las diferencias entre robot real y robot virtual.

- Extensión a varios robots.

- Propuesta de un algoritmo de evitación de obstáculos diseñado en Matlab/Simulink y planteado para poder comunicarlo con ROS/Gazebo.

- Validación de resultados emulados con robot virtual e implementados con robot real.

3.2. Robot P3-DX

La plataforma utilizada es una modificación del robot Pioneer P3-DX. Las mejoras incorporadas están detalladas en [10], obteniéndose el robot de la figura.

Ilustración 3.3: Montaje interior de la caja y plataforma P3-DX utilizada.

Cabe destacar que para una simulación en ROS/Gazebo que se acerque a la realidad se tendrá que diseñar el elemento de la caja gris en 3D en el simulador, ya que esta versión particularizada no existe en el servidor ROS/Gazebo.

3.3. Virtualización física y funcional del robot P3DX así como del entorno.

En este apartado se detalla la virtualización física y funcional del robot P3DX utilizado por el grupo de investigación Geintra, así como del entorno en el que se aplicarán los algoritmos de control, el pasillo Oeste de la primera planta de la Escuela Politécnica Superior de Alcalá.

3.3.1. Programación de ROS/Gazebo en PC de robot virtual

Como ya se ha comentado, un PC con las herramientas ROS/Gazebo corriendo sobre el sistema operativo GNU/Linux Ubuntu 14.04 LTS es tratado como el robot virtual donde se testea la solución final adoptada hecha en Matlab/Simulink desde otro PC, el centro remoto.

Primero se explica cómo se ha realizado el modelo del robot virtual en la herramienta Gazebo con todas sus físicas.

3.3.1.1. Creación del Robot P3-DX en Gazebo

Se pretende diseñar la plataforma robótica P3-DX descrita en el apartado 3.2 y en la Ilustración 3.3 junto con sus físicas.

Primero se describe cómo se ha diseñado lo que es la parte visual del robot virtual.

Como se dijo en el apartado 2.2.3.2, en Gazebo existe una utilidad para añadir componentes del robot en forma de cajas, esferas… pero esto es muy limitado. Gazebo también permite exportar archivos con extensión .stl emulando componentes creados a través de programas de diseño como pueden ser AutoCad o incluso Blender.

Por tanto, lo primero que se debe de hacer a la hora de realizar un buen modelo del robot virtual son los componentes del mismo con plataformas de diseño. En este TFM se han utilizado como base para la realización del modelo P3DX el metapaquete existente en el repositorio denominado “ua_ros_p3dx”.

Dentro de este paquete existen los “meshes” que se han utilizado como base para la realización del TFM. Estos son los archivos .stl emulando en 3D cada componente. A su vez, para tener una apariencia más acorde a la realidad y a nuestro robot P3DX se ha utilizado la herramienta Blender.

A continuación se pueden ver estos componentes finales en formato 3D:

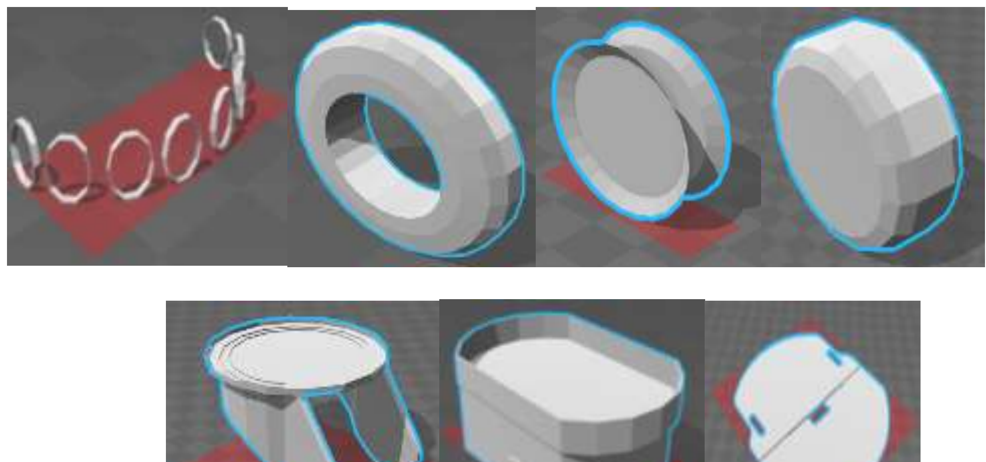


Ilustración 3.4: Componentes en 3D del robot virtual P3-DX.

De izquierda a derecha y de arriba abajo tenemos:

- Los anillos de 8 sensores sónar.
- Las ruedas delanteras que estarán ligadas al motor.
- Los tapacubos de las ruedas delanteras.
- La rueda trasera de giro libre.
- El soporte que une la rueda trasera con el chasis.
- El chasis del robot.
- La tapa superior del robot que va unida al chasis.

Quedando con estos elementos en Gazebo el siguiente modelo de P3-DX

Ilustración 3.5: Robot virtual inicial.

Las texturas y los colores más adelante se detalla como se pueden aplicar así como las uniones entre los componentes.

Como vemos en la Ilustración 3.5, a este robot le falta la caja gris que el proyecto COVE añadió al robot donde se encuentra diferentes elementos electrónicos: tarjeta VIA-EPIA , disco duro y ethernet-converter. Utilizando la herramienta Blender se ha creado una caja idéntica a la realidad la cual añadiremos al robot, quedándonos un robot virtual final como el de la siguiente imagen:

Ilustración 3.6: Robot virtual final utilizado.

En la caja se puede visualizar un "uno" para identificar el robot.

Una vez se tienen los componentes en 3D y en formato .stl hay que dotarles de física, color, texturas, así como de decir las uniones de cada uno de ellos para llegar al robot final de la anterior ilustración.

Para ello creamos un archivo con formato URDF como se expuso en el apartado 2.2.3.2 indicando los colores de los componentes, las texturas, las uniones y sus físicas.

A continuación se detalla el formato que se debe de seguir para dotar a los elementos de todas sus características para poder lograr el robot final a utilizar.

- **Añadiendo cinemática y dinámica a los elementos (<link>)**

El elemento <link> describe cada elemento con su dinámica y cinemática. La estructura de <link> es la siguiente:

```
<link name="my_link">
  <inertial>
    <origin xyz="0 0 0.5" rpy="0 0 0"/>
    <mass value="1"/>
     <inertia ixx="100"  ixy="0"  ixz="0" iyy="100" iyz="0" izz="100">
  </inertial>

  <visual>
    <origin xyz="0 0 0" rpy="0 0 0" />
    <geometry>
      <box size="1 1 1" />
    </geometry>
    <material name="Cyan">
      <color rgba="0 1.0 1.0 1.0"/>
    </material>
  </visual>

 <collision>
  <origin xyz="0 0 0" rpy="0 0 0"/>
    <geometry>
      <cylinder radius="1" length="0.5"/>
    </geometry>
  </collision>
</link>
```

Como se puede observar podemos añadir la masa del componente, su color, sus texturas…

El elemento <inertia> debe de aparecer en cada uno de los links si queremos que nuestro robot funcione en Gazebo como veremos más adelante en el apartado "Añadiendo los elementos a Gazebo" de este mismo capítulo.

Para asociar el archivo .stl a cada <link> basta con añadir la siguiente estructura en <visual>:

<mesh filename="package:// [ruta local del paquete donde se encuetra el .stl]/>

Una vez asociado basta con dirigirnos a él con el nombre que le hemos puesto link.

En la siguiente figura podemos ver de forma visual el elemento <link> con sus componentes principales:

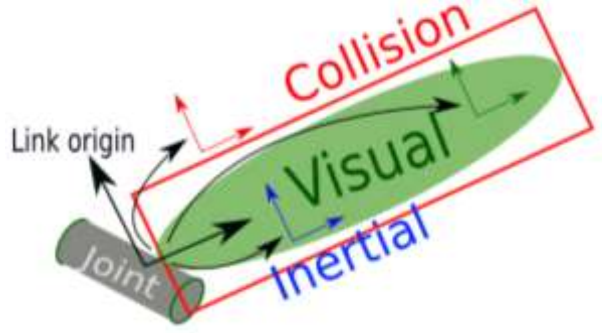


Ilustración 3.7: Elemento link con sus componentes principales.

Por ejemplo en nuestro caso, para el chasis tenemos el siguiente <link>:

```
<!-- chassis -->
        <link name="base_link">
                <inertial>
                        <mass value="7.8" />
                        <origin xyz="-0.05 0 0" />
                        <inertia ixx="1" ixy="0" ixz="0" iyy="1" iyz="0" izz="1" />
                </inertial>

                <visual name="base_visual">
                        <origin xyz="-0.045 0 0.148" rpy="0 0 0" />
                        <geometry name="pioneer_geom">
                                <mesh filename="package://p3dx_description/meshes/chassis.stl" />
                        </geometry>
                        <material name="ChassisRed">
                                <color rgba="0.851 0.0 0.0 1.0" />
                        </material>
                </visual>

                <collision>
                        <origin xyz="-0.045 0 0.145" rpy="0 0 0" />
                        <geometry>
                                <box size="0.35 0.25 0.14" />
                        </geometry>
                </collision>
        </link>
```

En este trabajo a cada componente se ha dotado de las características que más se acerca a la realidad utilizando el datasheet del robot P3DX proporcionado por MobileRobots [11].

- **Añadiendo uniones<join>**

El elemento <Joint> describe la cinemática y dinámica de una unión además de especificar los limites de seguridad de la union. La estructura de <join> la podemos ver en el siguiente ejemplo:

```
<joint name="my_joint" type="floating">
   <origin xyz="0 0 1" rpy="0 0 3.1416"/>
   <parent link="link1"/>
   <child link="link2"/>

   <calibration rising="0.0"/>
   <dynamics damping="0.0" friction="0.0"/>
   <limit effort="30" velocity="1.0" lower="-2.2" upper="0.7" />
   <safety_controller k_velocity="10" k_position="15" soft_lower_limi
t="-2.0" soft_upper_limit="0.5" />
</joint>
```

Como podemos ver, se pueden establecer las uniones así como la cinemática y dinámica de las mismas por ejemplo indicando la fricción o el límite de esfuerzo en Newtons.

En la siguiente figura podemos ver el elemento <joint> con sus componentes principales:

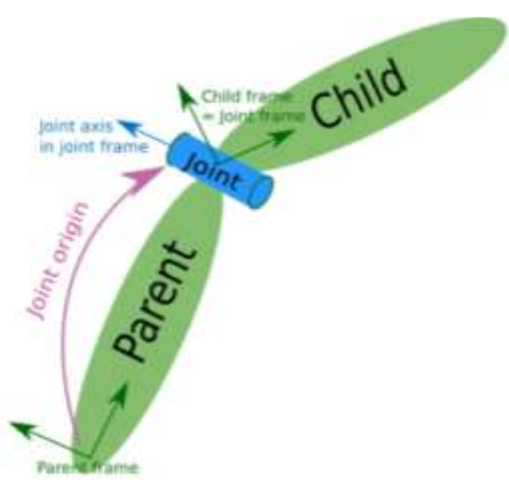


Ilustración 3.8: Elemento joint.

Por ejemplo en nuestro caso, para el soporte de la rueda trasera (quinta imagen de la Ilustración 3.4) tenemos el siguiente <joint>:

```
<joint name="base_swivel_joint" type="continuous">
        <origin xyz="-0.185 0 0.055" rpy="0 0 0" />
        <axis xyz="0 0 1" />
        <anchor xyz="0 0 0" />
        <limit effort="100" velocity="100" k_velocity="0" />
        <joint_properties damping="0.0" friction="0.0" />
        <parent link="base_link" />
        <child link="swivel" />
</joint>
```

Este soporte va unido al chásis, ya que el “parent link” es “base_link”, el cual está asociado al mesh del chásis como vimos en el apartado donde se describía el elemento <link>.

Así, se ha dotado de todas las uniones necesarias junto con sus cinemáticas y dinámicas al robot virtual utilizando como base el paquete ua_ros_p3dx como ya se dijo.

Hasta aquí se utiliza el formato URDF, pero para añadir nuestro robot a Gazebo lo necesitamos en formato SDF como se explica a continuación.

- **Añadiendo los elementos a Gazebo (<gazebo>)**

El elemento <gazebo> describe las características de los elementos en la simulación en Gazebo tales como la amortiguación, la fricción…

Como se ha comentado anteriormente, para poder añadir el robot a Gazebo no nos basta solo con el formato URDF ya que en este únicamente es posible indicar las propiedades cinemáticas y dinámicas del robot. Así mismo URDF no puede especificar la pose de robot dentro de un mundo y tampoco características que no son del robot tales como la luz del entorno, etc.

El formato SDF resuelve todos esos problemas. Es altamente escalable y facilita la adición y modificación de elementos. También permite definir los plugins para la comunicación ROS/Gazebo donde se pueden especificar los nombres de los topics.

Hay que tener en cuenta los siguientes pasos a seguir para hacer que un robot URDF funcione correctamente en Gazebo o, lo que es lo mismo, conseguir el formato SDF a partir del URDF, siendo el primer paso requerido:

- Añadir el elemento <inertia> a cada <link> propiamente especificado y configurado.

- Añadir el elemento <gazebo> a cada <link>, lo que hace es:
    - Convierte los colores a formato Gazebo.
    - Convierte los archivos stl a dae para permitir texturas.
    - Capaz de añadir plugins de sensores.

- Añadir el elemento <gazebo> a cada <joint> para
    - Especificar las dinamicas de amortiguación.
    - Añadir plugins de actuadores.

Una buena forma de trabajar a la hora de convertir un archivo URDF a SDF es cerciorándonos de que el modelo del robot en URDF funciona correctamente ejecutándolo en el simulador RVIZ, y si es así, transpolarlo con estas pautas a formato SDF.

Por ejemplo, la rueda trasera de giro libre quedaría de la siguiente forma para este trabajo con la etiqueta Gazebo:

```
<gazebo reference="center_wheel">
        <material>Gazebo/Black</material>
        <mu1>10.0</mu1>
        <mu2>10.0</mu2>
        <kp>1000000.0</kp>
        <kd>1.0</kd>
</gazebo>
```

Donde el argumento “reference” es el nombre del link que hayamos puesto anteriormente cuando describimos los links.

Para ver los distintos elementos que tiene gazebo (como lo son mu1 y mu2 para especificar los coeficientes de fricción), tanto para los links como para los joints se sugiere la referencia [12].

- **<u>Lanzando el robot al mundo Gazebo</u>**

Una vez tenemos los archivos que describen el robot, en este apartado se va a describir como lanzarlo al mundo Gazebo.

La realización de un archivo .launch permite lanzar el nodo Gazebo con el robot incluido en él tal que:

```
<launch>

        <!—Argumentos de configuración que se les puede pasar al archivo .launch -->
        <arg name="paused" default="false" />
        <arg name="use_sim_time" default="true" />
        <arg name="gui" default="true" />
        <arg name="headless" default="false" />
        <arg name="debug" default="false" />

        <!—Aquí lanzamos el mundo que queremos junto con sus características -->
        <include file="$(find gazebo_ros)/launch/empty_world.launch">
                <arg name="world_name" value=”empty_world.launch”
                <arg name="debug" value="$(arg debug)" />
                <arg name="gui" value="$(arg gui)" />
                <arg name="paused" value="$(arg paused)" />
                <arg name="use_sim_time" value="$(arg use_sim_time)" />
                <arg name="headless" value="$(arg headless)" />
        </include>

        <!—Aquí lanzamos el robot indicándole el archivo donde lo tenemos descrito -->
        <param name="robot_description"
                command="$(find xacro)/xacro.py '$(find p3dx_description)/urdf/pioneer3dx.xacro'" />
```

```
<!—Aquí corremos un Script echo en Python donde decimos la posición del robot dentro del mundo -->
<node name="urdf_spawner" pkg="gazebo_ros" type="spawn_model"
        respawn="false" output="screen" args="-x 1.5 -y 0 -z 0 -Y 3.141592654 -urdf -model p3dx -
param robot_description" />
</launch>
```

Por tanto, indicando en el archivo .launch el mundo que queremos lanzar, el robot que queremos lanzar y la posición de este último con el mundo se nos abrirá Gazebo con esas características.

Por ejemplo para el .launch arriba descrito, hemos lanzado un mundo vacío con el robot P3DX virtualizado descrito en la posición “x=1.5, y=0, z=0” y orientación (en radianes): X=0, Y=3.141592654, Z=0.

En la siguiente ilustración podemos ver el mundo y robot que se lanzan con ese archivo .launch:

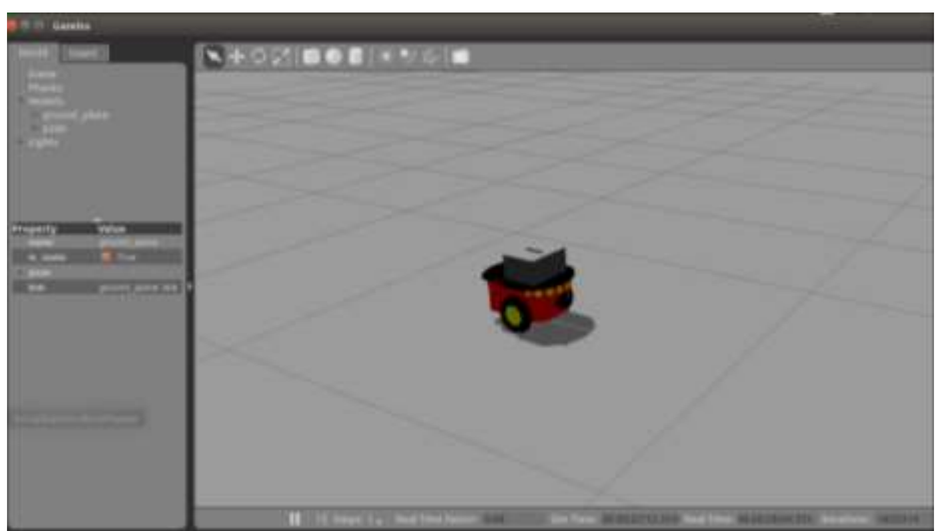


Ilustración 3.9: Modelo P3-DX final en un mundo vacío.

Ahora que ya hemos realizado el robot junto con sus características para que se parezca y se comporte lo más posible a nuestro robot real, vamos a realizar el entorno del pasillo de la zona Oeste primera planta de la Escuela Politécnica Superior de la UAH.

3.3.1.2. Creación del entorno en Gazebo

En el apartado 2.2.3.1 se explicó como diseñar entornos en Gazebo de dos formas, tanto con el building editor de Gazebo, como empezando de cero escribiendo un archivo .sdf.

En este trabajo se ha optado por recurrir al building editor para la creación del entorno.

El entorno que se pretende diseñar es el pasillo de la zona Oeste de la planta primera donde se encuentran los laboratorios del OL1 al OL6.

Este entorno lo podemos ver en el siguiente plano con una vista en planta del mismo:

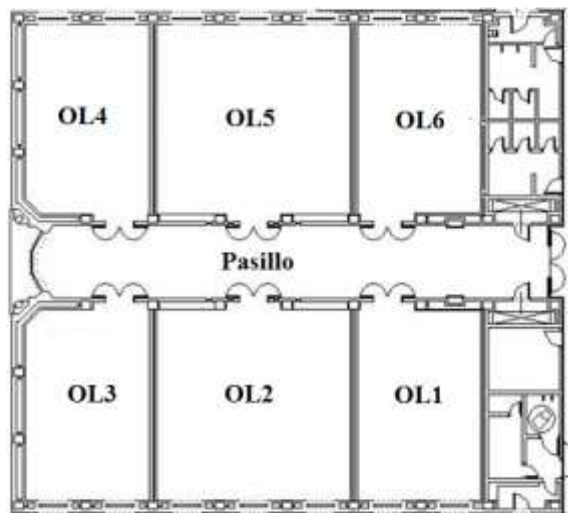


Ilustración 3.10: Pasillo Oeste Escuela Politécnica en formato CAD.

Las medidas del entorno en metros se pueden ver a continuación:

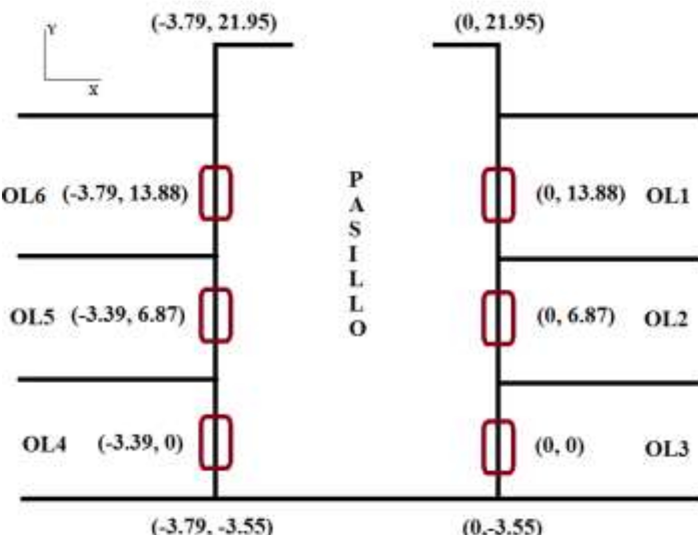


Ilustración 3.11: Medidas del entorno pasillo Oeste Escuela Politécnica.

Donde los cuadrados rojos son las puertas de 1.55 metros cada una y los puntos expuestos son las cuatro esquinas del pasillo así como el punto central de cada puerta. Como vemos se ha tomado como referencia (0, 0) el centro de la puerta del laboratorio OL3 debido a que será ahí donde nuestro robot empezará las trayectorias.

Una vez sabemos cómo es el entorno a diseñar, vamos a explicar cómo hacerlo en el building editor de Gazebo. La herramienta utilizada para este trabajo permite añadir puertas, ventanas, paredes y escaleras como podemos ver en la siguiente figura:

Ilustración 3.12: Paleta para añadir paredes, ventanas, puertas y escaleras en Building Editor de Gazebo.

- **Añadiendo paredes**

Para añadir paredes basta con dar al botón “add wall” y añadir una pared cualquiera. Después dando doble click sobre la pared creada se nos abre una ventana donde podemos poner los puntos inicial y final, así como la altura y el grosor de la puerta. A continuación podemos ver como se añadiría la pared con punto inicial (0, -3.55) y final (0, 21.95):

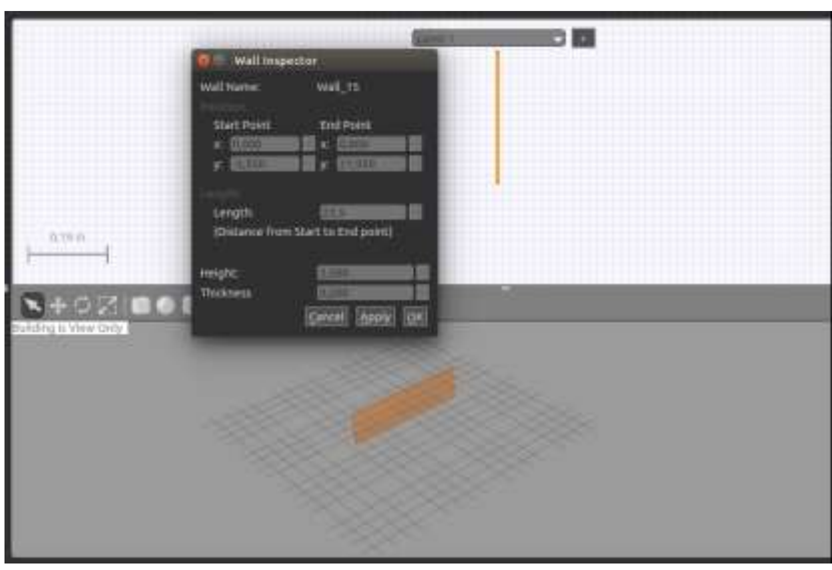


Ilustración 3.13: Añadir paredes a partir de sus puntos inicial y final en Buildin Editor de Gazebo.

Una vez hecho lo expuesto para cada pared, deberemos de colocar las puertas.

- **Añadiendo puertas**

Para añadir puertas basta con dar al botón “add door” y añadir una puerta cualquiera. Después, haciendo doble click sobre la puerta podemos definir su posición, así como el ancho y el alto de la puerta.

Cada puerta de nuestro entorno tendrá un ancho de 1,55 metros. Por ejemplo, la adición de la puerta puerta que se encuentra en la posición (0, 0), es decir, la puerta del laboratorio OL3, se puede ver en la siguiente ilustración:

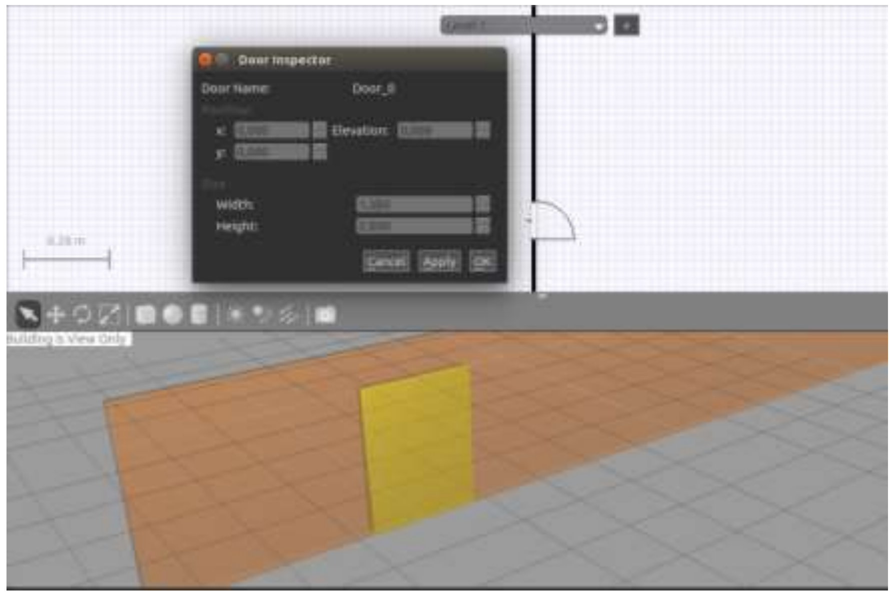


Ilustración 3.14: Añadir puertas a partir de su pose, ancho y alto en Building Editor de Gazebo.

Hacemos lo expuesto para cada una de las puertas que tenemos en el entorno, cada una con su pose correspondiente.

Para la realización de este trabajo se ha hecho un entorno sin tener en cuenta los laboratorios OL1 y OL6 dado que las trayectorias que se probarán no pasarán por estos laboratorios.

- **Pasando el entorno creado a archivo .world**

Una vez hemos puesto tanto las paredes como las puertas del entorno, este queda de la siguiente manera en Gazebo (sin tener en cuenta los laboratorios OL1 y OL6 como se dijo anteriormente):

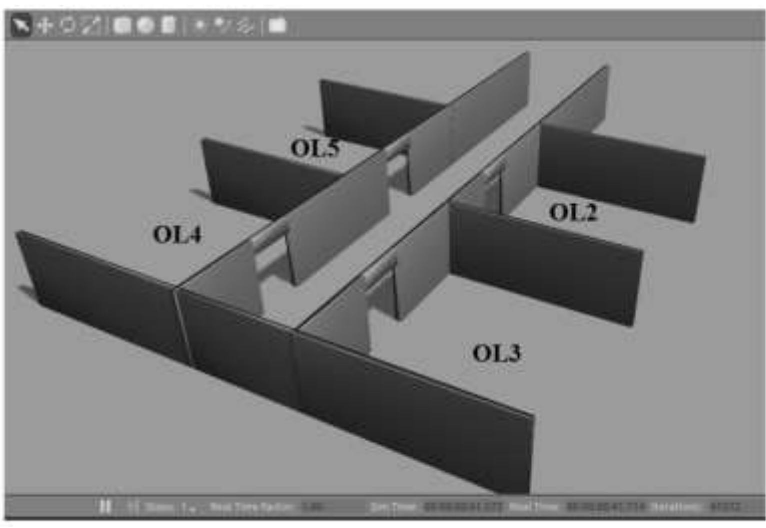


Ilustración 3.15: Pasillo Oeste (primera planta de la EPS) creado con building Editor en Gazebo.

Ahora, se guarda el entorno en un archivo .sdf, el cual tendrá la misma estructura explicada en el apartado 2.2.3.1.

Teniendo ya tanto el entorno como el robot creados podemos verificar que funcionan correctamente en Gazebo lanzando el robot virtual en el entorno.

- **Lanzando robot virtual en el entorno creado**

Ahora que tenemos tanto el entorno como el robot virtual creados vamos a lanzarlos en Gazebo. Para ello basta con programar un archivo .launch en el que lanzamos el nodo Gazebo con el entorno virtual creado y el robot en una posición y orientación determinadas. En el apartado 3.3.1.1 – lanzando el robot al mundo Gazebo – podemos ver la estructura de un archivo .launch.

A continuación describiremos paso por paso el archivo .launch que nos queda en este caso para lanzar los entornos y robot virtuales creados:

- Argumentos de configuración del archivo .launch para el comportamiento del nodo gazebo.

```
<!—Estos argumentos son configuración del archivo launch. Los ponemos por defecto. -->
<arg name="paused" default="false" />
<arg name="use_sim_time" default="true" />
<arg name="gui" default="true" />
<arg name="headless" default="false" />
<arg name="debug" default="false" />
```

  Estos argumentos los ponemos por defecto y significan lo siguiente si son true:

  - Paused: Inicia Gazebo en un estado de pausa.
  - Use_sim_time: Pide a los nodos de ros el tiempo de simulación publicado en Gazebo, publicado en el topic /clock.
  - Gui: Lanza la interfaz Gazebo para el usuario.
  - Headless: Desactiva cualquier función que llama a los componentes renderizados tipo Ogre.
  - Debug: Lanza gzserver (Servidor de Gazebo) en modo debug para ver que ocurre en cada momendo desde terminal.

- Lanzamos el mundo creado junto con argumentos de configuración como sigue:

```
<!—A continuación cargamos el mundo de Gazebo -->
<include file="$(find gazebo_ros)/launch/empty_world.launch">
        <arg name="world_name" value="$(find p3dx_gazebo)/ol3_floor/Planta_oeste.sdf" />
        <arg name="debug" value="$(arg debug)" />
        <arg name="gui" value="$(arg gui)" />
        <arg name="paused" value="$(arg paused)" />
        <arg name="use_sim_time" value="$(arg use_sim_time)" />
        <arg name="headless" value="$(arg headless)" />
</include>
```

  Lo que hacemos es lanzar un mundo vacío y posteriormente cargar nuestro mundo creado indicándolo en el argumento "world_name", poniendo el paquete donde se encuentra (p3sx_gazebo) y la ruta local "/ol3_floor/Planta_oeste.sdf". Los demás argumentos tienen el mismo valor y significado que en el anterior fragmento descrito.

- Lanzamos el robot virtual creado en el mundo:

```
<!—Cargamos el robot en parameter server -->
<param name="robot_description"
        command="$(find xacro)/xacro.py '$(find p3dx_description)/urdf/pioneer3dx.xacro'" />

<!—Corremos un script Python para indicar la posición y orientación del robot en el mundo -->
<node name="urdf_spawner" pkg="gazebo_ros" type="spawn_model"
        respawn="false" output="screen" args="-x 1.5 -y 0 -z 0 -Y 3.141592654 -urdf -model p3dx -param
        robot_description" />
```

Primero, en el parámetro "robot_description" cargamos el robot especificando en qué paquete se encuentra (p3dx_description) y la ruta local "/urdf/pioneer3dx.xacro".

A continuación corremos un script hecho en lenguaje Python enviando una llamada de servicio a "gazebo_ros" para indicar la posición y orientación del robot en el mundo. Recuérdese que el punto (0, 0) era el centro de la puerta del laboratorio OL3, por lo que en este ejemplo lo hemos puesto dentro del laboratorio OL3 orientado hacia la puerta, más concretamente en:

- Pose (1.5, 0, 0); (x, y, z)
- Orientación (0, pi, 0); (roll, pitch, yaw.)

Quedando el archivo .launch completo de la siguiente forma:

```
<launch>
	<!—Estos argumentos son configuración del archivo launch. Los ponemos por defecto. -->
	<arg name="paused" default="false" />
	<arg name="use_sim_time" default="true" />
	<arg name="gui" default="true" />
	<arg name="headless" default="false" />
	<arg name="debug" default="false" />

	<!—A continuación cargamos el mundo de Gazebo -->
	<include file="$(find gazebo_ros)/launch/empty_world.launch">
		<arg name="world_name" value="$(find p3dx_gazebo)/ol3_floor/Planta_Robot2015_oeste.sdf" />
		<arg name="debug" value="$(arg debug)" />
		<arg name="gui" value="$(arg gui)" />
		<arg name="paused" value="$(arg paused)" />
		<arg name="use_sim_time" value="$(arg use_sim_time)" />
		<arg name="headless" value="$(arg headless)" />
	</include>

	<!—Cargamos el robot en parameter server -->
	<param name="robot_description"
		command="$(find xacro)/xacro.py '$(find p3dx_description)/urdf/pioneer3dx.xacro'" />

	<!—Corremos un script Python para indicar la posición y orientación del robot en el mundo -->
	<node name="urdf_spawner" pkg="gazebo_ros" type="spawn_model"
		respawn="false" output="screen" args="-x 1.5 -y 0 -z 0 -Y 3.141592654 -urdf -model p3dx -param
robot_description" />
</launch>
```

Una vez que tenemos el archivo .launch basta con ir al terminar y ejecturlo de la siguiente forma:

```
$ roslaunch package_name file.launch
```

Especificando el paquete donde se encuentra el archivo y el nombre del archivo .launch.

Y ejecutándolo podemos ver el robot virtual creado dentro del mundo en la posición y orientación indicadas:

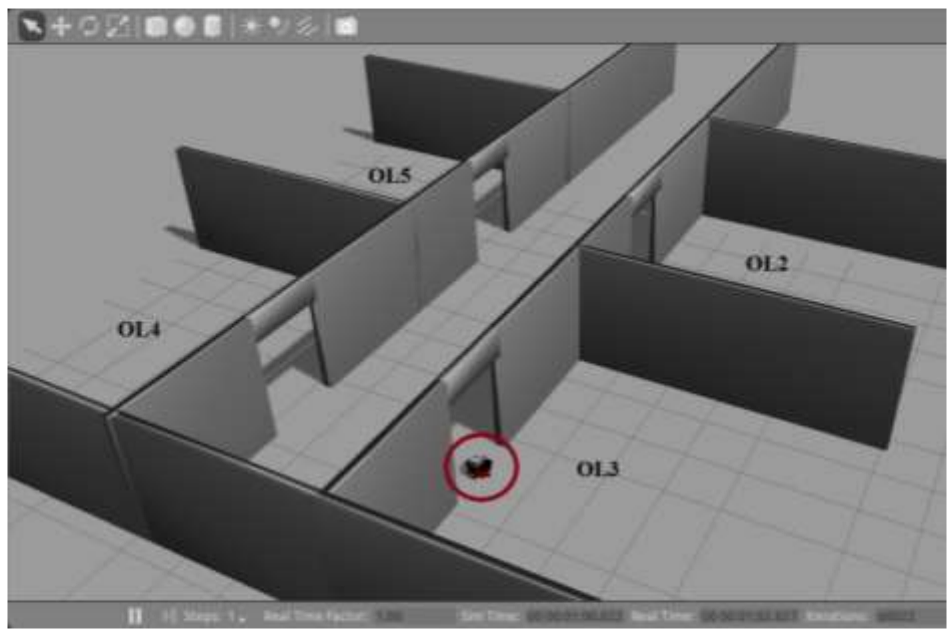


Ilustración 3.16: Robot virtual P3DX dentro del entorno virtual pasillo Oeste.

Una vez tenemos el mundo y el robot virtual creados correctamente y sabiéndolos lanzar, vamos a explicar a continuación cómo nos podemos comunicar con el robot para mandarle velocidades.

3.3.1.3. Intercambiando velocidades del robot de Gazebo desde ROS

En este apartado se va a explicar cómo crear un nodo y lanzarlo para enviar velocidades lineal y angular al robot así como recibir los valores de velocidades proporcionados por la odometría del robot virtual. Después en el apartado 3.5.1 se extenderá este código para que estas velocidades sean las mandadas por el Centro Remoto (PC con Matlab/Simulink) a través de un socket.

- **Enviar velocidades al robot:**

La idea general para enviar velocidades al robot es crear un nodo el cual publique datos de velocidad en el topic correspondiente del robot que tendrá asociado un plugin que se encargará de enviar estos comandos a los motores de las ruedas.

La idea básica la podemos ver en el siguiente gráfico:

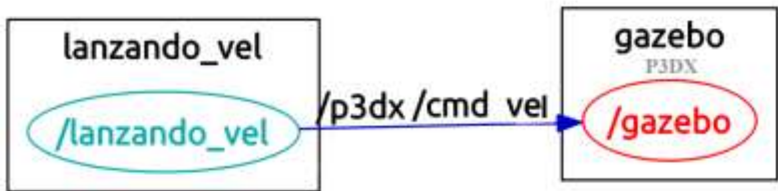


Ilustración 3.17: Gráfico de nodos y topics para enviar velocidades al robot virtual.

Como podemos ver tenemos dos nodos, uno que es “lanzando_vel” y otro que es “gazebo”. El primer nodo se encarga de publicar en el topic “/p3dx/cmd_vel” las velocidades deseadas a enviar al robot, y el nodo “gazebo” está subscrito a este topic que tendrá las velocidades enviadas y un plugin asociado para mover el robot. El plugin en cuestión es el siguiente:

```
<gazebo>
		<plugin name="differential_drive_controller" filename="libgazebo_ros_diff_drive.so">
			<alwaysOn>true</alwaysOn>
			<updateRate>100</updateRate>
			<leftJoint>base_right_wheel_joint</leftJoint>
			<rightJoint>base_left_wheel_joint</rightJoint>
			<wheelSeparation>0.39</wheelSeparation>
			<wheelDiameter>0.15</wheelDiameter>
			<torque>5</torque>
			<commandTopic>$p3dx/cmd_vel</commandTopic>
			<odometryTopic>$p3dx/odom</odometryTopic>
			<odometryFrame>odom</odometryFrame>
			<robotBaseFrame>base_link</robotBaseFrame>
		</plugin>
	</gazebo>
```

Donde como vemos en color rojo, indicamos el nombre del topic correspondiente a las velocidades lineal y angular, cuyo nombre es p3dx/cmd_vel.

Cuando se creó el robot en el apartado 3.3.1.1 se escribieron todos los plugin y topics que el robot debe de tener, y al lanzar con el archivo .launch el robot, estos topics están activos automáticamente para que se publiquen o se subscriban a ellos otros nodos.

Para poder ver los topics activos una vez lanzado el robot basta con ejecutar en el terminal el comando "rostopic list", obteniendo lo siguiente en nuestro caso:

```
alvaro@alvaro-ubuntu-PC: ~
alvaro@alvaro-ubuntu-PC:~$ rostopic list
/clock
/gazebo/link_states
/gazebo/model_states
/gazebo/parameter_descriptions
/gazebo/parameter_updates
/gazebo/set_link_state
/gazebo/set_model_state
/p3dx/base_pose_ground_truth
/p3dx/cmd_vel
/p3dx/laser/scan
/p3dx/odom
/rosout
/rosout_agg
/tf
```

Ilustración 3.18: Topics activos una vez lanzado el robot virtual.

Como vemos, los topics que están activos correspondientes al robot p3dx son los que tienen como primer tag "p3dx", y entre ellos está el mencionado cmd_vel, el cual es el encargado de mover al robot con las velocidades recibidas.

Una vez expuesta la idea general de como enviar velocidades al robot, vamos a ver cómo se puede programar el nodo "lanzando_vel" de la Ilustración 3.17, el cual se encargará de enviar velocidades al topic cmd_vel, o lo que es lo mismo, publicando en él.

Los nodos que programemos en ROS son archivos .cpp y se pueden editar con cualquier editor de texto.

La estructura es la siguiente:

- Antes de nada, debemos de inicializar el nodo que vayamos a crear, para ello se debe de llamar a la función ros::init la cual nos permite darle nombre al nodo, tal que:

```
ros::init(argc, argv, "lanzando_vel");
```

- Una vez inicializado debemos de crear un ros::NodeHandle, el cual será el principal punto de acceso a las comunicaciones de este nodo con el resto del sistema en ROS (topics y demás nodos), tal que:

```
ros::NodeHandle n;
```

  Solo hace falta crear un ros::NodeHandle tanto para publicar como para subscribirse a la vez.

- Lo siguiente que debemos de hacer es indicar en que topic queremos publicar de la siguiente manera:

```
ros::Publisher vel_pub_ = n.advertise<geometry_msgs::Twist>("p3dx/cmd_vel", 1);
```

La funcion advertise() es la manera en la que se dice a ROS en que topic queremos publicar datos. Esta función invoca al nodo “master” de ROS, el cual guarda el registro de quien esta publicando y quien subscribiéndose, y notificará a cualquiera que esté intentando publicar o subscribir en el topic a través de mensajes P2P. La función advertise() retorna un objeto Publisher, el cual nos permitirá publicar mensajes en el topic a través de la llamada publish(), la cual la mencionaremos más adelante que está dentro del loop principal del nodo. El segundo parámetro de advertise() es el tamaño del buffer de mensajes.

- Una vez, hemos iniciado el nodo, así como la comunicación para poder acceder a los topics e indicado en que topic se quiere publicar, se debe de hacer el loop principal del programa donde se publicarán los mensajes. El loop principal vendrá dado por una condición, la cual es ros::ok(), que es true siempre y cuando el sistema ROS funcione correctamente y no se cuelgue o el usuario no lo haya parado mediante Ctrl-C desde el terminal.

```
while (ros::ok())
```

- Creamos una variable del tipo “geometry_msgs::Twist”, dado que las velocidades en ROS son de este tipo:

```
geometry_msgs::Twist cmd;
```

- Ahora asignamos a esa variable cmd las velocidades lineales y angulares que se deseean publicar al robot de la forma:

```
cmd.linear.x = velocidad_linal_eje_x_deseada;
cmd.linear.y = 0;
cmd.linear.z = 0;
cmd.angular.x = 0;
cmd.angular.y = 0;
cmd.angular.z = velocidad_angular_eje_z_deseada;
```

Las velocidades lineales se aplican en el eje X en el sistema de coordenadas local del robot, así como las angulares en Z, o lo que es lo mismo, Yaw. Las unidades son metros/segundo para la velocidad lineal y radianes/segundo para la angular.

- Como dijimos antes, la función advertise() devuelve un objeto publisher que es el que nos permite publicar en el topic, por tanto una vez tenemos las velocidades que queremos mandar al robot, hacemos que el nodo las publique en el topic de la siguiente forma:

```
vel_pub_.publish(cmd);
```

donde vel_pub_ es el objeto ros::Publisher que devuelve advertise(), y cmd son las velocidades que se quieren publicar, asignadas en el punto anterior.

Quedando por tanto la siguiente programación para el nodo:

```
int main(int argc, char **argv)
{
        ros::init(argc, argv, "lanzando_vel");
        ros::NodeHandle n;
        ros::Publisher vel_pub_ = n.advertise<geometry_msgs::Twist>("p3dx/cmd_vel", 1);

        while (ros::ok()){
                geometry_msgs::Twist cmd;

                cmd.linear.x = velocidad_linal_eje_x_deseada;
                cmd.linear.y = 0;
                cmd.linear.z = 0;
                cmd.angular.x = 0;
                cmd.angular.y = 0;
                cmd.angular.z = velocidad_angular_eje_z_deseada;

                vel_pub_.publish(cmd);
        }
}
```

Para lanzar el nodo basta con ejecutar el siguiente comando por terminal:

```
rosrun <package> <executable>
```

Que en nuestro caso es: “rosrun lanzar_velocidades lanzar_velocidades_node”

- **Recibir velocidades del robot**

La idea general para recibir velocidades leídas por la odometría del robot es la misma que para enviarlas, que es crear un nodo el cual se subscriba al topic que contiene esta información, dicho topic es “p3dx/odom”, el cual podemos ver en la Ilustración 3.18.

La idea básica la podemos ver en el siguiente gráfico:

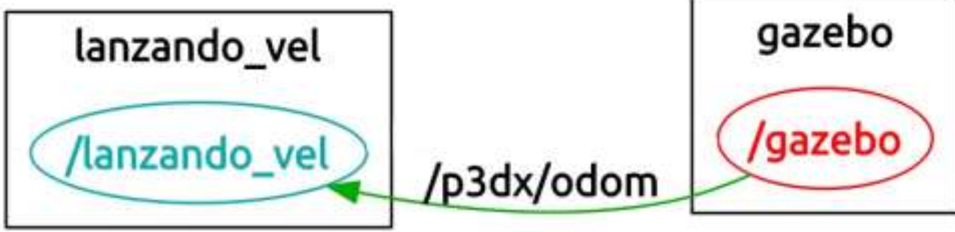


Ilustración 3.19: Gráfico de nodos y topics para recibir velocidades del robot virtual.

Una vez expuesta la idea general de como enviar velocidades al robot, vamos a ver cómo se puede programar el nodo "lanzando_vel" para subscribirse al topic correspondiente y obtener los datos:

- Antes de nada, debemos de inicializar el nodo que vayamos a crear, para ello se debe de llamar a la función ros::init la cual nos permite darle nombre al nodo, tal que:

```
ros::init(argc, argv, "lanzando_vel");
```

- Una vez inicializado debemos de crear un ros::NodeHandle, el cual será el principal punto de acceso a las comunicaciones de este nodo con el resto del sistema en ROS (topics y demás nodos), tal que:

```
ros::NodeHandle n;
```

- A continuación y fuera del loop principal debemos de crear una función callback la cual se ejecutará cada vez que llegue un nuevo mensaje al topic en el que se quiere subscribirse, tal que:

```
void velsubcallback(const nav_msgs::Odometry& msg)
{
	Velocidad_lineal_x= msg.twist.twist.linear.x;
	Velocidad_angular_z= msg.twist.twist.angular.z;

	ROS_INFO("velocidad lineal obtenida=%f, velocidad angular obtenida=&f)",
	Velocidad_lineal_x, Velocidad_angular_z);
}
```

  Los datos que contienen el topic al que estamos subscritos están en la variable msg que es del tipo nav_msgs::Odometry, que en los campos twist.twist.lineal.x tiene la velocidad en el eje X leída por la odometría del robot, y en el campo twist.twist.angular.z tiene la velocidad angular en Z leída de la odometría del robot.

- Lo siguiente que debemos hacer es indicar el topic al que nos queremos subscribir

```
ros::Subscriber odom_sub = n.subscribe("p3dx/odom", 1000, velsubcallback);
```

  La funcion subscribe() es la manera en la que se dice a ROS el topic del que queremos recibir datos. Esta función invoca al nodo "master" de ROS, el cual guarda el registro de quien está publicando y quien subscribiéndose, y notificará a cualquiera que esté intentándo publicar o subscribir en el topic a través de mensajes P2P. Estos mensajes son pasados a la función caalback descrita anteriormente. La función subscribe() retorna un objeto Subscriber al que estaremos subscritos hasta que queramos.

- En el loop principal se llamará a las funciones callbacks cada vez que haya un dato nuevo en este topic subscrito.

Tanto publicar como subscribirse a un topic se puede programar en un mismo nodo, ya que un nodo puede publicar y subscribirse a varios topics a la vez, quedando en este caso el siguiente diagrama de nodos y topics:

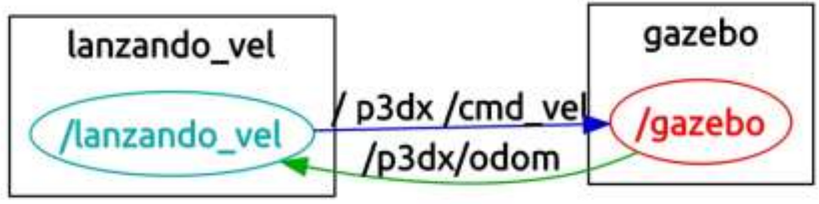


Ilustración 3.20: Nodo publicando velocidades y subscribiéndose a los datos del robot.

Como vemos, el nodo "lanzando_vel" publica en el topic p3dx/cmd_vel a la vez que se subscribe al topic "p3dx/odom".

La programación de este nodo queda da la siguiente forma:

```
void velsubcallback(const nav_msgs::Odometry& msg)
{
        Velocidad_lineal_x= msg.twist.twist.linear.x;
        Velocidad_angular_z= msg.twist.twist.angular.z;
        ROS_INFO("velocidad lineal obtenida=%f, velocidad angular obtenida=&f)",
        Velocidad_lineal_x, Velocidad_angular_z);
}
int main(int argc, char **argv)
{
        ros::init(argc, argv, "lanzando_vel");
        ros::NodeHandle n;
        ros::Publisher vel_pub_ = n.advertise<geometry_msgs::Twist>("p3dx/cmd_vel", 1);
        ros::Subscriber odom_sub = n.subscribe("p3dx/odom", 1000, velsubcallback);
        while (ros::ok()){
                geometry_msgs::Twist cmd;
                cmd.linear.x = velocidad_linal_eje_x_deseada;
                cmd.linear.y = 0;
                cmd.linear.z = 0;
                cmd.angular.x = 0;
                cmd.angular.y = 0;
                cmd.angular.z = velocidad_angular_eje_z_deseada;
                vel_pub_.publish(cmd);
                ros::spinOnce();
        }
}
```

Pudiéndolo lanzar con: rosrun lanzar_velocidades lanzar_velocidades_node .

Ya sabemos cómo crear un robot, crear un entorno, lanzarlos en el simulador Gazebo y cómo mandar velocidades al robot virtual a través de ROS así como subscribirse a cualquier topic para recibir información del robot.

Lo siguiente que se expondrá es ver toda la programación que se necesita hacer en el Centro Remoto (PC con Matlab/Simulink) para el diseño del controlador.

Posteriormente a esto, se extenderá el código del nodo de ROS para que las velocidades enviadas a los robots sean las procedentes del centro remoto y que este reciba las velocidades leídas por el topic odom, o lo que es lo mismo, por la odometría del robot virtual.

3.4. Descripción del algoritmo de control y generación de referencias a implementar en el PC Centro Remoto

En este apartado se detallará la programación a realizar en el Centro Remoto, o lo que es lo mismo, en el PC tratado como Host.

Como ya se ha comentado, este PC corre con un sistema operativo GNU/Linux Ubuntu 10.04 LTS en el que se encuentran instaladas las herramientas Matlab/Simulink/RTW y el kernel modificado de Linux RTAI.

En este apartado se explicará la identificación de la planta en VVEE del robot P3DX y un breve resumen de la solución adoptada de control para el seguimiento de las consignas, esto es, velocidad lineal y angular del robot.

3.4.1. Identificación de la planta P3DX

Al querer realizar un algoritmo de control, más concretamente un seguimiento de velocidad lineal y angular, debemos de identificar la planta robótica que se vaya a utilizar, en este caso, el robot P3DX.

Este modelo se identificará en VVE con el fin de poder utilizarlo en la herramienta Matlab/Simulink para poder diseñar el algoritmo de control. Si lo comparamos con ROS/Gazebo, este procedimiento sería la etapa de sacar el modelo del P3DX en Gazebo que se explicó en el apartado 3.3.1.1, pero en este caso, en VVE para poder utilizarlo en Matlab/Simulink.

El modelo en el espacio de estados del P3-DX ha sido obtenido mediante técnicas de identificación y validación a través de ensayos experimentales [13]. La velocidad linear

y angular, generadas en un PC remoto, son enviadas al robot para registrar la respuesta del mismo en lazo abierto. Un retardo constante en el canal de L segundos es incluido en el modelo de la planta, donde este elemento no lineal es aproximado mediante Padé, el cuál es un método que aproxima, en el dominio de Laplace, un retardo a través de un modelo racional. En este caso el retardo L del canal es aproximado como Padé(1,1):

$$e^{-Ls} \simeq \frac{1 - \frac{L}{2}s}{1 + \frac{L}{2}s}$$

Y el resultado del modelo en variables de estado del robot P3-DX en tiempo continuo es:

$$\dot{x}(t) = Ax(t) + Bu(t)$$

$$\dot{x}(t) = \begin{bmatrix} -4.094 & -0.015 & 1664 & 0.7227 \\ -0.008 & -5.042 & 0.326 & 2023 \\ 0 & 0 & -200 & 0 \\ 0 & 0 & 0 & -200 \end{bmatrix} x(t) + \begin{bmatrix} -4.159 & -0.002 \\ -0.001 & -5.057 \\ 1 & 0 \\ 0 & 1 \end{bmatrix} u(t)$$

$$y(t) = Cx(t)$$

$$y(t) = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{bmatrix} x(t)$$

Pero dado que en el laboratorio se trabaja en tiempo discreto con la unidad P3DX, se deberá de calcular el modelo en variables de estado en tiempo discreto, el cual, para un periodo de 100ms queda:

$$x_{k+1} = \overbrace{\begin{bmatrix} 0.6640 & -0.0010 & 5.6403 & -0.0069 \\ -0.0005 & 0.6040 & -0.0031 & 6.2673 \\ 0 & 0 & 2.06e^{-9} & 0 \\ 0 & 0 & 0 & 2.06e^{-9} \end{bmatrix}}^{G} x_k + \overbrace{\begin{bmatrix} -0.3132 & -0.0001 \\ -0.0001 & 0.3659 \\ 0.0050 & 0 \\ 0 & 0.0050 \end{bmatrix}}^{H} u_k$$

$$y_k = \overbrace{\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{bmatrix}}^{C} x_k$$

Donde las matrices G, H, C son las que se muestran y la matriz $D_{(2x2)}$ es una matriz de ceros.

Para poder implementar estas ecuaciones del modelo P3DX en Simulink, se sigue el esquema de la siguiente figura que, a su vez, se puede encapsular en un subsistema el cual llamaremos "P3DX" para que nos sea más cómodo manejarlo en Simulink:

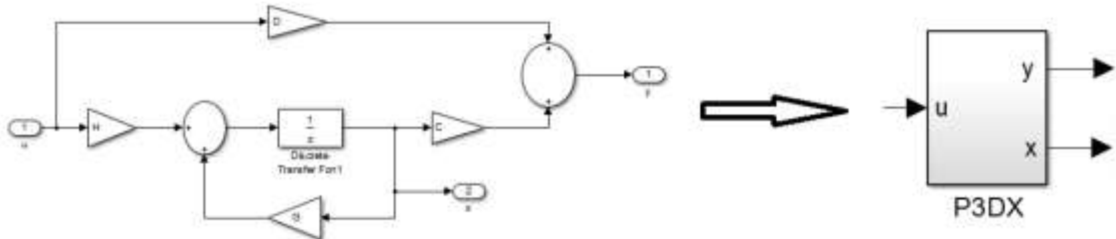


Ilustración 3.21: Modelo del P3DX en VVEE en Simulink.

Y a partir de ese modelo, ya podemos trabajar en el diseño de la solución de control de seguimiento de velocidades.

3.4.2. Solución de control para el seguimiento de consignas del P3DX, servosistema

Como se viene diciendo, en la etapa de diseño de la solución de control es conveniente realizarla con el modelo en VVEE de la planta P3DX identificada ya que si queremos hacer cambios en el diseño, pruebas… es más sencillo realizarlas con este modelo y no sobre la plataforma real o virtual, ahorrándonos tiempo, infraestructura y costes, ya que si se diseña sobre la plataforma real, un diseño fallido puede acarrear problemas en la planta real, y por el contrario, si se diseña sobre el robot virtual se perdería tiempo ya que cada vez que se quiera probar una modificación del diseño, habría que seguir un proceso más costoso en tiempo a la hora de implementarlo en el robot virtual (PC distinto al host) que si queremos probarlo directamente sobre el modelo en VVEE, que es correr simplemente el script de Matlab/Simulink.

El diseño de un regulador cambia el comportamiento dinámico con respecto al original de la planta, pero cuando aparecen perturbaciones externas en la planta, el regulador no es capaz de compensarlas, es por eso por lo que se propone el diseño de un servosistema en este apartado, para poder cerciorarse que la plataforma robótica P3DX siga las consignas mandadas por el centro remoto compensando cualquier perturbación.

El diseño del control se realiza en tiempo discreto dado que se realizará una implementación digital en la unidad robótica P3DX.

El servosistema que se realizará para el seguimiento de velocidades lineal y angular del robot P3DX tendrá la siguiente estructura en el diagrama de bloques de simulink:

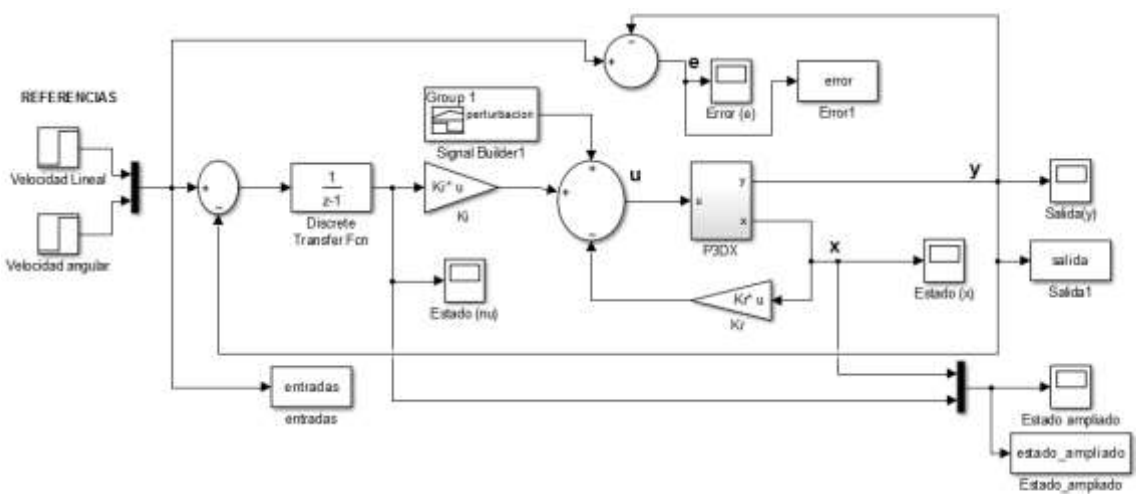


Ilustración 3.22: Diagrama de bloques de un servosistema en Simulink.

Donde el subsistema “P3DX” tiene la forma descrita en la Ilustración 3.21.

Las ecuaciones del servosistema son las siguientes:

$$\begin{cases} Planta\colon\ X_{k+1} = GX_k + Hu_k & (3.1) \\ Intregrador\colon \eta_{k+1} = \eta_k + e_k \quad donde\ el\ error\ es\quad e_k = r_k - CX_k \\ \qquad quedando\ \eta_{k+1} = \eta_k + r_k - CX_k \end{cases}$$

Y sumando ambas expresiones nos queda la siguiente expresión matricial:

$$\underbrace{\begin{bmatrix} X_{k+1} \\ \eta_{k+1} \end{bmatrix}}_{Xa} = \underbrace{\begin{bmatrix} G & 0 \\ -C & I \end{bmatrix}}_{Ga} \begin{bmatrix} X_k \\ \eta_k \end{bmatrix} + \underbrace{\begin{bmatrix} H \\ 0 \end{bmatrix}}_{Ha} u_k + \begin{bmatrix} 0 \\ I \end{bmatrix} r_k \tag{3.2}$$

Donde podemos ver el vector de estados ampliados (Xa) y las matrices Ga y Ha ampliadas, las cuales son las matrices del servosistema. Las matrices G, H y C son las de la planta robótica P3DX descrita en el anterior apartado.

A su vez podemos establecer la ley de control:

$$u_k = -K_r X_k + K_i \eta_k \tag{3.3}$$

La cual podemos expresar en formato matricial como:

$$u_k = -[K_r \quad -K_i] \begin{bmatrix} X_k \\ \eta_k \end{bmatrix} = -\mathrm{K}_a\, \mathrm{X}_{\mathrm{a,k}} \tag{3.4}$$

Existen diferentes métodos para determinar el valor de las ganancias Kr y Ki deseables para el diseño del servosistema. En este trabajo se calcularán a través del método por Lyapunov para sistemas discretos, en el cual, se consigue ajustar la respuesta del sistema mediante una matriz R y una matriz Q, las cuales son establecidas por el diseñador.

Se ha elegido este método dado que en el anterior trabajo [14] previo a este se realizó un estudio extenso al control basado en Lyapunov.

El teorema fundamental de este método dice lo siguiente:

El sistema $X_{a,k+1} = (G_a - H_a K_a)x_{a,k}$ *es asintóticamente estable si y solo si, dada alguna matriz Q simétrica definida positiva, existe alguna matriz P simétrica definida positiva, tal que P sea la única solución de la llamada ecuación de Lyapunov para tiempo discreto:*

$$G_a^T P G_a - P = -Q \tag{3.5}$$

Tal que la función candidata de Lyapunov cuadrática definida positiva es de la forma:

$$V_k = x_{a,k}^T P x_{a,k} \tag{3.6}$$

Siendo su derivada definida negativa,

$$V_{k-1} - V_k = G_a^T P G_a - P = -Q < 0 \tag{3.7}$$

lo cual garantiza la estabilidad del sistema.

Sustituyendo la matriz Ga ampliada del servosistema en la ecuación de Lyapunov para tiempo discreto tenemos la siguiente expresión:

$$G_a^T P G_a - G_a^T P H_a K_a - K_a^T H_a^T P G_a + K_a^T H_a^T P H_a K_a - P = -Q \tag{3.8}$$

Por tanto basta con encontrar la expresión de Ka para que dicha ecuación sea igual a la ecuación algebraica de Riccati para tiempo discreto ya que esta siempre tiene solución (la matriz P). Dicha ecuación algebraica de Riccati para tiempo discreto es la siguiente:

$$P = G_a^T P G_a - (G_a^T P Ha)(R + H_a^T P H_a)^{-1}(H_a^T P G_a) + Q \tag{3.9}$$

Y la expresión de la matriz Ka que lo garantiza es la siguiente:

$$Ka \; = \; (H_a^T P H_a + R)^{-1}(H_a^T P G_a) \tag{3.10}$$

siendo R una matriz definida positiva elegida por el usuario.

En definitiva, con la ganancia Ka teniendo la expresión arriba descrita, se garantiza la existencia de una matriz P solución por la ecuación algebraica de Riccati y, con ello, la existencia de una función de Lyapunov que garantiza la estabilidad del sistema.

Para la solución de la matriz P por la ecuación algebraica de Riccati se puede utilizar el comando "dare" de Matlab. Esta función utiliza la ecuación algebraica de Riccati en tiempo discreto como:

$$G^T XG - E^T XE - (G^T XH + S)(H^T XH + R)^{-1}(H^T XG + S^T) + C^T QC = 0 \quad (3.11)$$

Siendo:

- X la solución a la ecuación de dimensiones nxn (en nuestro caso es la matriz P)
- G, H, las matrices Ga y Ha ampliadas del servosistema.
- E, S, Q, R y C matrices de coeficientes reales conocidos.

Y necesitamos que S=0, E=I y C=I para que la ecuación (3.11) utilizada por "dare" tenga la misma expresión que la ecuación (3.9) que se ha descrito para diseñar el servosistema por Lyapunov, por lo que teniendo esto, el usuario simplemente tiene que elegir las matrices Q y R para la realización del servosistema.

Las fases de diseño del servosistema mediante Lyupunov serían las siguientes:

1. Fijar unos valores de R y Q (definidos positivos).

2. Obtener la solución P a partir de la ecuación algebráica de Riccati ("dare").

3. Obtener la matriz Ka, la cual contiene las ganancias Kr y Ki para el diseño del servosistema según la ecuación $K_a = [K_r \quad -K_i]$.

4. Una vez tenemos las ganancias Kr y Ki ya se puede implementar el servosistema en Simulink.

A la hora de realizar el servo, tenemos que tener presente las dimensiones de las matrices que estamos utilizando:

En cuanto a la planta:

- G Є R^{4x4} , matriz 4x4 de estado de la planta discretizada.
- H Є R^{4x2}, matriz 4x2 de entrada de la planta discretizada.
- C Є R^{2x4}, matriz 2x4 de salida de la planta.

En cuanto a la utilización de la función 'dare' de Matlab para la ecuación algebraica de Riccati:

- Ga Є R^{6x6}, matriz G ampliada 6x6.
- Ha Є R^{6x2}, matriz H ampliada 6x2
- Q Є R^{6x6}, matriz 6x6 simétrica definida positiva, eligida por el usuario.
- P Є R^{6x6}, matriz 6x6 solución de la ecuación de Ricatti y de Lyapunov.
- R Є R^{2x2}, matriz 2x2 de ajuste definida positiva, elegida por el usuario.

- S Є R^{6x2}, matriz de ceros 6x2 para la utilización de la función 'dare'.
- E Є R^{6x6}, matriz I6x6 para la utilización de la función 'dare'.

En cuanto a las ganancias Kr y Ki del sistema (objetivo de diseño):

- Ka Є R^{2x6}, matriz 2x6 la cual contiene Kr y Ki de la forma $Ka = [Kr\ -Ki]$
- Kr Є R^{2x4}, matriz 2x4 objetivo de diseño.
- Ki Є R^{2x2}, matriz 2x2 objetivo de diseño.

Por tanto, pasándole a la función "dare" de Matlab las matrices Ga, Ha, Q, R, S, y E, obtenemos la matriz P, los autovalores del servosistema y la matriz de diseño Ka.

- **<u>Validación del Servosistema por Lyapunov</u>**

Para la validación del servosistema en la planta del robot P3DX en VVEE descrita en la pág 79 se van a utilizar las matrices de diseño $Q = 10 * I_{6x6}$ y $R = \begin{bmatrix} 0.05 & 0 \\ 0 & 0.05 \end{bmatrix}$.

Para ver cómo afecta la elección de Q y R en el comportamiento del servosistema así como en el estudio del control por Lyapunov y Ricatti, se recomienda leer las referencias [14] [15] [16] [17].

Con estas matrices de diseño obtenemos lo siguiente:

- Matriz P solución a la ecuación algebraica de Ricatti:

```
P =
   27.0396   -0.0006    3.6769   -0.0040  -16.6067    0.0000
   -0.0006   26.7509   -0.0027    2.8666    0.0002  -16.4747
    3.6769   -0.0027   26.0420   -0.0189   -1.7882    0.0000
   -0.0040    2.8666   -0.0189   24.5960    0.0014   -1.4600
  -16.6067    0.0002   -1.7882    0.0014   26.3962    0.0000
    0.0000  -16.4747    0.0000   -1.4600    0.0000   26.3340
```

que como podemos ver es simétrica definida positiva, condición necesaria para la estabilidad por Lyapunov.

- Autovalores del servo: [0.3781, 0.3796, -0.0536, -0.0610, 0, 0], que como podemos ver el sistema es estable dado que todos se encuentran dentro de la circunferencia de radio unidad.

- Matriz Ka, de la cual se pueden sacar las matrices Kr y Ki:

$$K_a = \begin{bmatrix} 3.9949 & -0.0018 & 14.6320 & -0.0155 & -1.9190 & -0.0006 \\ -0.0007 & 3.2820 & -0.0056 & 16.8645 & -0.0003 & -1.6568 \end{bmatrix}$$

Kr -Ki

Con este diseño, ya se puede simular el diagrama de bloques de simulink del servosistema descrito en la Ilustración 3.22, el cual, las referencias que pondremos serán tipo escalón de valor 2 para la velocidad lineal y de valor 1 para la velocidad angular. En la siguiente imagen se comprueba el error y la salida del servosistema a lo largo del tiempo:

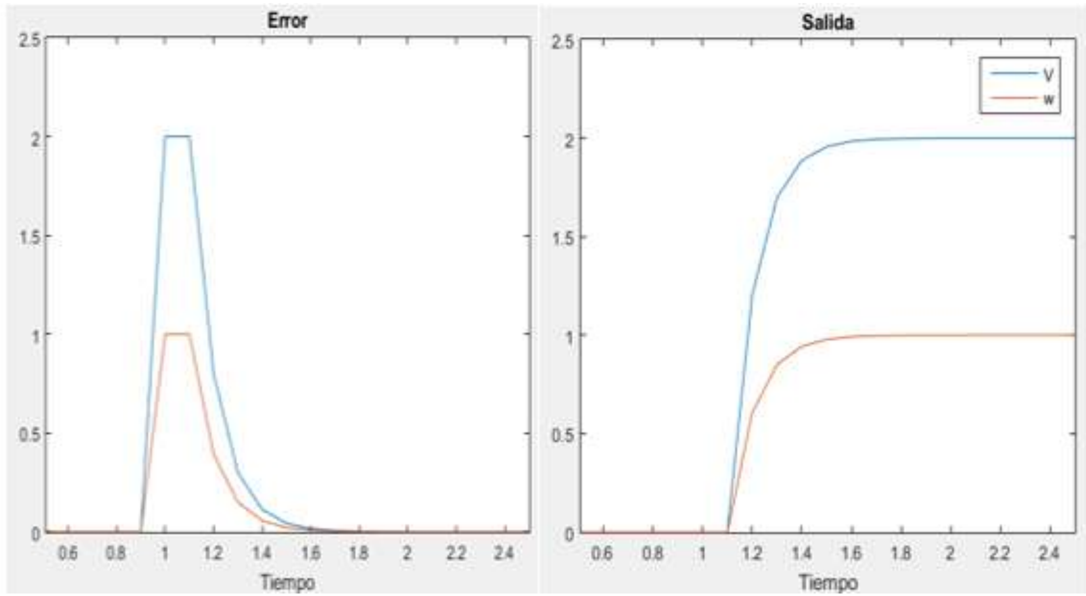


Ilustración 3.23: Error y salida del servosistema diseñado mediante Lyapunov.

Que como podemos observar, la salida sigue a las referencias aplicadas de 2 y 1 de velocidad lineal y angular, lógicamente, con una constante de tiempo, es decir, la planta tarda en seguir a la referencia. El error que se tiene es nulo en régimen permanente, lo cual es lógico ya que la salida sigue a la referencia.

Una vez se tiene diseñado y comprobado el algoritmo de control, se debe de probar en la plataforma robótica virtual, y posteriormente en la real para ver sus diferencias.

- **Servosistema con estimador**

Para poder probar el control en los robots es necesario un estimador de estados ya que no todas las variables de estado que describen el comportamiento del robot virtual/real son medibles debido a que solo se tiene acceso a las variables de salida y entrada de los robots. Para solucionar esto, se va a diseñar un estimador basado en Lyapunov que se añadirá al diagrama de bloques de la Ilustración 3.22 quedando:

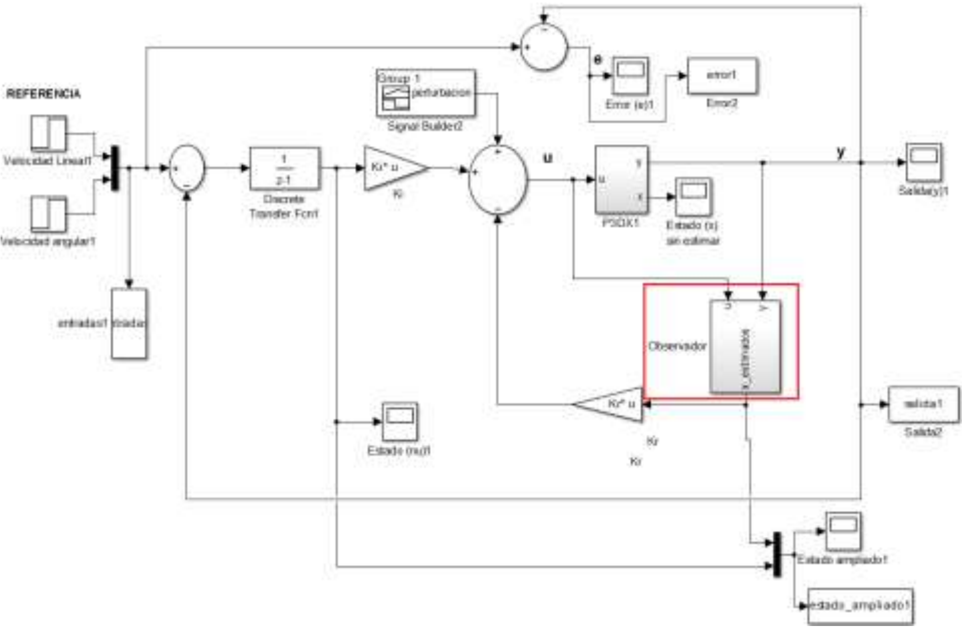


Ilustración 3.24: Diagrama de bloques de un Servosistema con observador en Simulink.

Donde el estimador tiene la siguiente estructura:

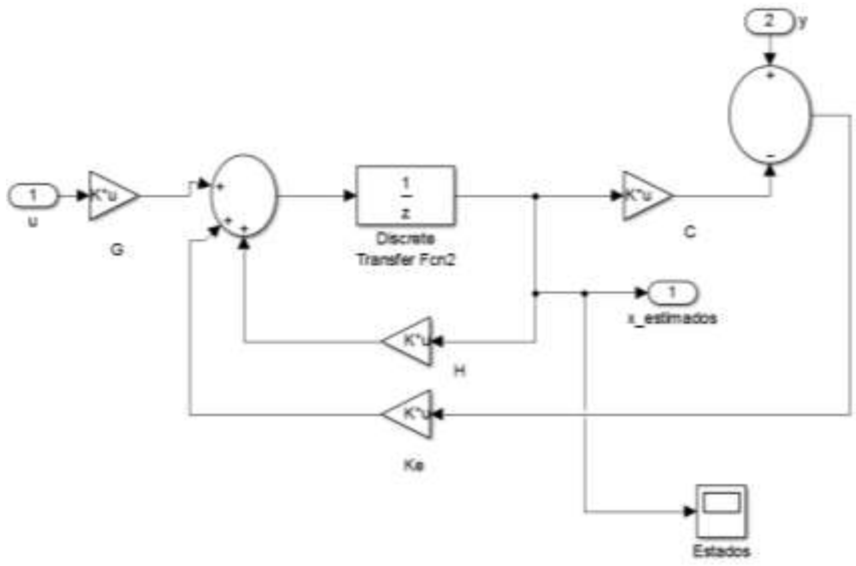


Ilustración 3.25: Diagrama de bloques del observador en Simulink.

La matriz de ganancia Ke es el objetivo de diseño, la cual se calculará de la misma forma que se calculó la matriz Ka para el diseño del servosistema anteriormente. Los autovalores del servosistema deben de ser más rápidos que los de la planta, y esto se puede controlar modificando la matriz Q de diseño del observador por Lyapunov (a mayor Q más rápido).

En el apartado 3.5.2 se explicará cómo modificar el diagrama de bloques de Simulink de la Ilustración 3.24 para que la planta sea la real o virtual en vez de la planta en VVEE.

3.5. Comunicación entre PC Centro Remoto – PC Robot Virtual / Robot Real

En este apartado se va a explicar como se ha de realizar la comunicación entre el centro remoto (PC 1) y el robot virtual (PC 2) o robot real.

Se describe como se integran todas las herramientas vistas para la implementación del algoritmo de control en el robot virtual/real.

3.5.1. Socket

La comunicación entre el centro remoto y el robot virtual y real require de una componente física y de otra lógica. La componente física se ha resuelto mediante conexión ethernet disponible tanto en el robot P3-DX como en el PC que virtualiza dicho robot. Se utiliza el estándar de comunicación 802.11 (Wifi). La componente lógica se ha resuelto mediante aplicaciones cliente-servidor que aportan la transparencia, fiabilidad y flexibiliad necesarias para este tipo de aplicación. Cada aplicación cliente-servidor tiene asociado un canal de comunicación, a cuyo descriptor se le conoce como socket [18]. Mediante el socket se establece el protocolo de comunicación (UDP en este caso), la dirección IP y el puerto, lo que permite identificar a cliente y servidor de forma única.

La siguiente figura describe el funcionamiento de la aplicación cliente-servidor implementada para el enlace centro remoto - robot virtual/real:

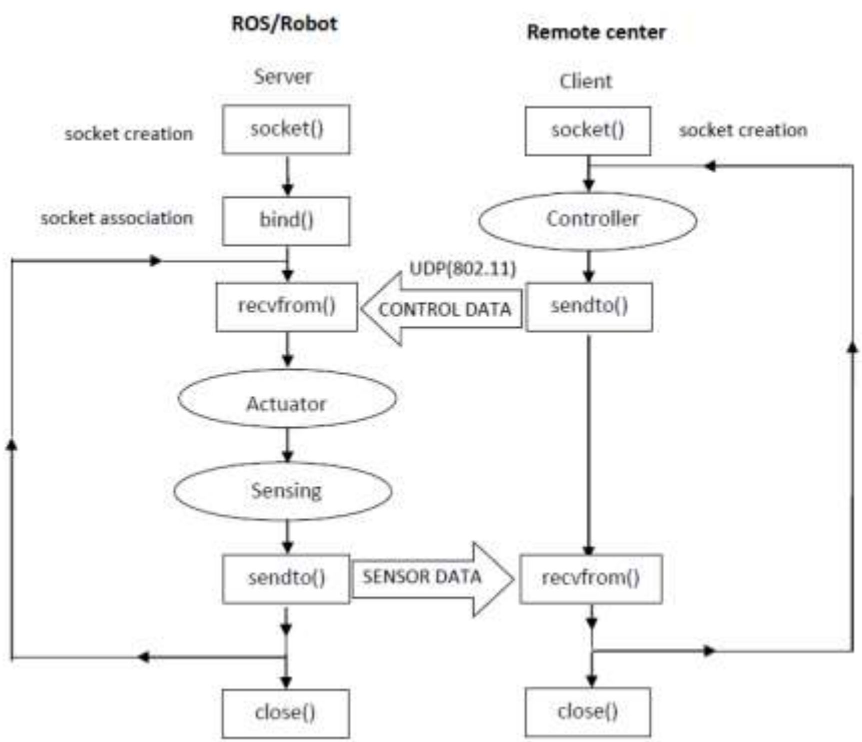


Ilustración 3.26: Protocolo de comunicación entre el centro remoto y el robot virtual/real.

Por tanto deberemos de programar un “socket” para establecer la comunicación y poder enviar y recibir datos del PC Centro remoto al robot virtual/real.

A continuación se describen las funciones del socket que aparecen en la Ilustración 3.26. En las funciones sendto() y recvfrom() se va a puntualizar las diferencias entre cuando se está comunicando con el robot virtual o con el robot real.

- Socket(): Esta función puede ser usada para crear un socket de cualquier protocolo soportado. La estructura de esta función es la siguiente:

```
#include <sys/types.h>
#include <sys/socket.h>
int socket(int domain, int type, int protocol);
```

  Donde los argumentos que acepta la función son:

    - Domain: la familia del protocolo a usar
    - Type: El tipo del socket
    - Protocol: el protocolo a usar que está dentro de la familia especificada en “domain”.

Esta función Socket() devuelve 0 o positivo si el socket se ha creado correctamente y -1 si ha habido algún error.

En nuestro caso, los argumentos que debemos de poner son los siguientes:

- Domain: AF_INET, para indicar que una dirección IP va a ser utilizada con el protocolo indicado.
- Type: sock_dgram, el cual permite pérdidas de paquetes y es eficiente.
- Protocol: 0, el cual es el usado para la creación de un socket UDP, protocolo orientado a mensajes.

Por tanto, la creación del socket quedaría de la forma:

```
int socket_ros= socket(AF_INET,SOCK_DGRAM,0);
```

- bind(): Esta función sirve para asociar una dirección al socket. La estructura de esta función es la siguiente:

```
#include <sys/types.h>
#include <sys/socket.h>
int bind(int sockfd, struct sockaddr *my_addr, int addrlen);
```

Donde los argumentos que acepta la función son:

- sockfd: el socket retornado por la función "socket()" anteriormente descrita.
- My_addr: la dirección que se quiere asignar al socket.
- addrlen: el tamaño de la dirección "my_addr" en bytes.

Esta función retorna un 0 si ha sido ejecutada satisfactoriamente, si no, retorna un -1.

En nuestro caso, los argumentos que debemos de poner son, en sockdf socket_ros, el cual es el creado con la función socket(), y para los argumentos my_addres y addrlen debemos de crear una dirección del tipo de la familia del protocolo a usar, en este caso una dirección AF_INET, para realizar esto, se escribe el siguiente código en nuestro caso:

```
/*Creamos un socket UDP */
int socket_ros= socket(AF_INET,SOCK_DGRAM,0);

/*Creamos una dirección AF_INET*/
struct sockaddr_in dir_serv_socket;
dir_serv_socket.sin_family=AF_INET;
dir_serv_socket.sin_addr.s_addr=inet_addr(Dirección IP del PC);
dir_serv_socket.sin_port=htons(puerto por el que se establecerá la comunicación);
```

```
/* Ahora asociamos la dirección al socket*/
int variable = bind(socket_ros, (struct sockaddr *)&dir_serv_socket, sizeof (dir_serv_socket);

/* Vemos si se ha asociado correctamente*/
if (variable==0)
    puts("Asociado satisfactoriamente");
```

Una vez se ha creado el socket y asociado a una dirección, veamos cómo se envían y reciben datos a través del socket con las siguientes funciones:

- sendto(): Esta función permite escribir datos y especificar la dirección de destino del receptor al mismo tiempo. La estructura de esta función es la siguiente:

```
int sendto(int s, const void *msg, int len, unsigned flags, const struct sockaddr *to, int tolen);
```

donde los argumentos son los siguientes:

  - s: Este primer argumento es el número de socket, el cual se puede obtener de la función socket().
  - Msg: es un puntero a la memoria asignada que almacena el mensaje que se desea enviar, es decir, este es el mensaje que se desea enviar al receptor.
  - Len: Es el tamaño en bytes del mensaje msg.
  - Flags: permite especificar algunas opciones, en muchos casos siempre se puede sustituir por cero ya que eso indica que no se quiere ninguna opción especial.
  - To: es un puntero a la dirección genérica del socket que se ha establecido, esto es, la dirección del receptor del mensaje.
  - Tolen: es el tamaño de la dirección del argumento “to”.

En cuanto al robot virtual, el mensaje a enviar al receptor (Centro Remoto) son las velocidades leídas de la odometría del robot virtual, o lo que es lo mismo, los datos obtenidos del topic suscrito. Esta función quedaría de la forma:

```
sendto(socket_ros,(char *)&lectura_ros,sizeof(dato_ros),0,(struct
    sockaddr*)&dir_cli_socket,sizeof(dir_cli_socket));
```

siendo lectura_ros las velocidades lineal y angular leídas de la odometría a través del topic p3dx/odom al que estamos suscritos y que asignamos estos datos a lectura_ros en la función callback descrita en la pág 76. Los argumentos To y Tolen son los correspondientes al receptor, en este caso, el Centro Remonto.

En cuanto al robot real, se enviará al centro remoto las velocidades leídas por los encóders del robot.

En el caso del Centro Remoto, el mensaje a enviar tanto para el robot virtual como para el real, serán las velocidades de consigna que se le quiere enviar al robot.

- recvfrom(): Esta función permite recibir y especificar la dirección del remitente al mismo tiempo. El programa esperará en esta línea de código hasta que lleguen datos, dado que así es como funciona la aplicación cliente-servidor. La estructura de la función es la siguiente:

```
int recvfrom(int s, void *buf, int len, unsigned flags, struct sockaddr *from, int *fromlen);
```

  donde los argumentos son:

  - s: Este primer argumento es el número de socket, el cual se puede obtener de la función socket().
  - buf: es un puntero que apuntará a la memoria donde se guardarán los datos recibidos.
  - Len: el tamaño máximo en bytes del bufer "buf".
  - Flags: Para especificar opciones.
  - From: Apunta a la memoria de la dirección del socket remitente, es decir, será la dirección del remitente.
  - Fromlen: el tamaño máximo en bytes de la dirección del remitente.

  En cuanto al robot virtual, el mensaje a recibir por parte del remitente (Centro Remoto) son las velocidades de consigna que se le mandarán al robot. Esta función quedaría de la forma:

```
recvfrom(socket_ros,(char*)&datos_ros,sizeof(dato_ros),0,(struct sockaddr *)&dir_cli_socket,(socklen_t *)&long_dir_cli);
```

  siendo datos_ros las velocidades lineal y angular recibidas del Centro Remoto, las cuales se publicarán en el topic "p3dx/cmd_vel" para mandárselas al robot virtual de Gazebo. Los argumentos "from" y "fromlen" son los correspondientes al remitente, en este caso, el Centro Remoto.

  En cuanto al robot real, las velocidades a recibir son las enviadas por el Centro Remoto.

  En el caso del Centro Remoto, el mensaje a recibir serán las velocidades leídas por parte de la odometría en el topic correspondiente del robot virtual, y por parte de los encóders en el robot real.

Una vez se lleva a cabo la comunicación de datos, es necesario cerrar el socket para indicar que ya no hay más datos a enviar y/o recibir. Para ello utilizamos la función "close(socket)", donde socket es el valor obtenido de la función socket().

Por tanto queda definido que para la comunicación del PC centro remoto con el robot virtual y con el real, es necesaria la implementación de este socket, el cual lo implementaremos en la S-Function de Simulink del centro remoto, en el nodo de comunicación de ROS para la comunicación con el robot virtual y en la S-Function implementada en el robot real para la comunicación con el robot real.

3.5.2. S-Function

Una vez se diseña el algoritmo de control sobre el modelo en VVEE del robot P3DX, se pretende ejecutar en el robot virtual/real.

Parte de esta comunicación se basa en las "S-Function", que es un bloque de Simulink que dispone de entradas y salidas, y permite su programación en lenguaje C, Fortran o Matlab. Este bloque se encuentra en la librería "User-Defined Functions" de Simulink:

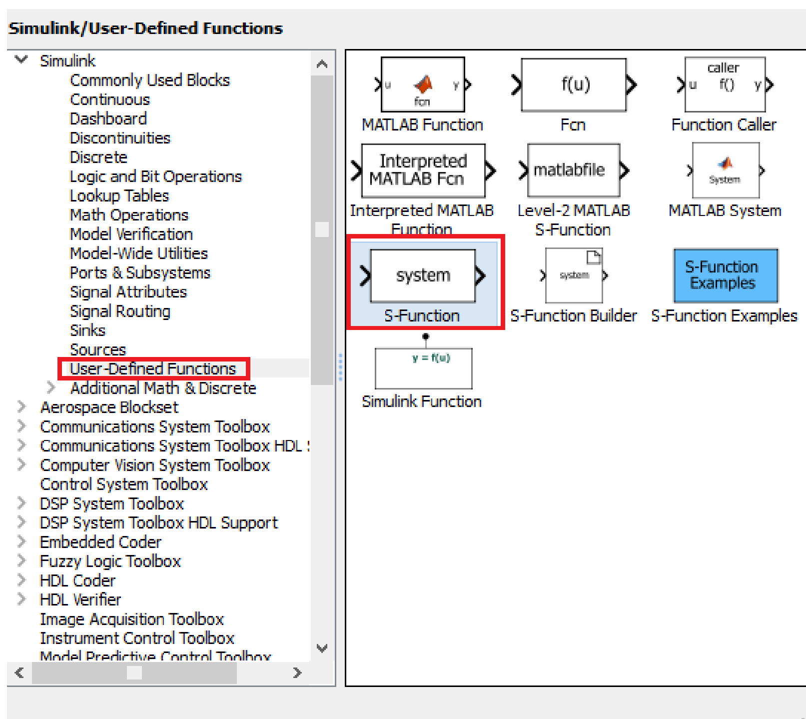


Ilustración 3.27: Librería de Simulink de la S-Function.

Por lo tanto, en el diagrama de bloques de Simulink de la Ilustración 3.24 basta con sustituir el bloque del modelo en VVEE del robot P3DX por este bloque S-Function quedando tal que:

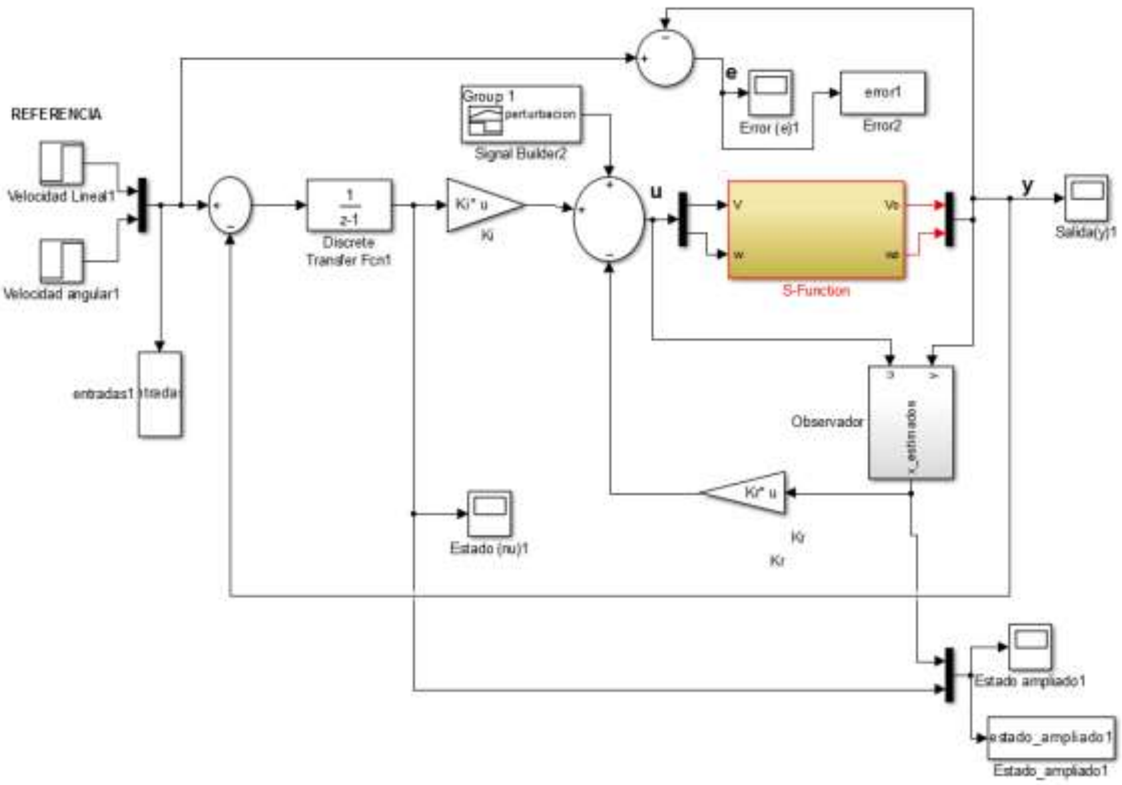


Ilustración 3.28: Diagrama de bloques de un Servosistema-estimador con S.Function en Simulink.

En la S-Function se implementará el socket descrito y se comunicará con el robot virtual y real únicamente cambiando la dirección IP a la que enviar y de la que recibir datos.

El código C que se debe de desarrollar para que Simulink pueda reconocerlo debe ser compilado como un archivo MEX, y su estructura mínima debe de ser la siguiente:

```
/*
S-FUNCTION simple
 */

#define S_FUNCTION_NAME  example
#define S_FUNCTION_LEVEL 2

#include "simstruc.h"
#include <fcntl.h>
#include <stdio.h>

/* Function: mdlInitializeSizes ===============================================
 * Abstract:
 *   Setup sizes of the various vectors.
 */
static void mdlInitializeSizes(SimStruct *S)
{
    ssSetNumSFcnParams(S, 0);
    if (ssGetNumSFcnParams(S) != ssGetSFcnParamsCount(S)) {
        return;
    }

    /* Configuracion de las entradas */
    if (!ssSetNumInputPorts(S, 1)) return;

    ssSetInputPortDataType(S, 0, SS_DOUBLE);
    ssSetInputPortWidth(S, 0, DYNAMICALLY_SIZED);

    /* Evitar error de lazo algebraico */
    ssSetInputPortDirectFeedThrough(S, 0, 0);

    /* Configuracion de las salidas */
    if (!ssSetNumOutputPorts(S,1)) return;

    ssSetOutputPortWidth(S, 0, DYNAMICALLY_SIZED);

    ssSetNumSampleTimes(S, 1);

    /* Configuracion optima para conectarse con dispositivos */
    ssSetOptions(S, SS_OPTION_PLACE_ASAP);
}


/* Function: mdlInitializeSampleTimes =========================================
 * Abstract:
 *    Specifiy that we inherit our sample time from the driving block.
 */
static void mdlInitializeSampleTimes(SimStruct *S)
{
    ssSetSampleTime(S, 0, 0.1); /* Periodo de muestreo */
    ssSetOffsetTime(S, 0, 0.0);
    ssSetModelReferenceSampleTimeDefaultInheritance(S);
}
```

```
#define MDL_START  /* Change to #undef to remove function */
#if defined(MDL_START)
  /* Function: mdlStart =====================================================
   * Abstract:
   *    This function is called once at start of model execution. If you
   *    have states that should be initialized once, this is the place
   *    to do it.
   */
  static void mdlStart(SimStruct *S)
  {
        /* Inicializacion de variables e hilos. Apertura de dispositivos. */

  }
#endif /*  MDL_START */

static void mdlOutputs(SimStruct *S, int_T tid)
{
 real_T  *y0 = ssGetOutputPortRealSignal(S,0); /* Se definen las salidas */

 /* Se definen las entradas */
 InputRealPtrsType uPtrs0 = ssGetInputPortRealSignalPtrs(S,0);

 #ifdef RT
   /* Acceso a hardware: unicamente cuando se ejecuta en tiempo real */
 #endif
}


/* Function: mdlTerminate =====================================================
 * Abstract:
 *    No termination needed, but we are required to have this routine.
 */
static void mdlTerminate(SimStruct *S)
{
  /* Cierre de dispositivos y destruccion de hilos */
  return;
}


#ifdef  MATLAB_MEX_FILE    /* Is this file being compiled as a MEX-file? */
#include "simulink.c"      /* MEX-file interface mechanism */
#else
#include "cg_sfun.h"       /* Code generation registration function */
#endif
```

Donde los métodos más importantes de la S-Function son los siguientes:

- mdlinitializeSizes: Especifica el número de entradas, salidas, estados, parámetros y otras características de la S-Function.
- mdlInitializeSampleTimes: Especifica los periodos de muestreo en los cuales la S-Function opera.
- mdlOutputs: Computa las señales que el bloque emite.
- mdlTerminate: Realiza cualquier operación requerida para la terminación de la simulación.

Para ver todos los métodos que se encuentran en la S-Function diríjase a [19].

En nuestro caso, las entradas y salidas se deben de configurar de la forma en la que:

- Las entradas sean las velocidades lineales y angulares del controlador
- Las salidas sean las velocidades lineales y angulares leídas de la odometría por el robot virtual/real (topic Odom en el caso del virtual, encóders en el caso del real).

En la siguiente figura se puede ver un diagrama que lo explica mejor:

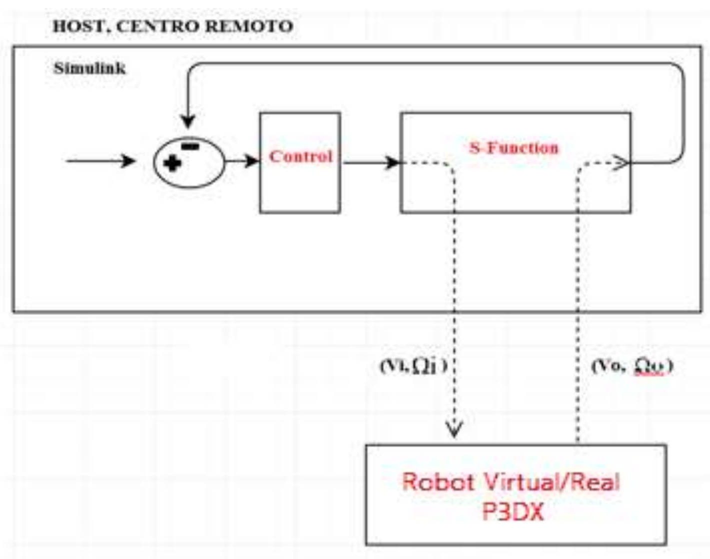


Ilustración 3.29: Comunicación con S-Function.

Por tanto, el socket descrito en el apartado 3.5.1 se deberá de implementar en la S-Function del centro remoto, y esta será igual tanto para comunicar con el robot real como con el virtual, cambiando únicamente la dirección IP dependiendo de con que robot queramos comunicar.

3.5.3. Diferencias en la comunicación entre el robot real y el virtual

Para ambos, la comunicación puede verse descrita en la siguiente ilustración:

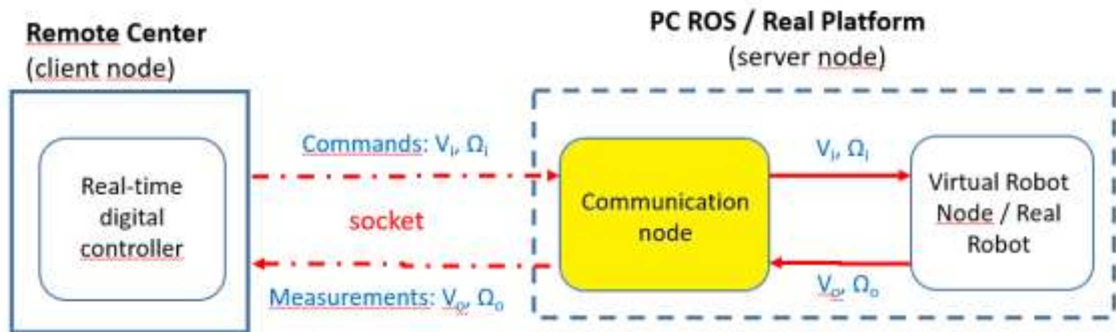


Ilustración 3.30: Comunicación entre centro remoto y robot real/virtual.

Donde el centro remoto se comunica con el nodo de comunicación y este a su vez con el robot virtualizado o con el real.

La única diferencia que existe reside en el nodo de comunicación (bloque amarillo en la Ilustración 3.30), es decir, en cómo el robot virtual/real maneja las velocidades recibidas por el centro remoto para enviarlas a las ruedas del robot, y en cómo leer las velocidades leídas de la odometría para enviarlas de nuevo al centro remoto.

3.5.3.1 Nodo de comunicación en robot virtual

El nodo de comunicación para el robot virtual es el descrito en el apartado 3.3.1.3, el cual es un nodo de ROS, y en él se utilizan los topics para enviar las velocidades recibidas del centro remoto a las ruedas del robot, y para obtener las velocidades leídas por el topic Odom del robot para así, enviarlas al centro remoto a través del socket.

Quedando lo siguiente:

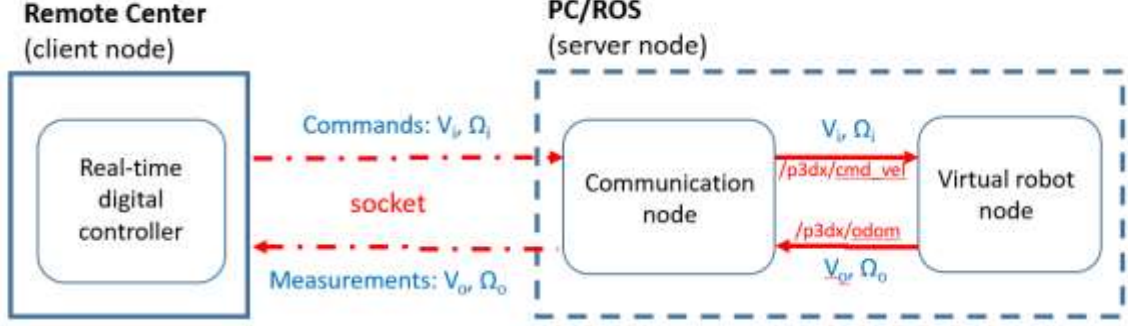


Ilustración 3.31: Comunicación completa entre Centro Remoto y Robot Virtual.

La comunicación completa entre centro remoto y robot virtual se describe en los siguientes pasos:

- 1º: El nodo de comunicación (lanzando_vel de la Ilustración 3.31) recibe los comandos de velocidad lineal y angular del centro remoto a través del socket.
- 2º: Este mismo nodo publica estas velocidades en el topic p3dx/cmd_vel del modelo P3DX para mover las ruedas del robot virtual.
- 3º: Este nodo se subscribe también al topic p3dx/odom obteniendo las velocidades medidas del robot virtual.
- 4º: A su vez, el nodo las envía al centro remoto a través del ya mencionado socket.

3.5.3.2. Nodo de comunicación en robot real

El nodo de comunicación para el robot real tiene que ser capaz de una vez recibidas las velocidades (alto nivel) a través del socket del centro remoto, traducirlas en comandos (bajo nivel) tal que el robot real pueda interpretarlas para mover la rueda izquierda (wl) y la rueda derecha (wr). Lo mismo con las velocidades leídas por los encóders, el nodo de comunicación tiene que ser capaz de transformar estas lecturas que vendrán del robot como comandos, a velocidades que el centro remoto pueda interpretar para realizar el control.

Existen dos interfaces con los cuales podemos hacer la transformación de velocidades del centro remoto (alto nivel) a comandos del robot real (bajo nivel) con el objetivo de poder conectar el sistema de control desarrollado en el centro remoto con el robot P3DX real.

Estos interfaces se denominan Player y ARCOS:

- **Player**

Es una mera interfaz, o lo que es lo mismo, un traductor entre protocolos. La librería Player facilita la comunicación entre un programa de alto nivel y el robot P3DX. En ámbitos generales el funcionamiento de esta interfaz es el siguiente:

1. Se inicializa el Player cliente que establece comunicación con el Player servidor haciendo uso de la librería de sockets (descrita en el apartado 3.5.1). El player cliente encapsula los comandos proporcionados por el control y se los envía al servidor, y cuando el control lo solicita, el cliente recoge los datos recibidos desde el servidor y se los proporciona al primero.
2. Respecto al Player servidor, es una interfaz entre sockets y el protocolo ARCOS que define los comandos de escritura en el propio robot P3DX a bajo nivel. Cuando el servidor recibe comandos desde el cliente, los encapsula en un paquete ARCOS y son enviados al controlador de bajo nivel implementado en el robot real. A su vez el servidor también recibe datos provenientes del P3DX, los cuales

son convertidos por el microcontrolador del protocolo ARCOS a sockets para enviárselos al Player Cliente.

A continuación se muestra una ilustración con el proceso:

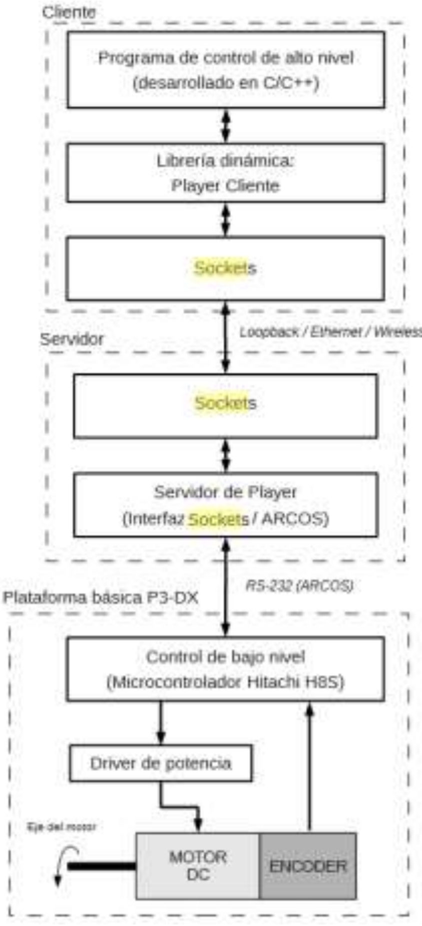


Ilustración 3.32: Interfaz entre un programa de control y el robot P3DX utilizando Player.

Por ejemplo, el servicio "position2d" en la interfaz Player provee de la información necesaria para realizar un control de velocidad del robot mediante lectura de encóders y escritura de comandos de velocidad. Para ello es necesario crear el servicio y posteriormente subscribirse a él como sigue:

```
position2d = playerc_position2d_create(client, 0);
```

```
playerc_position2d_subscribe(position2d, PLAYERC_OPEN_MODE);
```

Para poder ver más en profundidad esta interfaz así como sus comandos diríjase a la referencia [20].

- **ARCOS**

ARCOS es el protocolo de comunicación que utiliza la plataforma robótica real P3DX para comunicarse directamente a través del puerto serie. Este protocolo de comunicación se explica detalladamente en el manual del robot [21] y ha sido desarrollado por MobileRobots.

Como vimos en la Ilustración 3.32, este protocolo es utilizado por la librería Player para permitir la comunicación entre el programa cliente y el robot donde los comandos que se dirigen al robot y la información sensorial proveniente de éste, pasan por varias etapas de adaptación de protocolos (para un desarrollo a más alto nivel), generando un aumento de la incertidumbre del tiempo de llegada de la información.

Debido a que se está ejecutando un programa en tiempo real, interesa reducir esta incertidumbre, por lo que utilizando únicamente el protocolo ARCOS eliminamos las etapas intermedias de interfaz utilizadas por Player, pero por el contrario se deberá de trabajar a más bajo nivel. A continuación se presenta la nueva relación entre el programa de control y la planta real utilizando el driver correspondiente únicamente usando el protocolo ARCOS sin pasar por la interfaz Player:

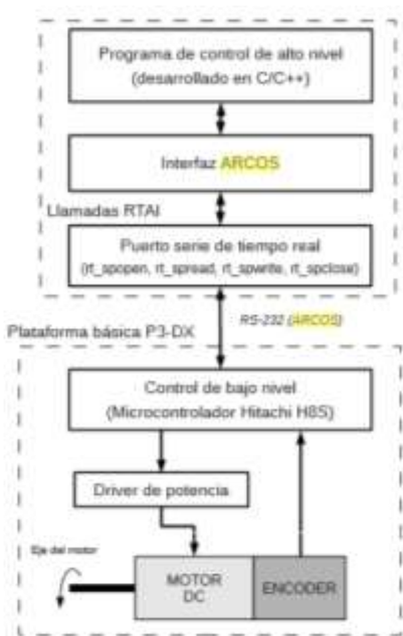


Ilustración 3.33: Interfaz entre un programa de control y el robot P3DX utilizando ARCOS.

Esta interfaz no es más que un conjunto de funciones que se encargan de generar paquetes con una determinada estructura para enviarlos por el puerto serie, y cuando corresponda, recibir los paquetes provenientes del robot para decodificar el contenido de los mismos.

Como se observa en la Ilustración 3.33, la interfaz ARCOS hace uso de llamadas al sistema operativo de tiempo real, las cuales están incluidas en el kernel modificado de Linux RTAI. Estas funciones son: rt_spopen, rt_spread, rt_spwrite y rt_spclose. Dichas funciones son la interfaz con el módulo de tiempo real rt_serial, ubicado en espacio de Kernel, el cual permite hacer uso del periférico de comunicación serie, respetando la prioridad del proceso que la llama y retornando el control al mismo cuando se ha cumplido la misión. Al estar utilizando RTAI, estas funciones tendrán un tiempo de respuesta e incertidumbre menor que las funciones clásicas que llaman al sistema operativo Linux (open, read, write y close) debido a que los procesos de tiempo real linkados con las librerías de RTAI tienen una prioridad mayor como se dijo en la sección 2.5.

El detalle de funcionamiento global de la interfaz ARCOS se divide en tres bloques fundamentales que son:

- P2OS_MainSetup: Permite la conexión al servidor.
- P2OS_MainProcess: Proceso principal para lectura y escritura del robot.
- P2OS_MainQuit: Permite la desconexión del robot.

Este funcionamiento está detallado en la siguiente ilustración:

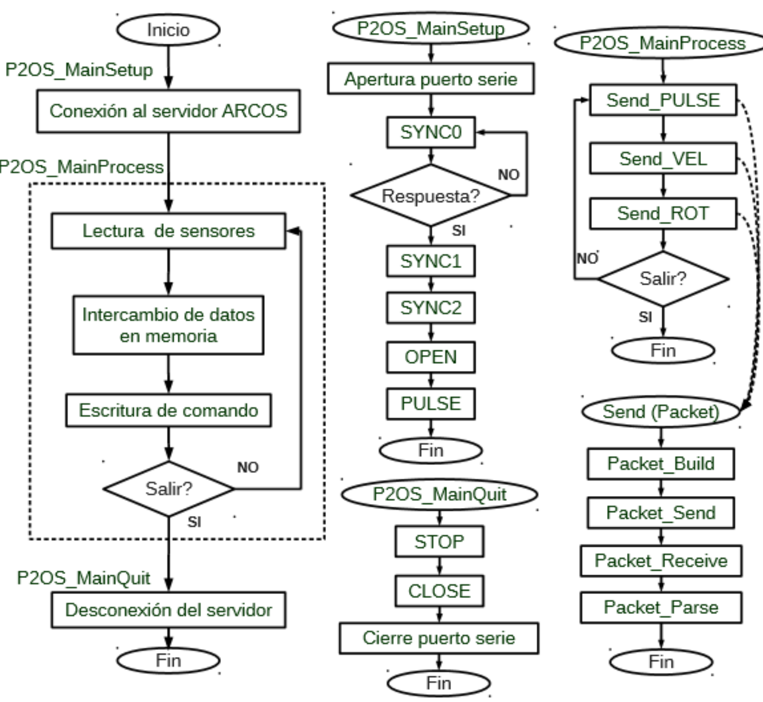


Ilustración 3.34: Detalle del funcionamiento interno de la interfaz ARCOS.

- **Diferencia entre protocolos y elección del mismo**

Cabe destacar que en este trabajo no se ha realizado el driver desde cero, si no que se ha utilizado el driver creado con el protocolo ARCOS por parte del grupo GEINTRA ya que se lleva trabajando con él durante años y está más que probado. A la hora de la elección de este Driver, se ha tomado como guía la referencia [10], en el que se hicieron estudios de los mismos, y en los cuales podemos ver la diferencia entre el interfaz Player y ARCOS en las siguientes figuras:

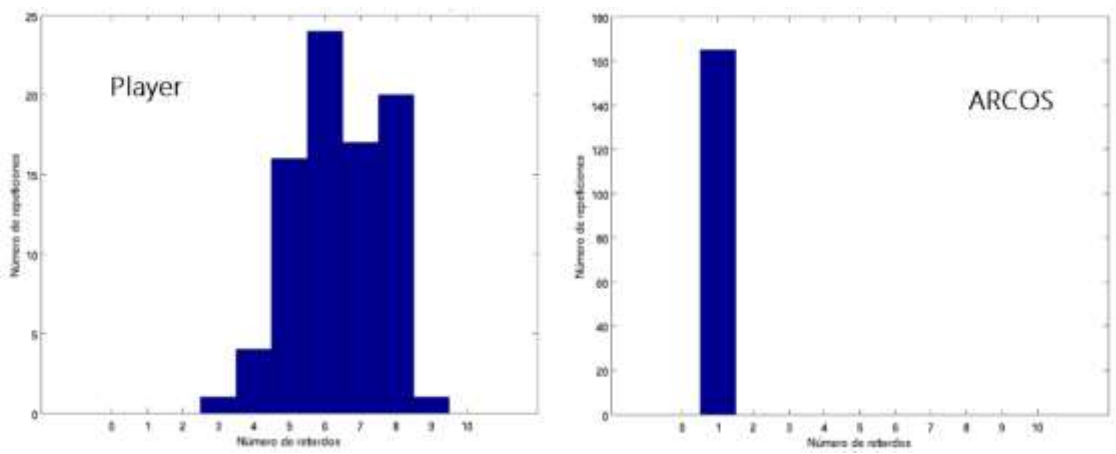


Ilustración 3.35: Análisis de la incertidumbre utilizando el driver Player (izquierda) y ARCOS (derecha) con un Ts=100ms.

En estas gráficas sacadas de la referencia [10], se visualiza un estudio estadístico entre Player y ARCOS, en los que se ha sometido al robot a un tren de pulsos periódico evaluando en cada flanco de subida la consigna, el número de muestras que se tarda en proporcionar una muestra no nula a la salida, y como podemos observar, la incertidumbre de llegada de la información es mayor usando el interfaz Player. Por tanto a priori es mejor utilizar el protocolo ARCOS directamente.

Además, observando la siguiente figura donde se muestra una ampliación de la evolución temporal de la consigna de velocidad lineal, la respuesta del robot utilizando Player y la misma repuesta utilizando ARCOS, se demuestra que el driver Player incluye varios retardos adicionales en la comunicación respecto al driver ARCOS:

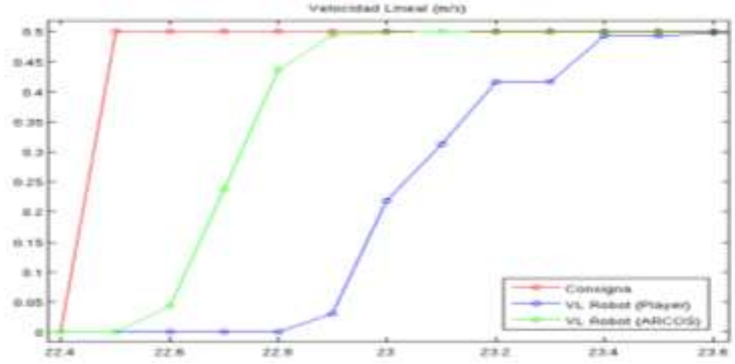


Ilustración 3.36: Comparación de la respuesta temporal de la velociddad lineal del robot real utilizando el driver Player y ARCOS.

Por tanto, con esta información se justifica la elección del driver ARCOS para aplicaciones de control.

Una vez explicado el driver a utilizar y como funciona, este tiene que ser implementado en el robot real. Para conseguir esta implementación hacemos uso de las S-Function. Esta S-Function tendrá asociado un fichero en C con la implementación del driver ARCOS:

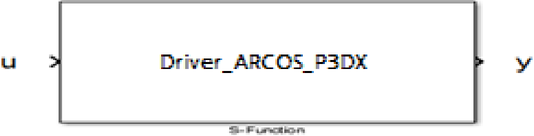


Ilustración 3.37: S-Function con el driver creado mediante la interfaz ARCOS.

Y para poder implementarla en el robot real, basta con crear un ejecutable siguiendo el procedimiento descrito en el anexo 0 y cargarlo en la tarjeta VIA-EPIA del robot real a través de la función ssh del terminal de simulink.

Quedando la comunicación entre el centro remoto y el robot real como sigue:

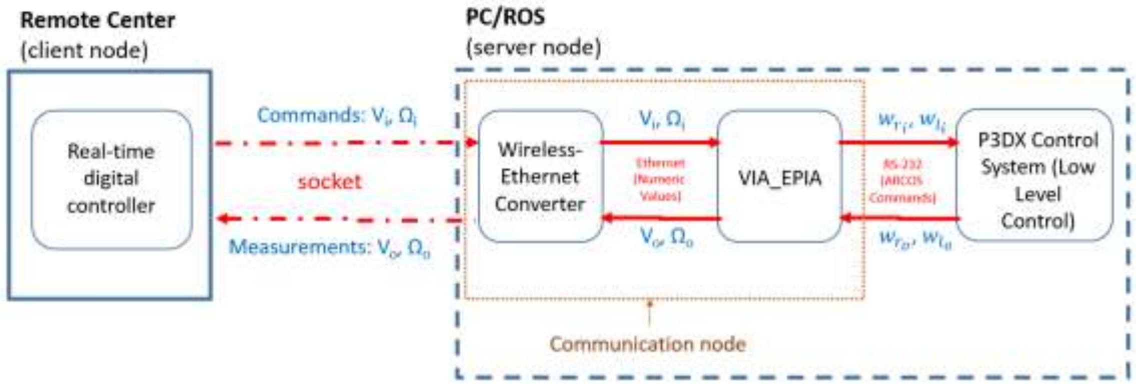


Ilustración 3.38: Comunicación completa entre Centro Remoto y Robot Real.

Donde el procedimiento seguido por el sistema es el siguiente:

1. El centro remoto envía vía wirelles a través de un socket las velocidades lineal y angular al Wirelless-Ethernet converter.
2. El conversor Wireless-Ethernet convierte esta señal Wireless en Ethernet.
3. Llegan las velocidades lineal y angular a la tarjeta VIA-EPIA, en la cual está implementado el driver ARCOS descrito en pág 100 en una S-Function, donde se traducen estas velocidades a comandos ARCOS para que el robot las entienda.
4. La tarjeta envía estos comandos a través de RS-232 al sistema de control de bajo nivel del P3DX para mover las ruedas (w_{r_i}, w_{l_i}).
5. A su vez, el robot P3DX envía las velocidades leídas por la odometría, pasando por los mismos componentes y procedimiento, hasta llegar al centro remoto.

3.6. Procedimiento a seguir para ejecutar la solución de control del Centro Remoto en Robot Virtual/Real

3.6.1. En Robot Virtual

Ya se han explicado tanto la programación en el ordenador que simula el robot virtual (ROS/Gazebo), la programación en el Centro Remoto (Matlab/Simulink/RTW) así como la comunicación entre ambos, por lo que ahora se explicará paso por paso el procedimiento a seguir para implementar el algoritmo de control realizado en el centro remoto en el robot virtual existente en el PC ROS/Gazebo:

1- En el PC ROS/Gazebo, se lanzan los dos nodos en ROS, el que permite la comunicación con el Centro Remoto y con el robot en Gazebo (nodo “lanzando_vel”), y el nodo que simula el P3DX en gazebo. Una vez lanzado el nodo de comunicación, este PC esperará a recibir velocidades provenientes del centro remoto dado que está programado mediante sockets (aplicaciones cliente-servidor).

2- En el Centro Remoto, se crea un ejecutable de la solución final de control con la S-Function siguiendo el procedimiento descrito en el anexo 0, en la cual está implementado el socket que permite la comunicación.

3- En el Centro Remoto, se lanza ese ejecutable indicando los argumentos –v –f tf, siendo tf el tiempo de simulación. Una vez lanzado, se empezarán a transmitir datos entre el Centro Remoto y el PC ROS/Gazebo, y veremos como el robot virtualizado se empieza a mover siguiendo las velocidades mandadas.

4- Una vez finalizada la simulación, los datos son recogidos en un .mat, los cuales los podemos visualizar y ver si el algoritmo de control y la simulación son los correctos, si no es así, se harían cambios en el algoritmo de control y/o en el modelo de Gazebo para conseguir lo que se desee.

Los resultados obtenidos para un ejemplo en concreto así como la comparación entre Robot Virtual y Robot Real se detallarán en el capítulo 6.

3.6.2. En Robot Real.

La idea general para la aplicación del algoritmo de control en el robot real es idéntica a la del robot virtual, pero ahora, cambiando la IP del socket para indicar que se va a realizar la comunicación con el robot real.

Este procedimiento es muy similar al explicado para el robot virtual. Los pasos a seguir son los siguientes:

1. Correr el nodo de comunicación (ejecutable implementado en el robot real, el cual tiene el protocolo ARCOS. Para ello:

    a. Implementar la S-Function con el driver ARCOS y generar el ejecutable siguiendo el procedimiento explicado en el anexo 0.

    b. Una vez tenemos el ejecutable, debemos de copiarlo en el target (tarjeta VIA-EPIA implementado en el robot real). Esto es una tarea simple gracias a que Linux dispone de un comando para copiar ficheros a través de la red. A continuación se muestra un ejemplo para transferir el fichero ejecutable generado (example) a la unidad móvil real ejecutando el siguiente comando:

```
host:/# scp /..../Desarrollo/02_RTAI_Example/test root@cove1:/home/cove/target
```

2. Una vez tenemos el ejecutable de tiempo real en la plataforma robótica real, debemos de ejecutarlo para que se implemente en la unidad target (robot real). Para ello es necesario iniciar sesión de manera remota a través del comando ssh, y una vez en el target, ejecutarlo con los argumentos –v –f tf:

```
host:/# ssh cove1

target:/# ./home/cove/target/test -v -f 20
```

3. Ahora, el robot real estará esperando a que el centro remoto le envíe datos a través del socket. Para ello, generamos un ejecutable con la solución de control donde habrá una S-Function con la implementación socket, y corremos este ejecutable desde el centro remoto.

4. Ahora se estarán enviando datos entre el centro remoto y el robot real, pudiendo ver que el robot real se empieza a mover.

5. Una vez se ha terminado de ejecutar el algoritmo en el robot real, se deben de copiar los resultados obtenidos al ordenador host, o lo que es lo mismo, al centro remoto para su posterior análisis. Para conseguirlo basta con utilizar el mismo comando del paso 3:

```
host:/# scp root@cove1:/home/cove/target/*.mat /ruta/..../
```

6. Una vez se copian los resultados en el host, tendremos un archivo .mat el cual se puede cargar en el workspace y analizar los datos ploteándolos en Matlab.

Hecho el procedimiento y analizado los datos, podemos decidir si la solución de control implementada es la adecuada para lo que se quiere, si no es así, se recomienda hacer cambios en ella y ejecutarla en el robot virtual hasta que se adecue a lo que se quiere, y una vez conseguido, implementarla en el robot real.

Los resultados obtenidos para un ejemplo en concreto así como la comparación entre robot virtual y robot real se detallarán en el capítulo 6.

Capítulo 4: Extensión del trabajo a un convoy de robots

A lo largo del capítulo 3 se ha presentado y explicado el proceso a seguir para la creación de un modelo virtual del robot P3DX con su cinemática, dinámica y físicas para que tanto su apariencia como su comportamiento sea lo más cercano al robot que se tiene en el grupo GEINTRA, más concretamente, en el OL3 donde se trabaja con dicha plataforma.

En dicho capítulo, se explicó todo lo relacionado a ROS/Gazebo que se debe de saber para la realización de la aplicación, así como lo relacionado a Matlab/Simulink y a la comunicación entre Centro Remoto – Robot Virtual – Robot Real.

En este capítulo se va a extender este trabajo, haciendo que ahora sea un convoy de robots tanto virtual como real a los que tengamos que aplicar la solución de control en vez de a un único robot. Se explicará las modificaciones que se deben de hacer tanto en ROS/Gazebo como en Matlab/Simulink para conseguir el objetivo fijado.

A continuación se describe la estructura de este capítulo:

- Virtualización física y funcional de un convoy de robots P3DX.

- Descripción del algoritmo de control para el guiado de robots P3DX en convoy.

- Comunicación entre PC Centro remoto – PC Convoy virtual/real comentando las diferencias entre convoy virtual y real.

- Procedimiento a seguir para ejecutar la solución de control del PC centro remoto en convoy virtual y convoy real.

4.1. Virtualización física y funcional de un convoy de robots P3DX

El convoy virtual correrá en el mismo PC en el que corría solamente un robot, pero esta vez con más robots virtuales.

En este apartado se explican las modificaciones a realizar en el PC que virtualiza el convoy, o lo que es lo mismo, en la herramienta ROS/Gazebo, para conseguir la virtualización física y funcional de un convoy de robots P3DX.

El convoy virtual está compuesto por robots P3DX virtuales idénticos al realizado en el apartado 3.3.1.1.

Por lo tanto, se debe conseguir un convoy de robots lanzados en el entorno del pasillo Oeste de la primera planta de la Escuela Politécnica Superior de Alcalá realizado en el apartado 3.3.1.2 y poder enviarles velocidades lineal y angular, así como recibirlas de los propios robots leyendo de su odometría, a cada uno de ellos.

Para conseguir un convoy virtual, vamos a seguir los siguientes pasos:

1. **Duplicación de paquetes**

Una vez tenemos creado un robot virtual como paquete en el directorio de ROS, este se tiene que duplicar tantas veces como robots queramos en el convoy. Por ejemplo, si queremos 4 robots, deberemos de replicar el paquete creado 4 veces, teniendo estos distintos nombres. En la siguiente imagen se puede observar:

Ilustración 4.1: Paquetes de robots P3DX para la generación de un convoy.

Como podemos ver en el recuadro rojo, están los paquetes de los robots, en este caso 4 robots, los cuales son idénticos en apariencia y comportamiento.

Por lo tanto, necesitamos tener tantos paquetes como robots queremos en el convoy, y para ello basta con duplicarlos con distintos nombre, y hacer algunas modificaciones que se detallan en el siguiente paso.

2. Modificación en los topics de los paquetes

Una vez tenemos el convoy, debemos de enviar y recibir velocidades a cada uno de los robots, para ello, estos tienen que tener topics de nombres diferentes, ya que si no es así, ROS/Gazebo no sabría a qué robot pertenece cada velocidad lineal y angular.

Para ello, en los archivos de descripción de los robots donde aparecen los nombres de los topics (véase pág 72) deben de ser cambiados para que cada robot tenga un nombre de topics específico. Por ejemplo para dos robots tenemos:

Robot 1 →

```
<gazebo>
    <plugin name="differential_drive_controller" filename="libgazebo_ros_diff_drive.so">
        <alwaysOn>true</alwaysOn>
        <updateRate>100</updateRate>
        <leftJoint>base_right_wheel_joint</leftJoint>
        <rightJoint>base_left_wheel_joint</rightJoint>
        <wheelSeparation>0.39</wheelSeparation>
        <wheelDiameter>0.15</wheelDiameter>
        <torque>5</torque>
        <commandTopic>$p3dx/cmd_vel</commandTopic>
        <odometryTopic>$p3dx/odom</odometryTopic>
        <odometryFrame>odom</odometryFrame>
        <robotBaseFrame>base_link</robotBaseFrame>
    </plugin>
</gazebo>
```

Robot 2 →

```
<gazebo>
    <plugin name="differential_drive_controller" filename="libgazebo_ros_diff_drive.so">
        <alwaysOn>true</alwaysOn>
        <updateRate>100</updateRate>
        <leftJoint>base_right_wheel_joint</leftJoint>
        <rightJoint>base_left_wheel_joint</rightJoint>
        <wheelSeparation>0.39</wheelSeparation>
        <wheelDiameter>0.15</wheelDiameter>
        <torque>5</torque>
        <commandTopic>$p3dx2/cmd_vel</commandTopic>
        <odometryTopic>$p3dx2/odom</odometryTopic>
        <odometryFrame>odom</odometryFrame>
        <robotBaseFrame>base_link</robotBaseFrame>
    </plugin>
</gazebo>
```

Como podemos observar, cada robot tiene un nombre de topic diferente, donde se diferencia claramente cuáles son los topics para cada robot.

Para que no sea un trabajo engorroso, se puede poner una etiqueta al principio de cada archivo donde se describen los topics tal que, por ejemplo para el robot 2:

```
<property name="robot" value="p3dx2"/>
...
<commandTopic>${robot}/cmd_vel</commandTopic>
```

Así solo tendremos que cambiar el valor de la etiqueta para identificar el robot que es.

Una vez cambiados los topics y lanzado el convoy al mundo de Gazebo, por ejemplo 4 robots, podemos ver los topics activos mediante el comando rostopic list, teniendo:

```
alvaro@alvaro-ubuntu-PC:~$ rostopic list
/clock
/gazebo/link_states
/gazebo/model_states
/gazebo/parameter_descriptions
/gazebo/parameter_updates
/gazebo/set_link_state
/gazebo/set_model_state
/p3dx/base_pose_ground_truth
/p3dx/cmd_vel
/p3dx/laser/scan
/p3dx/odom
/p3dx2/base_pose_ground_truth
/p3dx2/cmd_vel
/p3dx2/laser/scan
/p3dx2/odom
/p3dx3/base_pose_ground_truth
/p3dx3/cmd_vel
/p3dx3/laser/scan
/p3dx3/odom
/p3dx4/base_pose_ground_truth
/p3dx4/cmd_vel
/p3dx4/laser/scan
/p3dx4/odom
/rosout
/rosout_agg
/tf
```

Robot 1
Robot 2
Robot 3
Robot 4

Ilustración 4.2: Topics pertenecientes a los diferentes robots virtuales del convoy.

3. <u>Lanzando el convoy al entorno en Gazebo</u>

Una vez tenemos los robots del convoy como paquetes y configurados sus topics, es hora de lanzarlos al entorno Gazebo creado en el apartado 3.3.1.2.

En la pág 69 se vió como lanzar un único robot al mundo de Gazebo, o lo que es lo mismo, como lanzar este paquete. Por tanto vamos a seguir este código para lanzar cada uno de los paquetes de los robots para así formar un convoy, que para nuestro ejemplo será un convoy de 4 robots.

Por tanto debemos de lanzar los 4 paquetes indicados en la Ilustración 4.1 identificando para cada uno de ellos en el fichero .launch, el nombre que tendrá cada robot en el simulador Gazebo, el nombre del paquete al que pertenecen y la posición y orientación del robot.

En la siguiente ilustración se observa cómo se lanzan 4 robots en Gazebo con las características que se deben indicar para cada robot en color morado y en rojo el valor cambiado:

Robot1

```
<!-- Load the URDF Robot 1 into the ROS Parameter Server -->
        <param name="robot_description"    Nombre del robot en Gazebo
               command="$(find xacro)/xacro.py '$(find p3dx_description)/urdf/pioneer3dx.xacro'" />
                                                                                    Paquete
        <!-- Run a python script to the send a service call to gazebo_ros to spawn a URDF robot 1-->
        <node name="urdf_spawner" pkg="gazebo_ros" type="spawn_model"
                respawn="false" output="screen" args="-x 0 -y 0 -z 0 -Y 3.141592654 -urdf -model
p3dx -param robot_description" />                              Posición y orientación
```

Robot2

```
        <!-- Load the second URDF into the ROS Parameter Server -->
        <param name="robot_description2"
              command="$(find xacro)/xacro.py '$(find 3dx2_description)/urdf/pioneer3dx2.xacro'" />

    <!-- Run a python script to the send a service call to gazebo_ros to spawn  the second URDF robot-->
        <node name="urdf_spawner2" pkg="gazebo_ros" type="spawn_model"
                respawn="false" output="screen" args="-x 1 -y 0 -z 0 -Y 3.141592654 -urdf -model
p3dx2 -param robot_description2" />
```

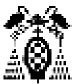
Robot3

```
        <!-- Load the third URDF into the ROS Parameter Server -->
        <param name="robot_description3"
             command="$(find xacro)/xacro.py '$(find p3dx3_description)/urdf/pioneer3dx3.xacro'" />

    <!-- Run a python script to the send a service call to gazebo_ros to spawn the third URDF robot -->
        <node name="urdf_spawner3" pkg="gazebo_ros" type="spawn_model"
                respawn="false" output="screen" args="-x 2 -y 0 -z 0 -Y 3.141592654 -urdf -model
p3dx3 -param robot_description3" />
```

Robot4

```
        <!-- Load the fourth URDF into the ROS Parameter Server -->
        <param name="robot_description4"
             command="$(find xacro)/xacro.py '$(find p3dx4_description)/urdf/pioneer3dx4.xacro'" />

    <!-- Run a python script to the send a service call to gazebo_ros to spawn  the second URDF robot-->
        <node name="urdf_spawner4" pkg="gazebo_ros" type="spawn_model"
                respawn="false" output="screen" args="-x 3 -y 0 -z 0 -Y 3.141592654 -urdf -model
p3dx4 -param robot_description4" />
```

Ilustración 4.3: Estructura .launch para lanzar un convoy de 4 robots en Gazebo.



Junto con esto, escribiendo el código para lanzar el entorno Gazebo del pasillo Oeste como se indicó en la pág 69 y lanzando el archivo .launch desde el terminal con el comando "roslaunch", podemos ver el convoy de robots en el entorno creado:

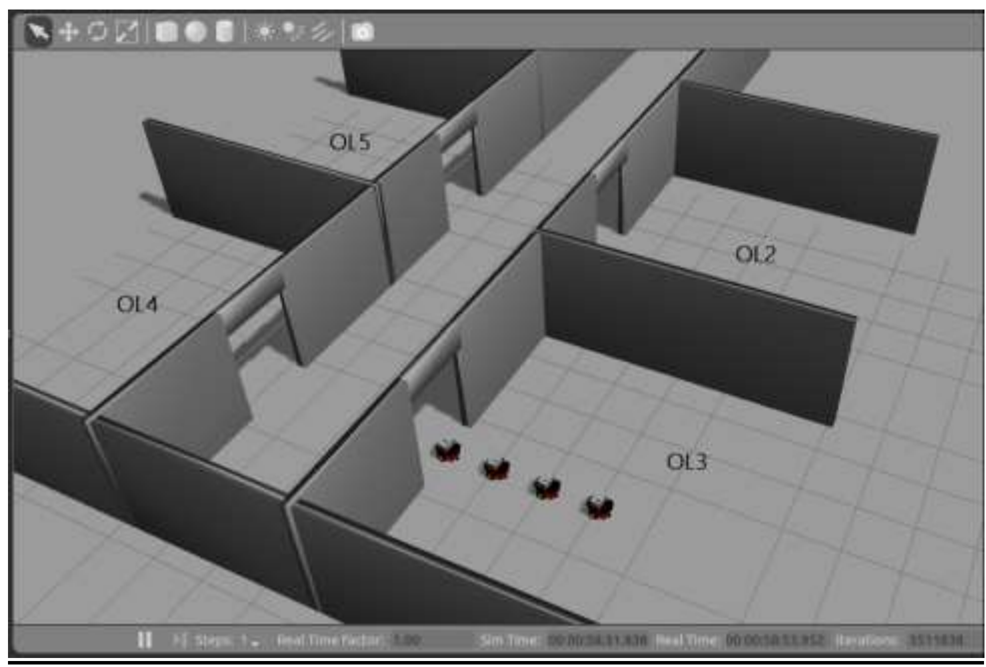


Ilustración 4.4: Convoy de 4 robots P3DX en el entorno Gazebo.

Para una mejor identificación de los robots se ha añadido a cada caja un número identificativo tal que:

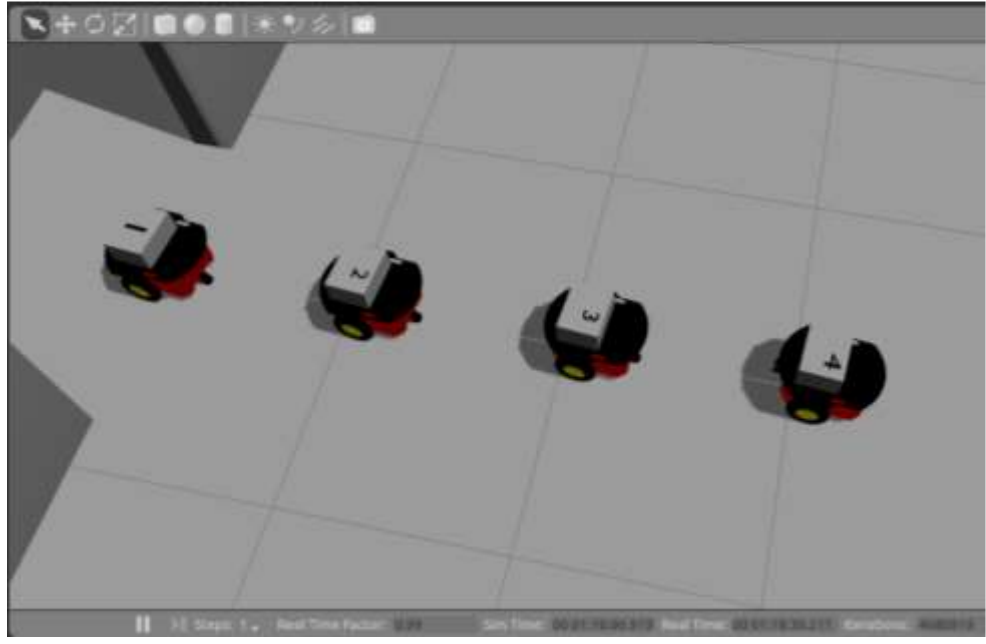


Ilustración 4.5: Convoy de 4 robots con cajas identificativas.

Donde podemos identificar cada uno de los robots únicamente viendo el número adherido en la caja gris.

4. Enviar y recibir velocidades de cada robot perteneciente al convoy, virtualización funcional

Para enviar y recibir velocidades de cada uno de los robots, debemos de crear un nodo como el del apartado 3.3.1.3, donde enviábamos y recibíamos velocidades de un único robot.

Los cambios que se deben de realizar para ahora comunicarnos con un convoy (de 4 robots para el ejemplo), son los siguientes:

- Publicar en los 4 topics de velocidad

Ahora, en vez de publicar velocidades en un solo topic (un robot) deberemos de publicar velocidades en 4 topics (4 robots).

Para ello deberemos de crear 4 publicaciones, una para cada topic, o lo que es lo mismo, una para cada robot:

```
ros::Publisher vel_pub_ = n.advertise<geometry_msgs::Twist>("p3dx/cmd_vel", 1);
ros::Publisher vel_pub_2 = n.advertise<geometry_msgs::Twist>("p3dx2/cmd_vel", 1);
ros::Publisher vel_pub_3 = n.advertise<geometry_msgs::Twist>("p3dx3/cmd_vel", 1);
ros::Publisher vel_pub_4 = n.advertise<geometry_msgs::Twist>("p3dx4/cmd_vel", 1);
```

Estas velocidades que se quieren publicar vendrán del centro remoto a través de un socket, el cual se describió en el apartado 3.5, donde las velocidades vendrán en una estructura, la cual contendrá las variables "lineal" para la velocidad lineal y "angular" para la velocidad angular, y serán arrays de 4 datos, un dato para cada robot. La declaración de la estructura es la siguiente:

```
typedef struct
{
unsigned long data_number=0;
double linear[4];
double angular[4];
}dato_ros;
```

Por tanto a partir de la funcion revcfrom() recibiremos los datos de la estructura dentro del loop principal, y estos datos los publicaremos en cada robot de la forma:

Recibir datos_ros del Centro Remoto a través del socket

```
recvfrom(socket_ros,(char*)&datos_ros,sizeof(dato_ros),0,(struct sockaddr
*)&dir_cli_socket,(socklen_t *)&long_dir_cli);
```

Publicar velocidades en Robot 1

```
geometry_msgs::Twist cmd;
cmd.linear.x = datos_ros.linear[0];
cmd.linear.y = 0;
cmd.linear.z = 0;

cmd.angular.x = 0;
cmd.angular.y = 0;
cmd.angular.z = datos_ros.angular[0];
vel_pub_.publish(cmd);
```

En Robot 2

```
cmd.linear.x = datos_ros.linear[1];
cmd.linear.y = 0;
cmd.linear.z = 0;

cmd.angular.x = 0;
cmd.angular.y = 0;
cmd.angular.z = datos_ros.angular[1];
vel_pub_2.publish(cmd);
```

En Robot 3

```
cmd.linear.x = datos_ros.linear[2];
cmd.linear.y = 0;
cmd.linear.z = 0;

cmd.angular.x = 0;
cmd.angular.y = 0;
cmd.angular.z = datos_ros.angular[2];
vel_pub_3.publish(cmd);
```

En Robot 4

```
cmd.linear.x = datos_ros.linear[3];
cmd.linear.y = 0;
cmd.linear.z = 0;

cmd.angular.x = 0;
cmd.angular.y = 0;
cmd.angular.z = datos_ros.angular[3];
vel_pub_4.publish(cmd);
```

La explicación del código para cada robot se incluyó en el apartado 3.3.1.3.

Como podemos ver, en el robot 1 publicamos las velocidades provenientes del socket en el elemento [0] del array de la velocidad lineal y angular, para el robot 2 es el elemento [1], para el 3 es el elemento [2], y por último para el robot 4 el elemento [3].

Con esto descrito ya podemos publicar velocidades a cada uno de los robots del convoy, donde estas velocidades provienen del centro remoto cuya comunicación con él se realiza a través de un socket, tal y como se describió a lo largo del apartado 3.5

Ahora vamos a ver como recibir velocidades de cada robot.

- <u>Recibir datos leídos por la odometría en los topics de cada robot</u>

Para recibir datos leídos por la odometría de cada robot debemos de subscribirnos al topic de cada uno de ellos, por lo tanto, necesitamos tantos subscriptores como robots tengamos. En el caso de 4 robots:

```
ros::Subscriber odom_sub = n.subscribe("p3dx/odom", 1000, velsubcallback);
ros::Subscriber odom_sub = n.subscribe("p3dx2/odom", 1000, velsubcallback);
ros::Subscriber odom_sub = n.subscribe("p3dx3/odom", 1000, velsubcallback);
ros::Subscriber odom_sub = n.subscribe("p3dx4/odom", 1000, velsubcallback);
```

Estos datos también viajarán al centro remoto a través de una estructura del mismo tipo que la que hemos descrito para datos_ros. Esta estructura se llamará lectura_ros.

Tanto publicar como subscribirse a un topic se puede programar en un mismo nodo como se vió para el caso de un solo robot (vease pág 77), ya que un nodo puede publicar y subscribirse a varios topics a la vez, quedando en este caso el siguiente diagrama de nodos y topics:

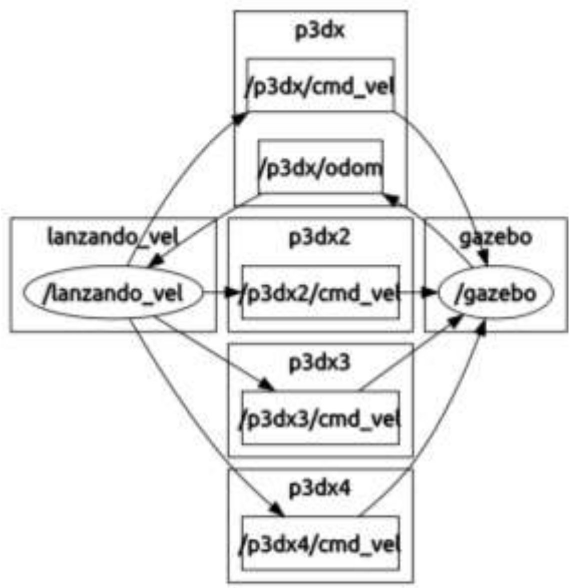


Ilustración 4.6: Nodo publicando velocidades y subscribiéndose a los datos de un convoy de 4 robots.

En la ilustración anterior podemos observar que el nodo “lanzando_vel” está publicando en los topics de velocidad de cada uno de los robots. A su vez, el nodo gazebo, o lo que es lo mismo, el modelo virtual del robot P3DX está publicando en el nodo Odom de cada uno de los topics. El nodo “lanzando_vel” está subscrito a estos topics Odom, los cuales tiene un campo “twist.twist” donde podemos conseguir las velocidades leídas de la odometría como explicamos en la pág 76 y también la odometría del robot si se quisiese.

En la Ilustración 4.6 el nodo “lanzando_vel” solo está subscrito al topic Odom del primer robot, esto es así solo en la ilustración, ya que se ha decidido ilustrarlo así para un mejor entendimiento y que la figura no sea engorrosa, pero en realidad está subscrito a los topic Odom de cada robot.

Así es como publicamos y subscribimos velocidades en un convoy virtualizado en ROS/Gazebo. Se puede ver la comunicación completa entre el centro remoto y el convoy virtual en la Ilustración 4.9, donde los datos (velocidades y odometria) publicados y subscritos viajan al centro remoto a través del socket en las estructuras datos_ros y lectura_ros.

Esto es todo lo que se debe modificar en las herramientas ROS/Gazebo para la realización de un convoy, así como enviar y recibir datos del mismo.

4.2. Descripción del algoritmo de control para el guiado de robots P3DX en convoy

El objetivo que se desea testear como algoritmo de control es que un conjunto de robots (convoy) realicen la misma trayectoria uno detrás de otro.

Para ello, la idea básica que se seguirá será, en una primera aproximación, la aplicación de un servosistema para cada robot:

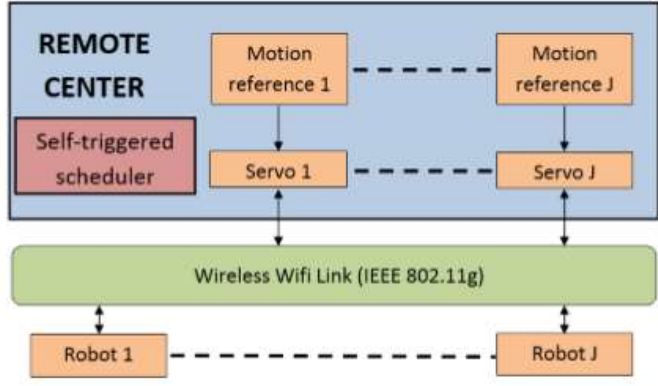


Ilustración 4.7: Propuesta de control a un conjunto de robots.

En la anterior figura podemos ver la idea de este control.

Existen numerosas soluciones para el control de unidades en convoy, por ejemplo, las descritas en [10] [22].

4.3. Comunicación entre PC Centro Remoto - Convoy Virtual/Real

La comunicación entre los PC Centro Remoto – Convoy Virtual/Real se realiza a través de aplicaciones cliente-servidor donde el descriptor es el mismo socket explicado en el apartado 3.5. Y para conseguir esta comunicación desde Matlab/Simulink se utilizan los bloques S-Function al igual que para un único robot, por lo que la idea general de comunicación es la misma descrita a lo largo del apartado 3.5, pero realizando algunas modificaciones en el socket del centro remoto dependiendo de si se comunica con el convoy virtual y real, y en el nodo de comunicación del convoy virtual.

El nodo de comunicación del convoy real es idéntico al explicado en el apartado 3.5.3.2 para cada robot que lo conforma. Esto es así dado que cada robot tendrá un canal de comunicación independiente (IP diferente) con el centro remoto, enviando y recibiendo velocidades sin interferir en la comunicación de los demás robots que conforman el convoy con el PC centro remoto. Por tanto se utilizará la misma S-Function que se utilizó en el apartado antes mencionado para cada robot del convoy.

4.3.1. Modificación en el socket del centro remoto

En cuanto al convoy virtual, este es visto por el socket como un solo cliente-servidor ya que solo existe un canal de comunicación entre el centro remoto y el PC que virtualiza el convoy. El nodo de comunicación de ROS es el que se encarga de procesar la información que llega del centro remoto y aplicarla a los robots del convoy por separado.

Por tanto, esta vez los datos que viajan desde el PC centro remoto al convoy virtual es una estructura donde están, para el caso de un convoy con 4 robots, 4 velocidades lineales y 4 angulares, una para cada robot, que comparándolo con el caso de un solo robot aquí teníamos únicamente una velocidad lineal y otra angular.

Esta estructura la hemos denominado datos_ros y es la siguiente:

```
typedef struct
{
unsigned long data_number=0;
double linear[4];
double angular[4];
}dato_ros;
```

Por otro lado, los datos de la velocidad leída por la odometría del robot viajan a su vez en otra estructura del mismo tipo que datos_ros, denominada lectura_ros.

Estos datos se transfieren a una única dirección IP, la cual es la dirección IP del PC que virtualiza el convoy.

Por el contrario, para la comunicación con el convoy real, tenemos tantos canales de comunicación (sockets) como robots tengamos ya que el centro remoto tiene que enviar datos a tantas direcciones IP como robots haya.

Por lo tanto no necesitamos una estructura, si no identificar bien las velocidades lineales y angulares de cada robot y enviarlas a la dirección IP correspondiente del robot, la cual es estática. Por lo que el nodo de comunicación de cada robot real que conforma el convoy no se modifica con respecto al explicado en el apartado 3.5.3.2 dado que cada robot es tratado de forma individual por el control.

Esta diferencia en cuanto a sockets dependiendo si se está comunicando con el convoy virtual o real queda reflejada en la siguiente figura:

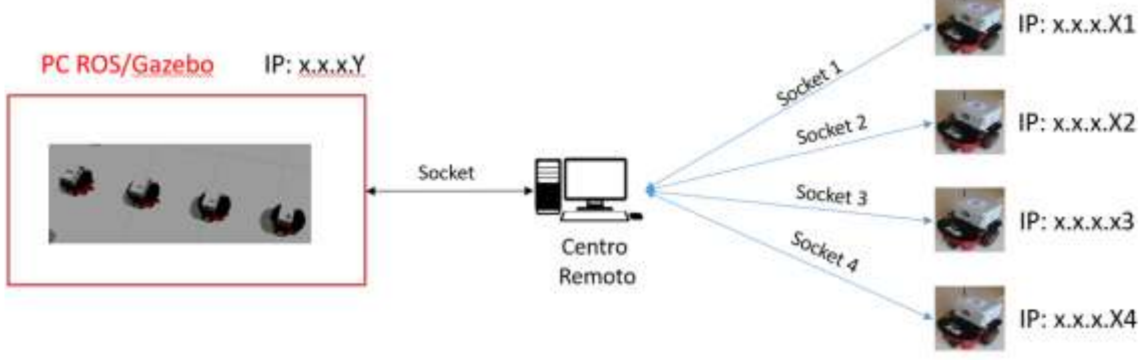


Ilustración 4.8: Sockets dependiendo de si se comunica con convoy virtual o real.

Donde se puede observar que el centro remoto se comunica con una dirección IP a través de un único socket (implementado en el nodo de comunicación de ROS) con el convoy virtual, y con 4 direcciones IPs a través de 4 sockets (implementados cada uno en la S-Function de cada VIA-EPIA y en el centro remoto) con el convoy real (suponiendo que este esté compuesto por 4 robots).

4.3.2. Modificación en nodo de comunicación del Convoy Virtual

La única modificación que hay que implementar en el nodo de comunicación de ROS es que ahora, como dijimos en el apartado anterior, los datos llegan desde el centro remoto en una estructura, por lo que hay que tratarlos como tal y enviar a cada robot el elemento de la estructura correspondiente. Por ejemplo para el caso de un convoy con 2 robots, tenemos que el elemento linear[0] de la estructura "datos_ros" que viene del socket es la consigna de velocidad lineal correspondiente al robot 0 del convoy, y el elemento lineal[1] la correspondiente al robot 1.

Esta comunicación con el convoy virtual se puede ver en la siguiente imagen:

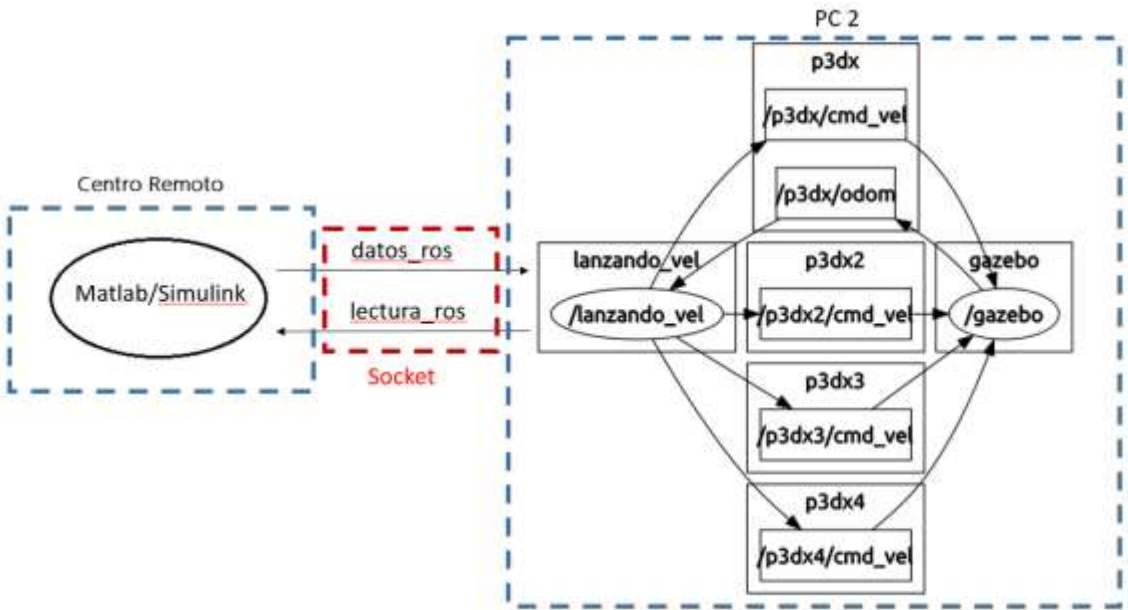


Ilustración 4.9: Comunicación entre Centro Remoto y convoy de 4 robots ROS/Gazebo.

4.4. Procedimiento a seguir para ejecutar la solución de control del Centro Remoto en Convoy Virtual y Convoy Real

4.4.1. En Convoy Virtual

El procedimiento a seguir para implementar el algoritmo de control al convoy es idéntico que el del apartado 3.6.1, ya que el convoy es visto como un único nodo, el nodo de ROS, el cual es este el que gestiona el convoy virtual.

4.4.2. En Convoy Real

El procedimiento a seguir para la implementación del algoritmo de control en el convoy real es el siguiente:

1. Se crean los ejecutables a través de la herramienta RTW de cada unidad robótica.

2. Se crea el ejecutable del algoritmo de control del centro remoto.

3. Se corren los ejecutables de cada robot. Ahora, estos estarán esperando a que el centro remoto les envíe datos al ser el canal de comunicación un socket cliente-servidor.

4. Se corre el ejecutable del centro remoto.

5. Una vez terminada la prueba, se pueden recoger los resultados en Matlab y plotearlos.

Capítulo 5: Propuesta de evitación de obstáculos

A lo largo de los capítulos 3 y 4 se ha explicado la aplicación de un control de seguimiento de velocidades tanto a un solo robot real o virtual como a un convoy formado por robots reales o virtuales.

En este capítulo 5 se propone un algoritmo de control borroso tipo Mamdani de evitación de obstáculos para implementarlo a un único robot P3DX virtual o real.

Una de las razones de proponer este control es porque la herramienta Gazebo permite añadir y quitar cualquier tipo de obstáculos (forma, tamaño…) en el entorno. Esto es muy útil debido a que nos permite probar este tipo de algoritmo de una manera sencilla y con garantías.

En este capítulo se detalla lo siguiente:

- Descripción del algoritmo de control para evitación de obstáculos.
- Implementación del algoritmo en robot virtual y en robot real.

5.1. Descripción del algoritmo de control para evitación de obstáculos

Existen numerosas técnicas con las que abordar un control de evitación de obstáculos como pueden ser: método del campo de potencial [23], histograma de vector de campo [24], algoritmo inteligente de error [25], control borroso [26], entre otros.

Para un mejor conocimiento en el estudio de la comparación entre métodos de control para la evitación de obstáculos se puede analizar la referencia [27].

En este trabajo se ha optado por técnicas borrosas para el diseño del control, más concretamente, tipo Mamdani.

La lógica borrosa proporciona una metodología formal para aplicar el conocimiento heurístico humano al control de procesos que se quiere implementar. Algunas razones que justifican el uso de lógica borrosa en el control son: conceptualmente fácil de entender e implementar, es flexible y tolerante a la imprecisión de datos, permite modelar funciones no lineales…

Los componentes de un control borroso son "conjuntos borrosos" y un conjunto de reglas del tipo If-Then que codifican el comportamiento del robot móvil. La principal dificultad en el diseño de este tipo de control es la formulación eficaz de estas reglas If-Then.

El siguiente diagrama de bloques de Simulink representa el control borroso de evitación de obstáculos diseñado para una posición del obstáculo:

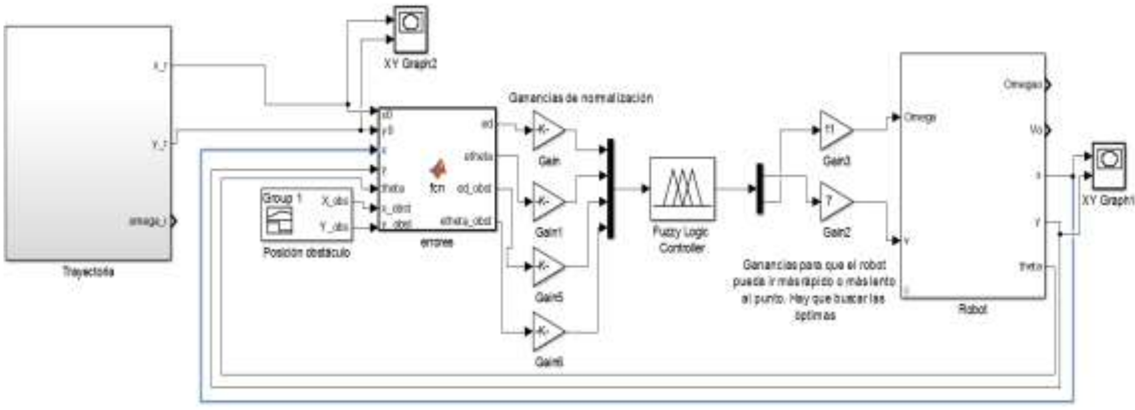


Ilustración 5.1: Diagrama de bloques FLC con evitación de obstáculos y modelo cinemático del robot.

Como vemos tenemos un bloque "trayectoria" en el cual a partir de unas velocidades lineales y angulares dadas se calcula mediante el modelo cinemático la posición (x,y) y la orientación objetivo que el robot debe de seguir.

El bloque final "Robot" son las ecuaciones del modelo cinemático del robot P3DX.

El bloque de "posición obstáculo" representa la posición (x,y) donde se encuentra el próximo obstáculo a esquivar por el robot.

Los demás bloques existentes en el diagrama de bloques corresponden al control borroso, donde podemos identificar las siguientes partes:

- Ganancias de normalización: Son las ganancias a la entrada/salida del controlador borroso. Adaptan el rango de variación de las entradas o salidas al rango de variación de las variables lingüísticas del FLC (Fuzzy Logic Control). Generalmente los sistemas borrosos trabajan con variables normalizadas por lo que se precisa de estas ganancias.

- Fuzzy Logic Controller: Es el bloque de Simulink que se utiliza para un control borroso. En él se pueden diferenciar las siguientes partes:

Ilustración 5.2: Partes del Fuzzy Logic Controller.

  - Borrosificación: En esta parte se convierten los valores numéricos en borrosos.
  - Base de conocimiento: Aquí se encuentran todos los algoritmos de control, conjuntos, reglas… del control borroso. Éstas debemos de programarlas a través de la toolbox "fuzzy".
  - Desborrosificación: Convierte los valores borrosos a numéricos.

A la entrada de la lógica borrosa tenemos un bloque de errores en el que tenemos el cálculo de:

- $e_d = \sqrt{(x - x_o)^2 + (y - y_o)^2}$, error de posición respecto al punto de destino.
- $e_\vartheta = \arctan\left(\frac{y_o - y}{x_o - x}\right) - \vartheta$, error de orientación respecto al punto de destino.
- $e_{dobst} = \sqrt{(x - x_{obst})^2 + (y - y_{obst})^2}$, error de posición respecto al obstáculo.
- $e_{\vartheta obst} = \arctan\left(\frac{y_{obst} - y}{x_{obst} - x}\right) - \vartheta$, error de orientación respecto al obstáculo.

donde (x,y) es la posición actual del robot, (x_o, y_o) es la posición del punto destino y $x_{obst}, y_{obst})$ es la posición del obstáculo.

A la salida de la lógica borrosa tendremos la velocidad lineal y la velocidad angular que se aplica al robot una vez pasa por el control borroso.

- **Reglas utilizadas**

Las variables de las reglas que se utilizan en una lógica borrosa son las entradas y salidas de la misma, que en nuestro caso son los errores (entradas) y las velocidades lineal y angular (salidas). Estas reglas representan el comportamiento del robot.

- En cuanto al seguimiento de trayectorias:

A la hora de realizar un seguimiento de trayectorias por control borroso, la estrategia que se plantea es la siguiente:

· Si el robot está cerca del objetivo y orientado a él, entonces la velocidad lineal debe de ser baja (robot lento) y la velocidad angular 0 (sin orientación). Si está cerca y por el contrario no está orientado hacia el punto objetivo, entonces ahora la velocidad angular del robot tendrá el signo correspondiente a compensar esa orientación.
· Si el robot está lejos del objetivo, entonces la velocidad lineal deberá de ser elevada para que el robot vaya rápido. En este caso la velocidad angular tiene menos importancia aunque es conveniente que si el robot no está orientado al punto, compensar un poco la orientación (no tiene tanta importancia como la puede tener si estamos cerca del objetivo ya que en este caso se debe corregir en un menor tiempo esta orientación).

Siguiendo estas nociones, las reglas utilizadas han sido las siguientes:

· Si e_d es cerca y e_ϑ negativo entonces V lento y W muy negativo
· Si e_d es cerca y e_ϑ cero entonces V lento y W cero
· Si e_d es cerca y e_ϑ positivo entonces V lento y W muy positivo

Como vemos si estamos cerca y desorientados al punto objetivo, la velocidad lineal es baja y la angular es alta en el sentido para compensar el error de orientación y hacer que el robot se oriente cuanto antes al punto (ya que estamos cerca). Si el robot está orientado al punto ($e_\vartheta = 0$) entonces W debe de ser cero para que el robot viaje en línea recta.

· Si e_d es lejos y etheta negativo entonces V rápido y W negativo
· Si e_d es lejos y etheta cero entonces V rápido y W cero
· Si e_d es lejos y etheta positivo entonces V rápido y W positivo

Al contrario si estamos lejos de nuestro objetivo conviene que la velocidad lineal sea elevada para llegar antes al mismo. La velocidad angular también debe de compensar el error de orientación, pero en este casi no es tan importante dado que estamos lejos de nuestro objetivo, por lo que ahora no tenemos “muy…” que significa velocidad angular elevada para que el robot gire más y se compense antes el error.

- <u>En cuanto a la evitación de obstáculos:</u>

La estrategia que se va a seguir es tener en cuenta el obstáculo solo cuando está cerca y actuar sobre las velocidades, si el obstáculo está lejos entonces no hacemos nada sobre el control del robot. Sabiendo esto, las reglas que se añaden a las del apartado anterior son las siguientes:

- Si e_{dobst} es cerca y $e_{\vartheta obst}$ es negativo entonces V lento y W muy positivo.
- Si e_{dobst} es cerca y $e_{\vartheta obst}$ es cero entonces V lento y W negativo.
- Si e_{dobst} es cerca y $e_{\vartheta obst}$ es positivo entonces V lento y W muy negativo.

Como vemos, si el obstáculo está cerca y tenemos una orientación sobre él, giramos hacia la misma dirección para seguir esquivándolo. Si es cero basta con girar en una orientación da igual cual, pero hay que girar para sortear el obstáculo. Y como podemos observar en las reglas, en ninguna de ellas aparece si e_{dobst} está lejos por lo que el control no hará modificaciones en las velocidades del robot si el obstáculo está lejos.

- **<u>Conjuntos borrosos utilizados</u>**

Un conjunto borroso puede contener elementos de forma parcial, es decir, que la propiedad de que un elemento X pertenezca al conjunto $A(X \in A)$ puede ser cierta con un determinado grado de pertenencia.

Se debe de crear un conjunto borroso para cada entrada y salida del FLC.

En cuanto a las entradas tenemos los siguientes conjuntos borrosos:

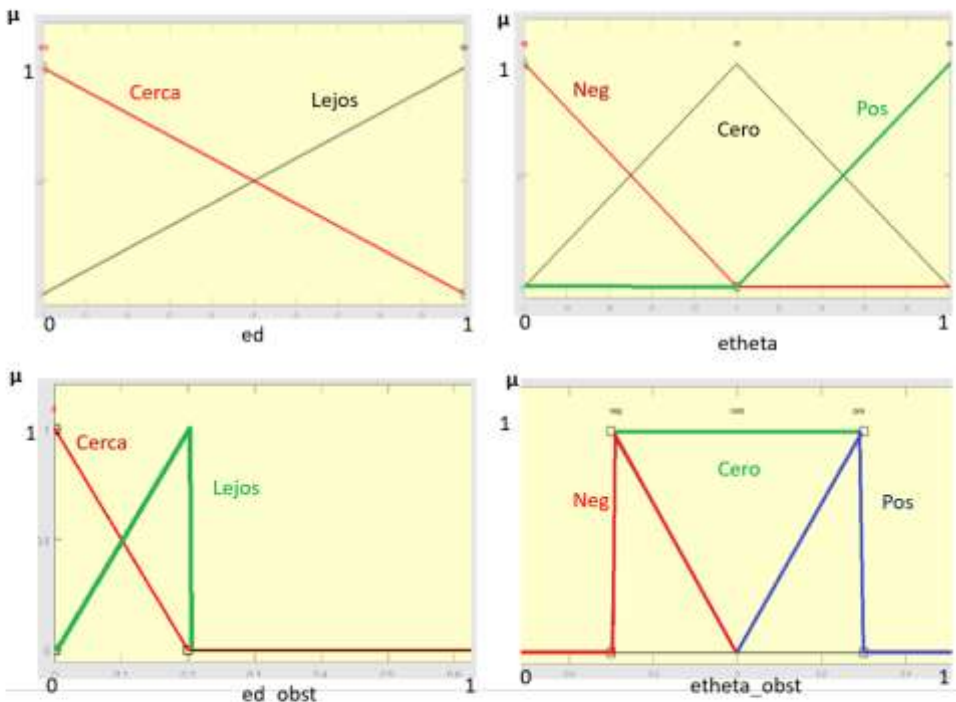


Ilustración 5.3: Conjuntos borrosos de las entradas.

En cuanto a los errores de posición, para "cerca" vemos que un grado de pertenencia de 1 es cuando el error de posición es nulo, lo que quiere decir que o el punto destino o el obstáculo están cerca. Al contrario con "lejos".

Para los errores de orientación vemos que tenemos 3 casos, si el error es positivo (punto objetivo u obstáculo están con un ángulo positivo respecto al obstáculo, negativo (lo contrario que positivo) y cero (cuando el punto objetivo u obstáculo está en línea recta con respecto al robot).

En cuanto a las salidas tenemos la velocidad lineal y angular, las cuales representan lento y rápido, y el giro del robot (muy negativo, negativo, cero, positivo, muy positivo).

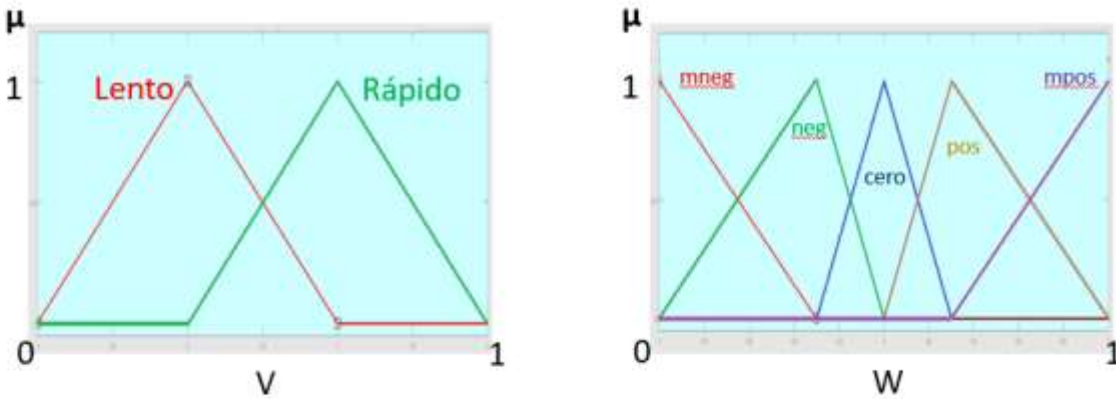


Ilustración 5.4: Conjuntos borrosos de las salidas.

Una vez se explican las reglas, los conjuntos borrosos y el diagrama de bloques del control, vamos a ver su comportamiento.

Para ello, mediante el bloque "trayectoria" se va a generar una trayectoria formando un 8 con radio 1 metro, con velocidades de 0.25 m/s y 0.25 rad/s. El obstáculo que deberá esquivar el robot se sabrá a priori, siendo su posición: (1,0).

Simulando el diagrama de bloques de la Ilustración 5.1 con estas premisas tenemos la siguiente trayectoria seguida por el modelo cinemático del robot:

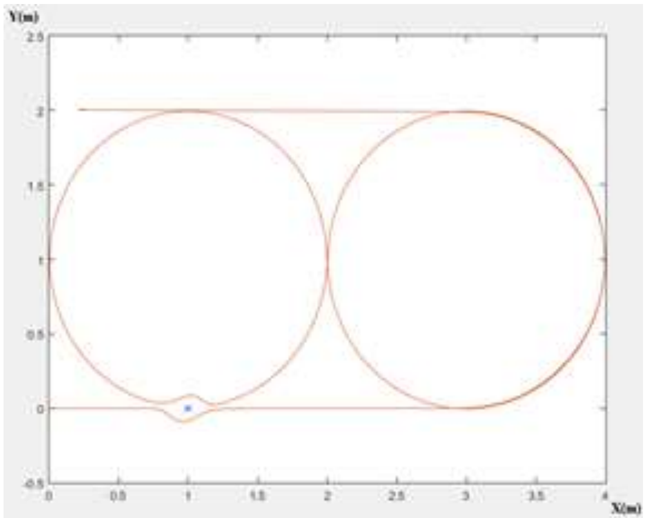


Ilustración 5.5: Trayectoria seguida por el modelo cinemático del robot con el control FLC de evitación de obstáculos.

Donde podemos ver que el control esquiva perfectamente el obstáculo situado en (1,0).

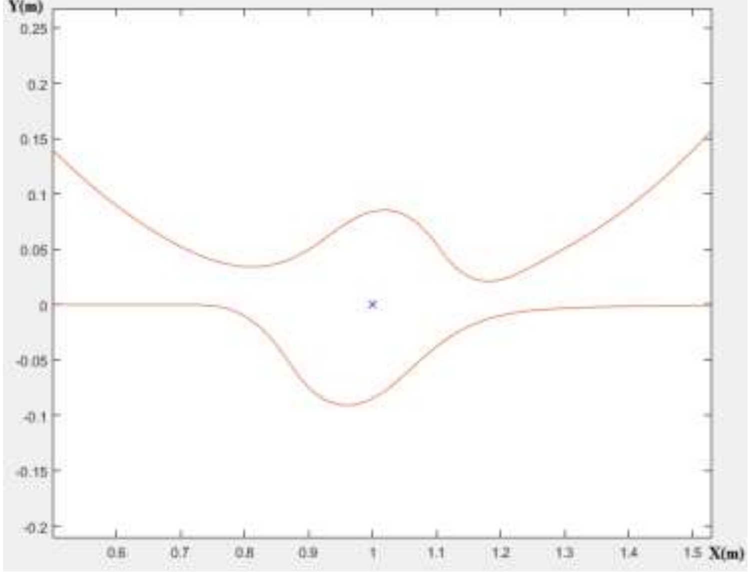


Ilustración 5.6: Evitación del obstáculo situado en (1,0).

Los parámetros que influyen en la distancia de seguridad que deja el robot con respecto al obstáculo son las ganancias que entran al controlador borroso multiplicando a los errores en posición y orientación respecto del obstáculo:

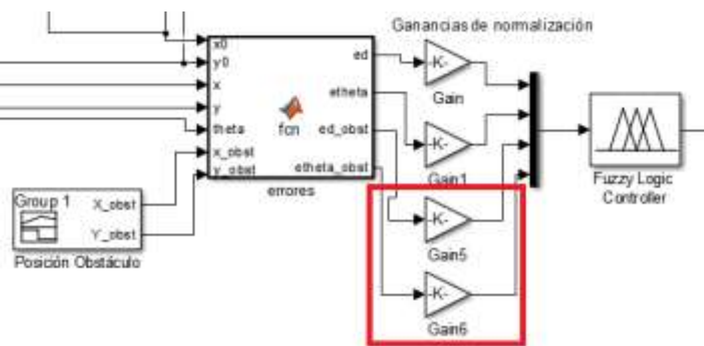


Ilustración 5.7: Ganancias para controlar la distancia de seguridad con respecto al obstáculo.

Cuanto más pequeñas sean estas ganancias mayor es la distancia de seguridad, pues al ser más lenta la respuesta del robot, este ha de iniciar la evitación de obstáculos desde puntos más alejados a este.

Veamos en el siguiente apartado como se puede implementar para un robot virtual y real.

5.2. Implementación en robot virtual/real

En el control anterior, el obstáculo era representado mediante un punto dentro de la trayectoria. El problema que se presenta en la vida real es el tamaño de los obstáculos, ya que estos no los podemos representar como un punto. La solución que se plantea es representar los obstáculos con numerosos puntos delimitando el objeto (periferia) a esquivar, tal que por ejemplo si se quiere esquivar una caja:

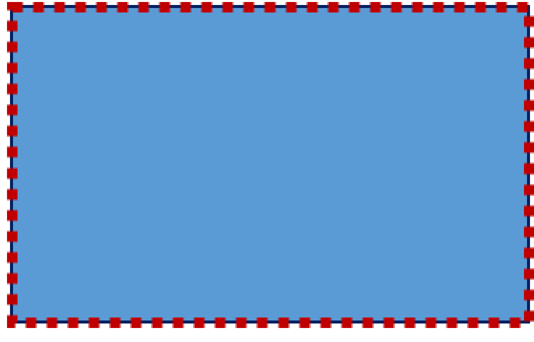


Ilustración 5.8: Obstáculo: Azul representa la caja a esquivar. Los puntos rojos representan los obstáculos a esquivar por parte del control.

donde para el controlador, los puntos rojos serán todos los obstáculos que se deberán esquivar. Así solucionamos el problema del tamaño que tenga la caja, aunque este, debe de ser sabido a priori, así como su posición, para poder introducir en el control todos los puntos a esquivar.

Con esta idea, vamos a ver cómo se comporta el control con un ejemplo. Con la misma trayectoria que el apartado anterior, el robot debe esquivar una caja de 15 centímetros de longitud cada lado situada en (1.8, 0). Para el control, los puntos a esquivar por el mismo serán los indicados en la Ilustración 5.8 en color rojo, separados 1 centímetro.

Podemos observar la trayectoria seguida por el robot en la siguiente figura:

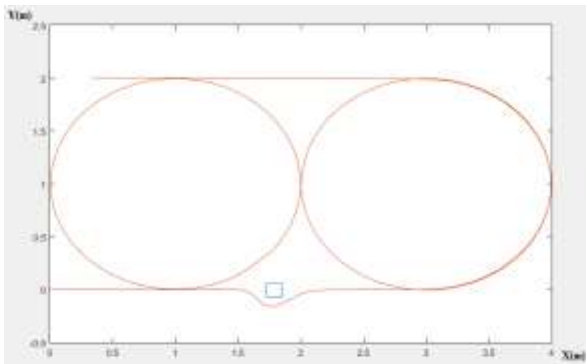


Ilustración 5.9: Trayectoria seguida por el robot esquivando una caja como obstáculo.

Donde como vemos, el robot esquiva la caja representada en azul con una distancia de seguridad, la cual se puede modificar con las ganancias de la Ilustración 5.7.

5.2.1. En Robot Virtual

Como vemos en la Ilustración 5.10, los datos que necesitamos obtener son la posición x,y del robot y la orientación theta. Para ello, basta con enviar al centro remoto a través del socket las velocidades lineal y angular subscribiéndose al topic "odom" tal y como se explicó en el apartado 3.3.1.3, y calcular en el centro remoto la posición y orientación del robot a través de las ecuaciones del modelo cinemático en tiempo discreto. Quedando por tanto, el subsistema "robot" de la Ilustración 5.1 como sigue:

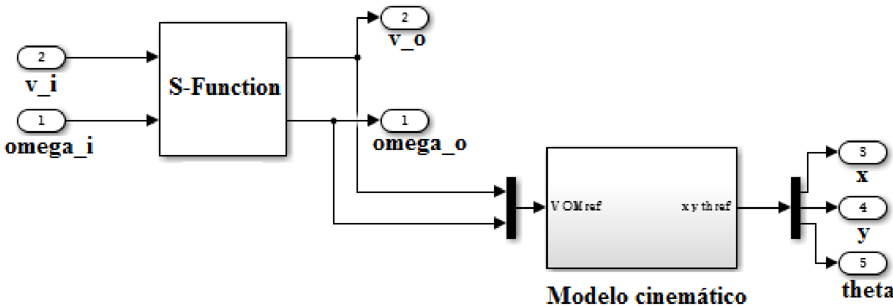


Ilustración 5.10: Subsistema "robot", con S-Function y modelo cinemático.

El bloque S-Function permitirá la comunicación mediante sockets entre el centro remoto y el PC virtualizando el robot, y el modelo cinemático permitirá obtener la posw (x, y, th) a partir de las velocidades de salida que proporciona el topic "odom" del robot virtualizado. Quedando el esquema de comunicación como sigue:

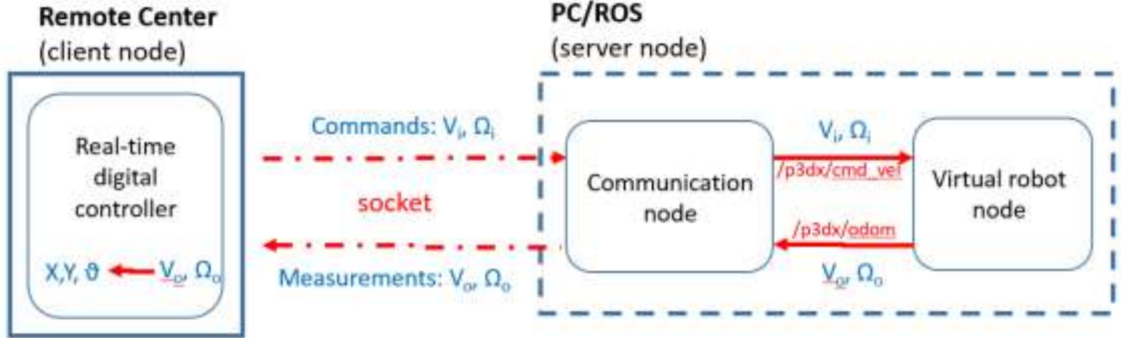


Ilustración 5.11: Comunicación entre centro remoto y robot virtual calculando la pose en el centro remoto.

Otra alternativa sería obtener directamente del topic "odom" la posición y orientación del robot y prescindir del modelo cinemático en el centro remoto. Para ello, la posición se puede sacar como:

```
msgs::Odometry& msg
x = msgs.pose.pose.position.x
y = msgs.pose.pose.position.y
```

donde x,y es la posición del robot.

La orientación del robot viene dada por el topic en quaternios. Por tanto basta con transformarlos en ángulos de Euler como sigue:

```
msgs::Odometry& msg

quaternion = ( msg.pose.pose.orientation.x,
               msg.pose.pose.orientation.y,
               msg.pose.pose.orientation.z,
               msg.pose.pose.orientation.w );

euler = tf.transformation.euler_from_quaterion(quaternion);

roll = euler[0];
pitch = euler[1];
yaw = euler[2];
```

Siendo yaw la orientación respecto al eje Z, o lo que es lo mismo, theta, la cual es la que necesitamos.
Entonces con esta alternativa el esquema de comunicación quedaría como:

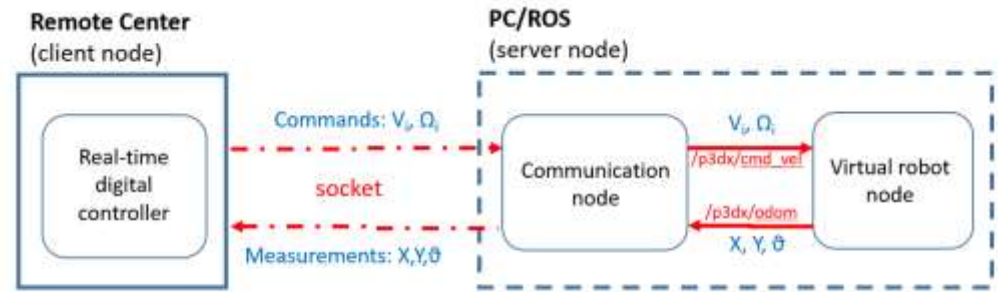


Ilustración 5.12: Comunicación entre centro remoto y robot virtual obteniendo la pose del topic odom..

Donde se prescindiría del modelo cinemático y por tanto, la entrada a la S-function serían las velocidades de entrada al robot virtual y la salida la posición y orientación del mismo.

- **Introducir un obstáculo en Gazebo**

Se pueden introducir objetos (obstáculos) de dos formas a través de Gazebo.

Una forma es introduciendo los objetos ya creados por la comunidad, los cuales están en el repositorio de Gazebo. Para ello basta con irse a la pestaña Insert de Gazebo-gazebosim.org/models y seleccionar el objeto que se desea introducir:

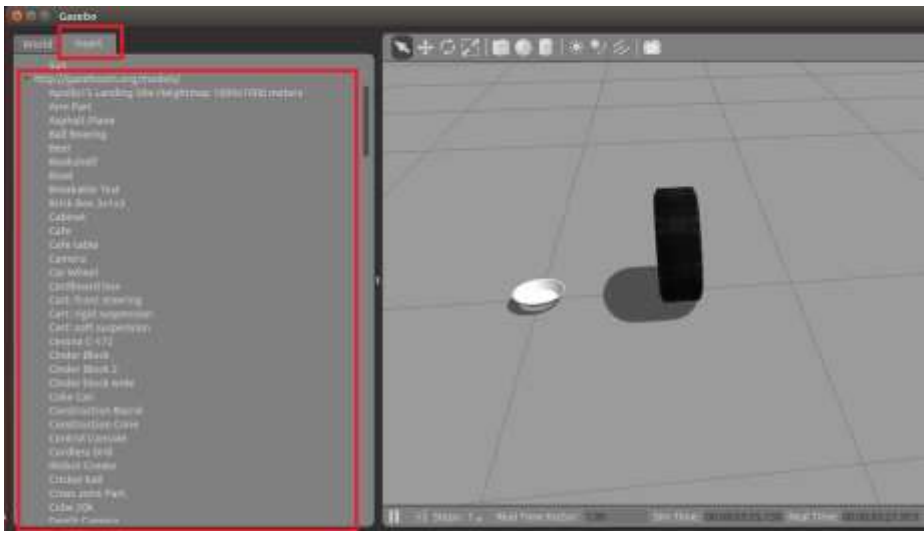


Ilustración 5.13: Insertar obstáculos del repositorio en Gazebo.

En la ilustración anterior se han insertado un bowl y una rueda de coche como obstáculos. El tamaño y la pose de estos objetos se puede visualizar y ser modificado en la pestaña World de gazebo y en su correspondiente campo del objeto. Por ejemplo para la rueda del coche:

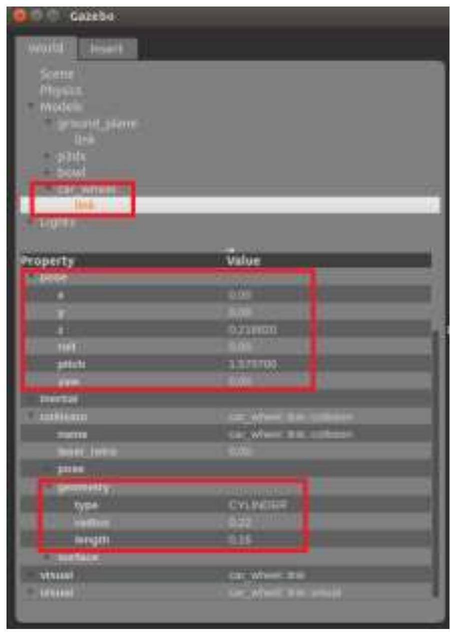


Ilustración 5.14: Modificación de pose y tamaño de los objetos introducidos en Gazebo.

La pose de todos los objetos sean del tipo que sean se refiere a la posición del centro geométrico del mismo.

El otro método es el que se utiliza en este TFM. Este método consiste en crear un archivo .sdf indicando la pose y el tamaño del objeto que se desea introducir, tal que:

```
<?xml version='1.0'?>
<sdf version='1.4'>

 <model name='Obstaculos'>
  <link name='obstaculo1'>
   <pose>0 0 0 0 0 0</pose>
   <visual name='obstaculo1_Visual'>
    <pose>1.8 0 0.075 0 0 0</pose>
    <geometry>
     <box>
      <size>0.15 0.15 0.15</size>
     </box>
    </geometry>
    <material>
     <script>
      <uri>file://media/materials/scripts/gazebo.material</uri>
      <name>Gazebo/Grey</name>
     </script>
    </material>
   </visual>
   <collision name='Wall_0_Collision_0'>
    <geometry>
     <box>
      <size>0.15 0.15 0.15</size>
     </box>
    </geometry>
    <pose>1.8 0 0.75 0 0 0</pose>
   </collision>
  </link>
  <static>1</static>
 </model>
</sdf>
```

En este .sdf se ha introducido una box posicionada en x=1.8 y=0 z=0.075 metros, con un tamaño de 0.15 metros cada lado. Se puede ver que existe una pose y size tanto para <visual> como para <collision>. <visual> es lo que se visualizará en Gazebo, y <collision> es lo que tendrá colisiones. En nuestro caso, mostraremos una caja la cual tendrá toda ella colisiones, por tanto ambos campos tendrán las mismas "poses" y "sizes". Esto es útil dado que por ejemplo para los cristales, se quiere visualizar en gazebo, pero se puede no querer que tengan colisiones debido a que los láseres no colisionan en ellos.

Una vez se crea el fichero .sdf con los obstáculos, este se lanza desde un archivo .launch al igual que se explicó en la pág 69. Por ejemplo si el fichero se llama “obstaculoWorld.sfd”:

```
<!—A continuación cargamos el mundo de Gazebo -->
<include file="$(find gazebo_ros)/launch/empty_world.launch">
        <arg name="world_name" value="$(find p3dx_gazebo)/ol3_floor/obstaculoWorld.sdf" />
        <arg name="debug" value="$(arg debug)" />
        <arg name="gui" value="$(arg gui)" />
        <arg name="paused" value="$(arg paused)" />
        <arg name="use_sim_time" value="$(arg use_sim_time)" />
        <arg name="headless" value="$(arg headless)" />
</include>
```

Lanzando el fichero .launch donde también se agrega un robot móvil P3DX se puede visualizar el obstáculo con la unidad móvil.

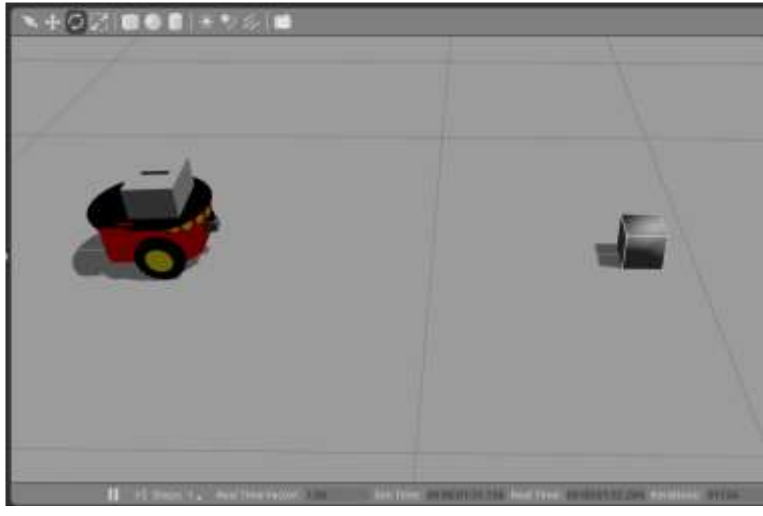


Ilustración 5.15: Caja obstáculo junto con unidad móvil P3DX en Gazebo.

5.2.2. *En Robot Real*

Para la comunicación del PC centro remoto con el robot real se utiliza la S-Function descrita en 3.5.3.2 con la interfaz ARCOS.

Es la misma idea que para el robot virtual, donde esta vez se obtienen las velocidades de las ruedas (wr,wl) mediante el encóder, calculando las velocidades de salida del robot (vo,wo) en la tarjeta Via Epia y calculando, a partir de estas, la posición y orientación del robot en el centro remoto a través del modelo cinemático.

Quedando la comunicación entre PC centro remoto y robot real de la siguiente manera:

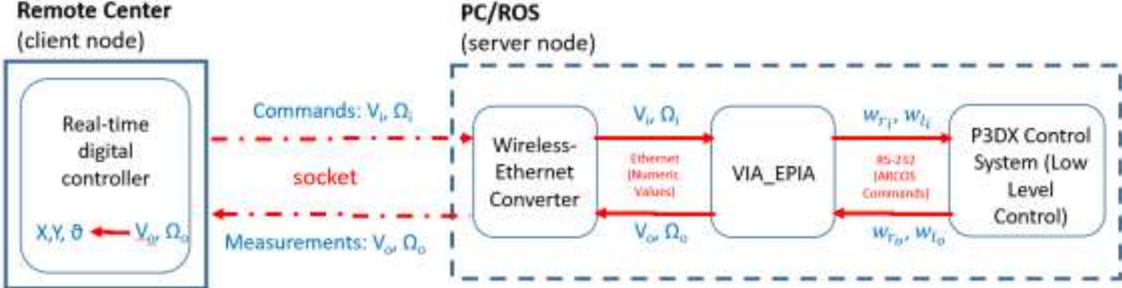


Ilustración 5.16: Comunicación entre centro remoto y robot real obteniendo la pose y orientación del mismo.

Capítulo 6: Validación de resultados emulados con robot virtual e implementados en robot real

Una vez se ha explicado el procedimiento del trabajo se va a validar el mismo ilustrando un ejemplo tanto para el servosistema en un único robot virtual/real como para el algoritmo de control para un conjunto de robots virtual/real.

6.1. Servosistema en robot virtual y robot real

El sistema en cuestión es un servosistema de velocidad el cual es ejecutado por el centro remoto en Matlab/Simulink, que primero controlará un robot virtual P3DX en un PC distinto al centro remoto, donde estarán corriendo las herramientas ROS/Gazebo, y finalmente en una unidad robótica real del P3DX. Los tres nodos Wifi comparten el mismo centro de comunicación, un router inalámbrico WHR-HP-54.

Lo que se pretende es que el robot siga una trayectoria lineal partiendo del laboratorio OL3 de la Escuela Politécnica Superior de Alcalá hasta el laboratorio OL5 mandando consignas de velocidad lineal y angular.

En la siguiente imagen se puede observar el robot virtual y el entorno generado con ROS/Gazebo en el lado izquierdo, mientras que en el lado derecho se tiene el robot real y el entorno real. Esta imagen es captada en el punto de partida OL3.

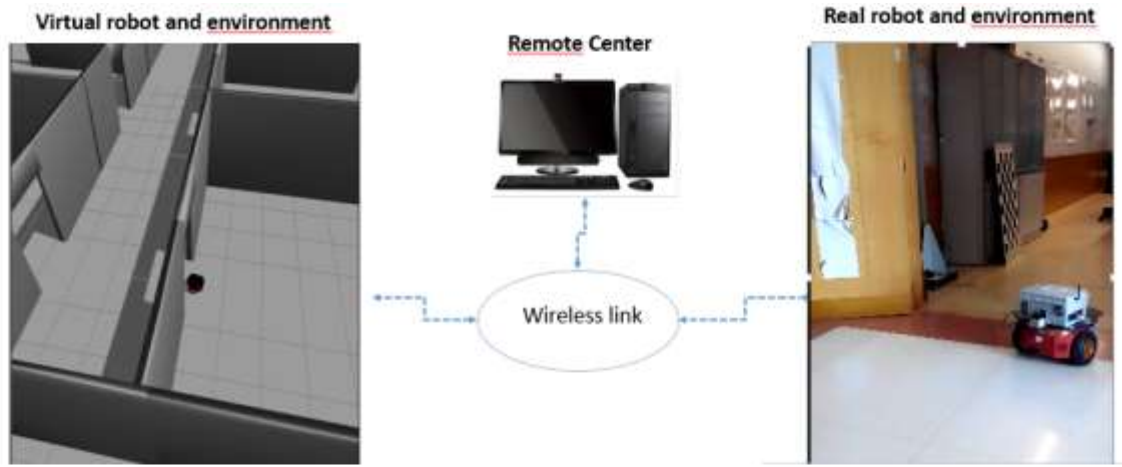


Ilustración 6.1: Captura del movimiento del robot virtual (izquierda) y real (derecha) a lo largo del pasillo Oeste en el punto de partida OL3.

La siguiente imagen muestra el movimiento algunos segundos después del punto de partida OL3:

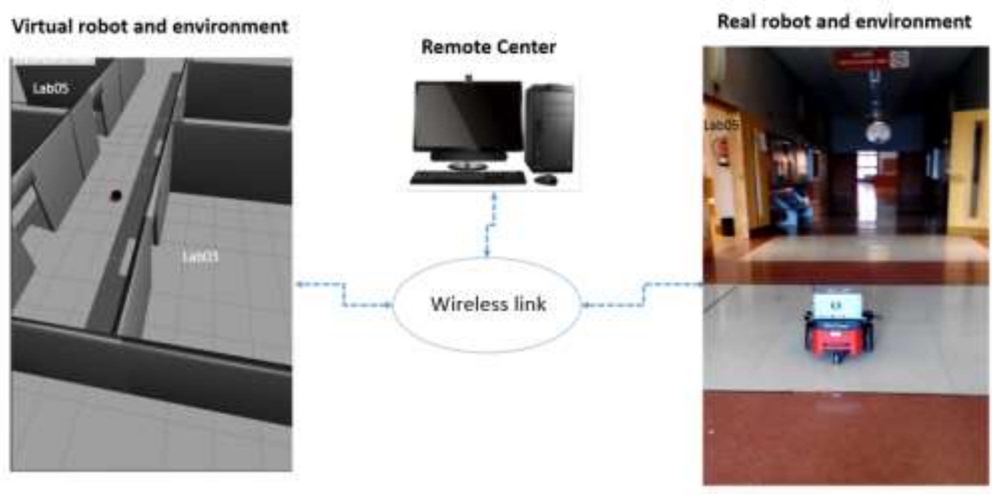


Ilustración 6.2: Captura del movimiento del robot virtual (izquierda) y real (derecha) a lo largo del pasillo Oeste algunos segundos despues de partir del OL3.

Mientras que en la última ilustración podemos ver otra captura pero esta vez cerca del punto de destino, el OL5:

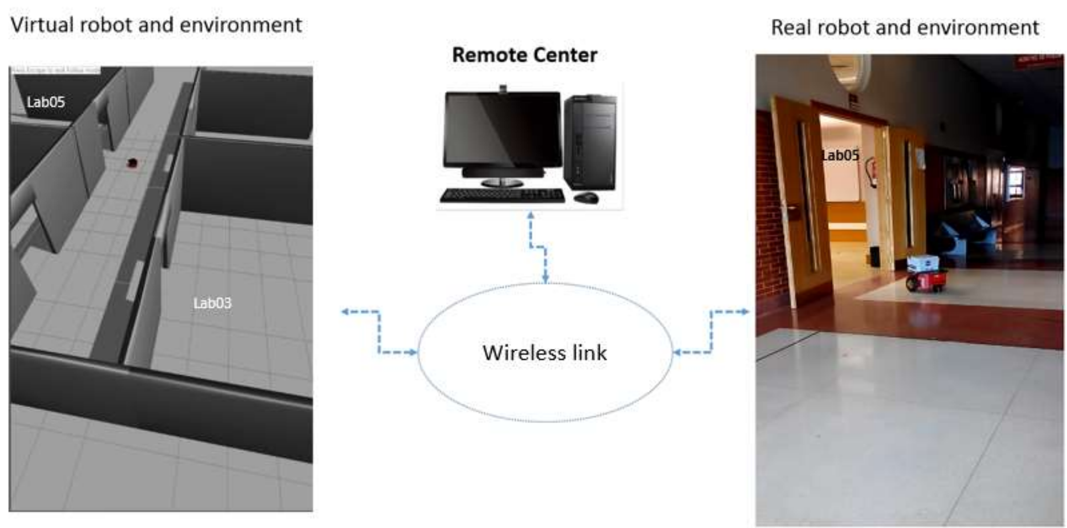


Ilustración 6.3: Captura del movimiento del robot virtual (izquierda) y real (derecha) a lo largo del pasillo Oeste cerca del punto objetivo OL5.

Durante la aplicación, se ha podido observar que tanto el robot virtual y el real siguen el mismo camino partiendo del mismo punto y llegando al mismo punto objetivo.

Una vez se realiza el ejemplo tanto en el robot virtual como en el real, vamos a ver si el modelo realizado en Gazebo como robot virtual se comporta igual que el modelo real, o lo que es lo mismo, vamos a ver si conseguimos nuestro objetivo.

Para ello, vamos a recoger los datos de las velocidades lineal y angular leídas por la odometría tanto del robot virtual de ROS/Gazebo como del robot real al aplicarles la consigna que consigue que la plataforma vaya del OL3 (punto de partida) al OL5 (punto de destino). Estos datos son recogidos por el Centro Remoto donde podemos analizarlos utilizando Matlab, ya que son datos recogidos en un .mat. Para ello basta con cargar el archivo .mat (load 'archivo.mat') y plotear los resultados:

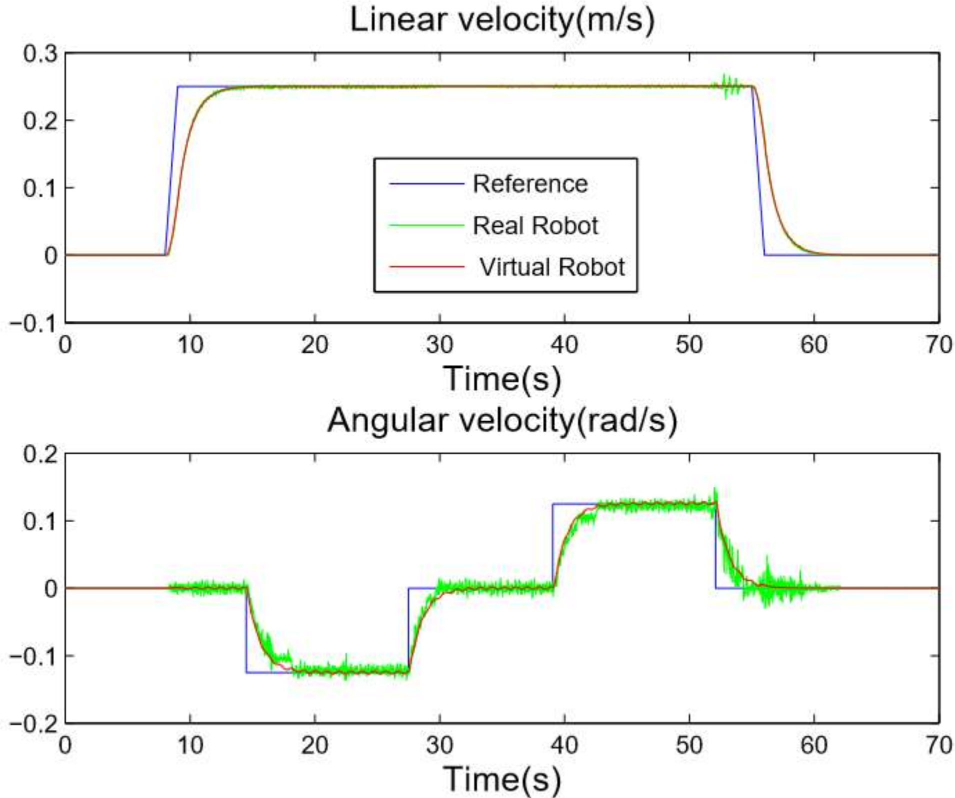


Ilustración 6.4: Velocidad lineal (arriba) y velocidad angular (abajo) reportadas tanto por el robot virtual(rojo) y real(verde) respecto a la consigna(azul).

La línea azul tanto para velocidad lineal como angular es la referencia que se le aplican a los robots real y virtual. Estas velocidades permiten conducir la unidad robótica partiendo del OL3 para llegar al OL5, con una longitud aproximada de camino de 10 metros, incluyendo giros y caminos rectos como se puede observar en Ilustración 6.1, Ilustración 6.2 e Ilustración 6.3. El tiempo de muestreo de control es de 10ms.

Podemos observar en la Ilustración 6.4 que el servocontrol aplicado al robot virtual (línea roja) ha entregado velocidades leídas por la odometría aceptables, ya que si las

comparamos con el robot real (línea verde) podemos observar que ambas siguen la misma trayectoria e incluso tienen el mismo tiempo de respuesta.

La diferencia más relevante entre el robot real y el virtual es el escenario real ya que este tiene un nivel más alto de ruido, principalmente, en la velocidad angular leída por la odometría reportada por el sistema real P3DX.

Con lo aquí expuesto, podemos finalizar diciendo que se ha conseguido un modelo virtual valido del robot P3DX, pudiendo hacer pruebas en este robot virtual asegurándonos que se comportará de manera muy cercana a la realidad.

6.2. Guiado de un conjunto de robots: unidades P3DX y virtualización con ROS

El control que se aplica para la validación en un conjunto de robots es la idea de la Ilustración 4.7 en la que se utiliza un servosistema para cada robot.

A continuación se puede ver el diagrama de bloques de simulink:

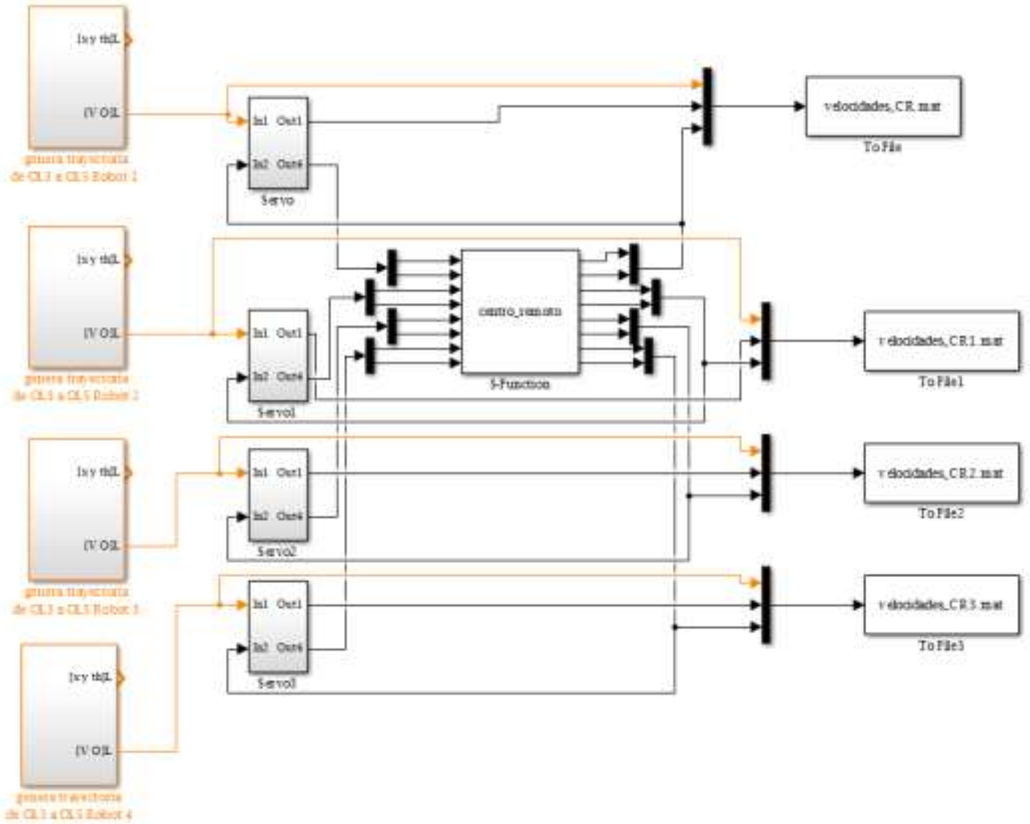


Ilustración 6.5: Diagrama de bloques de Simulink del control aplicado al conjunto de 4 robots.

Donde la S-Function es el único elemento que variará dependiendo si estamos validando el control en los robots virtuales o en los robots reales, enviando a una dirección IP para el caso del conjunto virtual y a 4 direcciones IPs para el caso del conjunto real.

Este control es ejecutado en el centro remoto, en Linux, a partir del diseño con Matlab/Simulink. Primero el centro remoto actúa sobre los robots virtuales en el correspondiente PC donde corren las herramientas ROS/Gazebo, y finalmente sobre las unidades robóticas reales.

Lo que se pretende es que los robots sigan alineados una trayectoria no lineal partiendo del laboratorio OL3 de la Escuela Politécnica Superior de Alcalá hasta el laboratorio OL5 mandando consignas de velocidad lineal y angular.

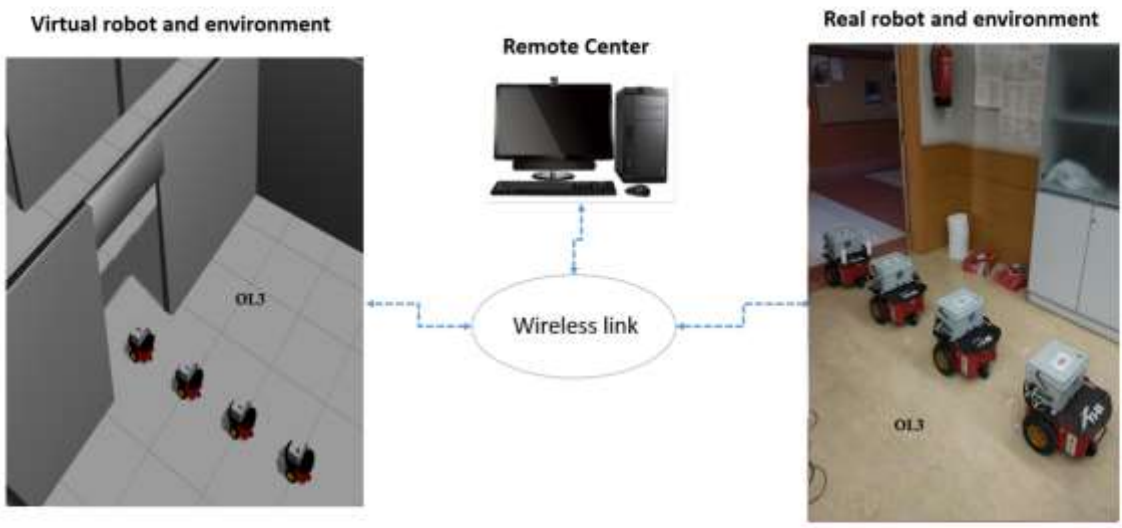


Ilustración 6.6: Captura del movimiento del conjunto virtual (izquierda) y real (derecha) a lo largo del pasillo Oeste en el punto de partida OL3.

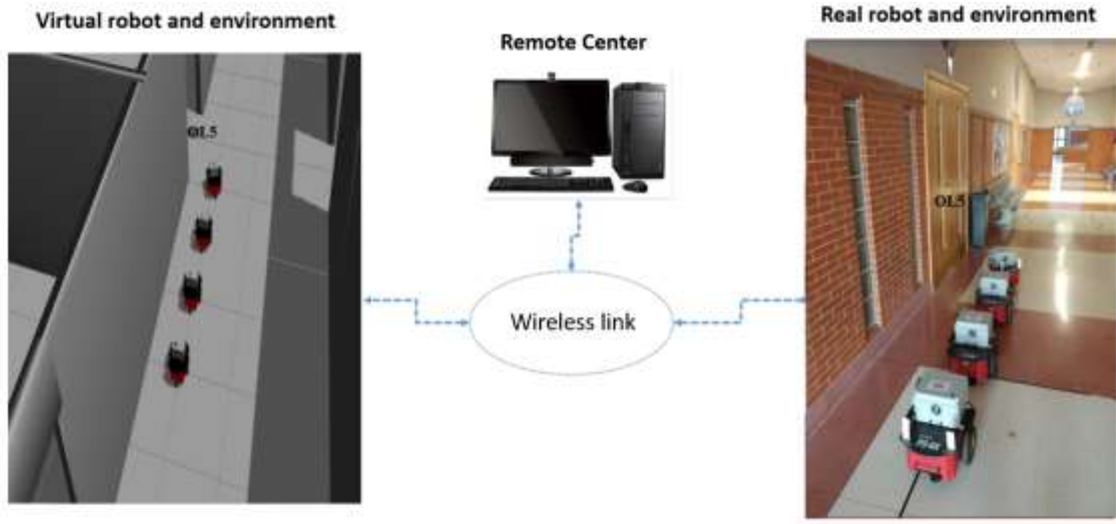


Ilustración 6.7: Captura del movimiento del conjunto virtual (izquierda) y real (derecha) a lo largo del pasillo Oeste cerca del punto objetivo OL5.

Al querer que los robots sigan la misma trayectoria uno detrás de otro, la consigna de velocidad lineal será idéntica para los 4 robots:

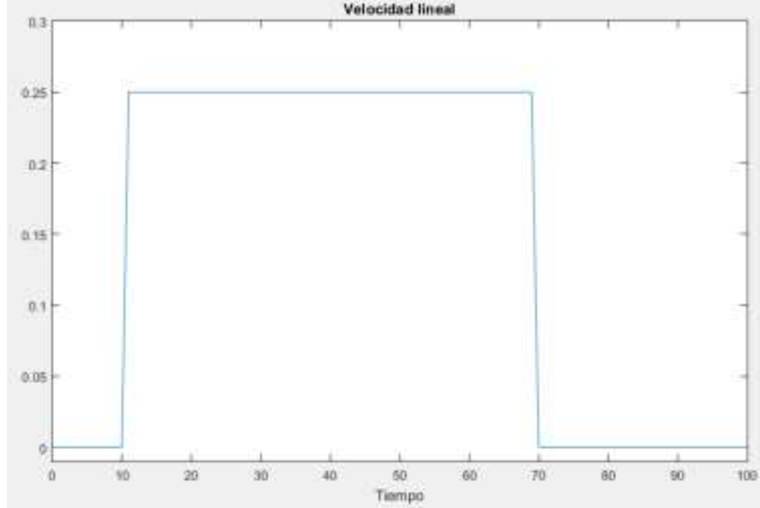


Ilustración 6.8: Consigna de velocidad lineal aplicada a cada robot.

Sin embargo, la velocidad angular tendrá que estar desfasada en el tiempo debido a que cada robot deberá girar a su respectivo tiempo dependiendo de la posición inicial de la que partan.

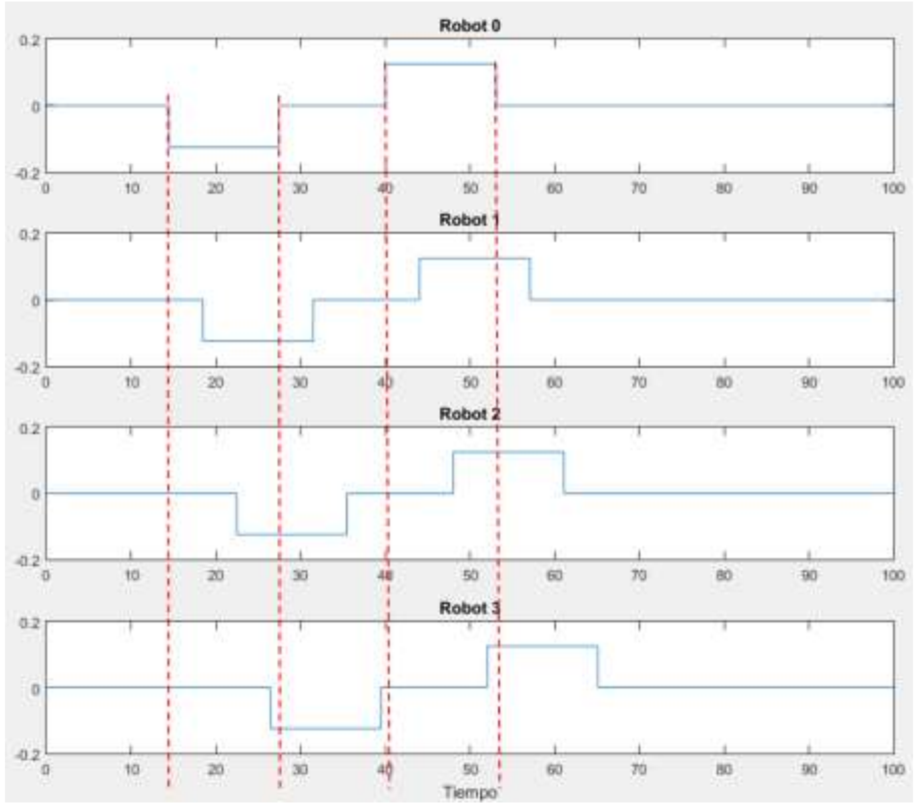


Ilustración 6.9: Consignas de velocidad angular aplicadas a los robots del conjunto.

Como vemos, el robot 0 (unidad líder) es el que gira primero. Después irían el robot 1, el robot 2 y el robot 3.

Los modelos virtuales son idénticos, sin embargo, los reales pueden tener algún tipo de variación en el peso, deformación de las ruedas… pero se puede asumir que también son idénticos. Por lo tanto, viendo que el control funciona, la respuesta de cada uno de los robots virtuales y reales (así como su explicación) es la misma que en la Ilustración 6.4 debido a que se realizó esta misma trayectoria pero para un solo robot.

Capítulo 7: Conclusiones

En este TFM se ha contribuido a la integración de instrumentos para el uso de la robótica, y sobre todo, para tareas que exigen un control remoto de varios robots. El trabajo presentado permite validar la solución de control diseñada en una herramienta matemática como es Matlab/Simulink, y ponerla en práctica en un robot virtualizado (dinámica y físicas), con herramientas potentes para este fin como son ROS y Gazebo, para después implementar el control en robots reales.

El trabajo realizado en un único robot virtual y real se ha extendido a un conjunto de unidades, implementando un algoritmo de seguimiento en el centro remoto.

También se ha propuesto un algoritmo de evasión de obstáculos a través de técnicas borrosas tipo Mamdani, el cual también se puede implementar primero en emulación (ROS/Gazebo) y después en ejecución con robot real.

El estudio en este trabajo es especialmente útil para tareas que exigen un control remoto de varios robots. Los elementos claves en esta comunicación son el uso de aplicación cliente-servidor a través de sockets de comunicación entre el centro remoto (ejecutando la ley de control) y los robots real y virtual. Así como el empleo de nodos de ROS para el intercambio de información entre centro remoto y robot virtual. En cualquier caso, el intercambio de información con el centro remoto se programa en la S-Function de Matlab/Simulink.

La posibilidad de validar cualquier diseño de robótica (diferentes niveles de control, generación de trayectorias, evitación de obstáculos, etc) en un entorno emulado es una idea atractiva para la enseñanza y la investigación. Además, los ajustes apropiados sobre los controladores en un mundo virtualizado antes de los experimentos en el mundo real, siempre significan ahorros de recursos y tiempo.

7.1. Trabajo futuro

En cuanto a la virtualización de un conjunto de robots en un mismo entorno, se ha desarrollado un único canal de comunicación entre el PC centro remoto y el PC de ROS, siendo este el encargado de gestionar el envío y recepción de velocidades de cada uno de los robots. Como trabajos futuros se proponen varias alternativas para realizar el control remoto de varias unidades robóticas:

- Tener una única IP para el conjunto virtual (un único PC-ROS) pero enviando a tantos puertos (sockets) distintos como robots tengamos, haciendo así que haya tantos nodos de comunicación como robots haya.
- Tener múltiples direcciones IPs, una por cada PC virtualizando a un único robot, que es lo que se tiene en un conjunto de robots reales.

También, se propone replicar de forma virtual (diseñado en ROS) el control realizado en la referencia [22], de forma que cada unidad implementa un control local dedicado a mantener la distancia de seguridad de los seguidores del convoy. Mientras que el centro remoto se encarga del correcto seguimiento de la trayectoria de todas las unidades robóticas.

En este trabajo todos los controles se han realizado de forma periódica, pero en un futuro se pretende realizar los controles de forma aperiódica (asíncrona) con el objetivo de mejorar la disponibilidad de los recurso de comunicación, y con ello reducir los retardos y pérdida de paquetes en el canal de comunicación, tal y como se propone en la referencia [28].

ANEXO

Anexo 0: Implementación de S-Function y generación de ejecutable con RTW

- <u>Compilación del fichero C como MEX file</u>

Para incorporar el fichero en C en la S-Function, se debe de compilar para asegurarnos que no tenga errores. Como se dijo antes, para que Simulink reconozca el programa desarrollado se debe de tratar como un fichero MEX. Para compilarlo se ejecuta desde la consola de Linux el comando "make", y el fichero Makefile, que indica al compilador gcc y MEX como realizar su tarea, está detallado a continuación:

```
MEXSUFFIX  = mexglx
MATLABHOME = /opt/matlab
MEX        = mex
CXX        = gcc-4.1
CFLAGS     = -fPIC -pthread -DMX_COMPAT_32 -DMATLAB_MEX_FILE
LIBS       = -L/usr/local/lib -lm -lpthread -ldl -L/usr/realtime/lib -llxrt
INCLUDE1   = -I$(MATLABHOME)/extern/include -I/usr/realtime/include
INCLUDE2   = -I$(MATLABHOME)/simulink/include
MEXFLAGS   = -cxx CC='$(CXX)' LD='$(CXX)'

example.$(MEXSUFFIX): example.o
        $(MEX) $(MEXFLAGS) $(LIBS) -output example $^

example.o: example.c
        $(CXX) $(CFLAGS) $(INCLUDE1) $(INCLUDE2) -c $^

clean:
        rm -rf *.o
        rm -rf *.mexglx
```

En "example" ponemos el nombre de nuestro fichero creado en C. Este proceso creará un archivo con extensión mexglx.

- <u>Integración de la S-Function en Simulink</u>

Lo siguiente que debemos de hacer es indicar al bloque S-Function que programa queremos que ejecute en él, que en este caso será "example":

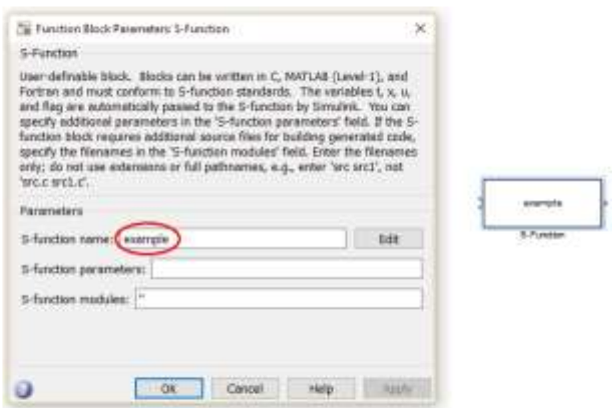


Ilustración 0.1: Configuración del bloque S-Function.

- **Configurar el entorno Simulink**

También debemos de configurar el entorno de Simulink para definir el período de muestro, el tiempo de ejecución y el tiempo final de la ejecución.

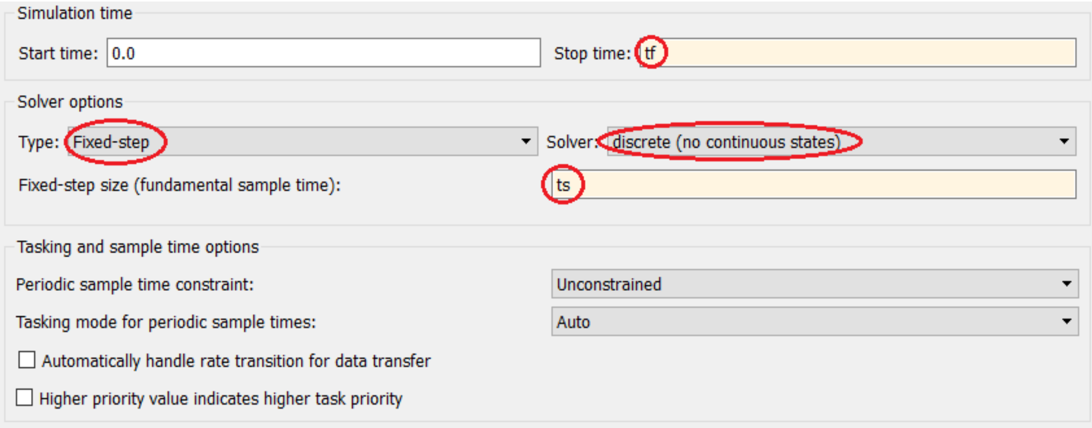


Ilustración 0.2: Configuración Simulink.

Por otro lado, se debe establecer la configuración de RTW, en la cual se debe de indicar que se desea utilizar el target RTAI y la plantilla del Makefile descrita a utilizar, que en este caso al utilizar RTAI es rtai-custom.tmf, el cual indica como debe RTW compilar y linkar los programas y librearías de RTAI. A continuación se muestra esta configuración:

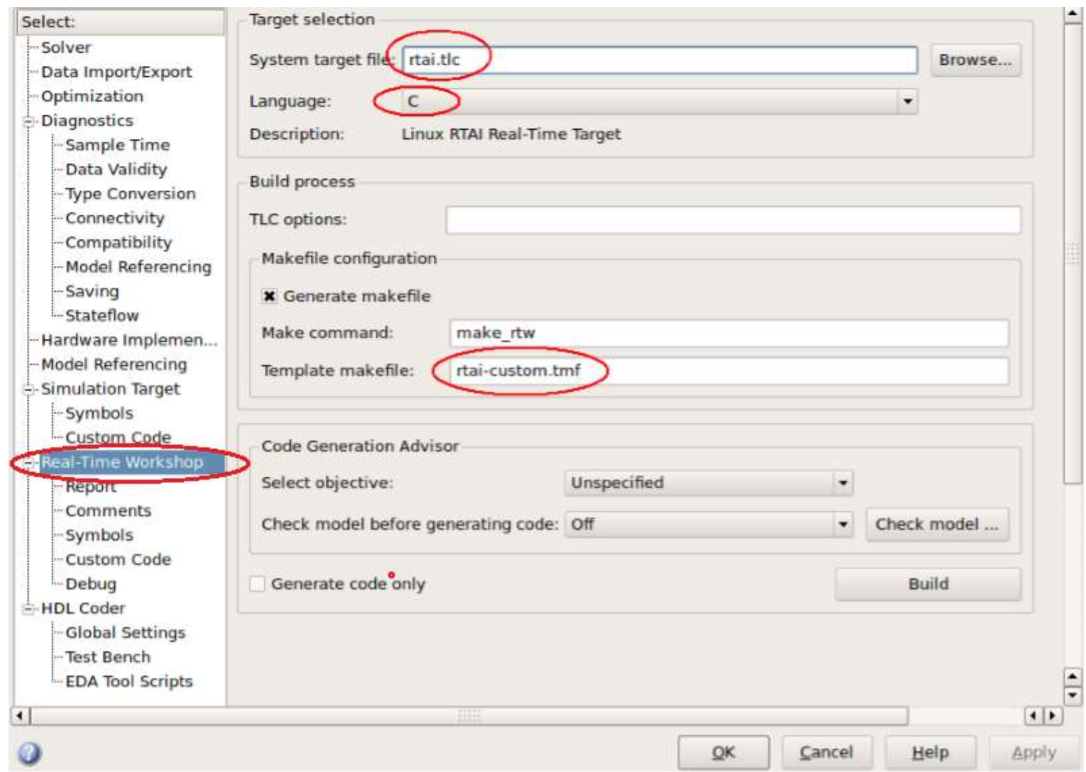


Ilustración 0.3: Configuración de RTW en entorno Simulink.

- **Generación de código**

Para iniciar el proceso de generación de código, únicamente se debe de acceder al menú de Simulink "Tools-RealTimeWorkshop-BuildModel", o también pulsando Ctrl+B.

En este proceso aparecen con frecuencia errores de compilación que a veces no se detectan en la generación del fichero "mexglx" cuando compilamos el archivo en C, por lo que es necesario volver a editar este archivo, compilarlo desde la consola y volver a intentar la generación de código con RTW si esto sucediera.

En la siguiente figura se muestra el resultado de una generación de código exitosa:

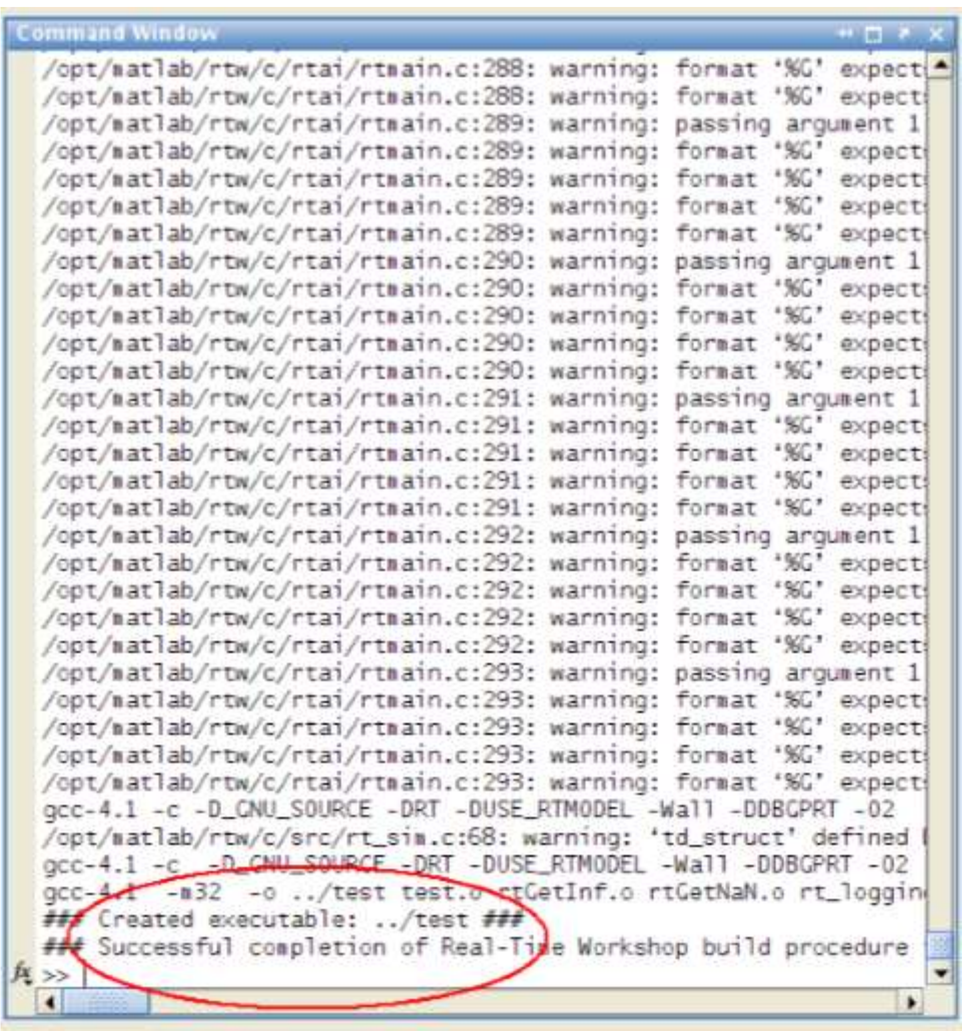


Ilustración 0.4: Resultado exitoso en el proceso de generación de código.

Al finalizar el proceso, este crea un ejecutable, el cual es el archivo final del proceso.

Este ejecutable tiene extensión .exe en Windows y sin extensión en Linux como es nuestro caso. El nombre del ejecutable creado se llama test como podemos ver en la ilustración 0.4.

- **Correr el ejecutable**

Una vez se tiene el ejecutable final creado, se debe de lanzar desde el terminal de Linux junto con los argumentos –v –f tf, donde tf es el tiempo de simulación en segundos.

A continuación se detalla el lanzamiento del ejecutable en tiempo real:

```
Target settings
===============
  Real-time : HARD
  Timing    : internal / periodic
  Priority  : 0
  Finaltime : 20.000000 [s]
  CPU map   : f

Target info
===========
  Model name              : test
  Base sample time        : 0.100000 [s]
  Number of sample times  : 1
  Sample Time 0           : 0.100000 [s]

Target is running.
Final time occured.
Target is stopped.
Target is terminated.
```

Ilustración 0.5: Lanzamiento de ejecutable creado con Matlab/Simulink/RTW.

En nuestro caso, al lanzar el ejecutable creado en Matlab/Simulink, comenzarán a enviarse y recibirse las velocidades tanto de control como las leídas de la odometría por el robot virtual. Antes de lanzar este ejecutable, el PC que simula el robot virtual debe estar preparado para la comunicación, es decir, se debe de lanzar el nodo “lanzando_vel” explicado en el apartado 3.3.1 antes que este ejecutable. Este proceso se detalla paso por paso en el apartado 3.6.1.

Bibliografía

[1] «Página Oficial de MARTe,» [En línea]. Available: www.omg.org/omgmarte/.

[2] E. Eberbach y S. Phoha, «SAMON: communication, cooperation and learning of mobile autonomous robotic agents,» *Tools with Artificial Intelligence,* 1999.

[3] S. Carpin, M. Lewis, J. Wang, S. Balarkisky y C. Scrapper, «USARSim: a robot simulator for reseach and education».

[4] «Página Oficial de Robot Operating System,» [En línea]. Available: www.ros.org.

[5] «Plataformas robóticas soportadas por ROS,» Septiembre 2016. [En línea]. Available: http://wiki.ros.org/Robots.

[6] «Página Oficial de Gazebo,» [En línea]. Available: www.gazebosim.org/.

[7] R. Bucher, «Rapid Controller Proto-typing Qith Matlab/Simulink and Linux,» *Oulu Findland,* 2003.

[8] R. Bucher y L. Dozio, «CACSD under RTAI Linux With RTAI-LAB».

[9] G. Quaranta y P. Mantegazza, «Using Matlab-Simulink RTW to Build Real Time Control applications in User Space with RTAI,» *Real Time Linux Workshop,* 2001.

[10] M. Salazar, Integración de Matlab/Simulink/RTW y Linux/RTAI para aplicaciones de control y comunicación en robtótica, Trabajo Fin de Carrera. Ingeniería Electrónica. EPS. Marzo, 2011.

[11] «Datasheet P3-DX,» 2016. [En línea]. Available: http://www.mobilerobots.com/Libraries/Downloads/Pioneer3DX-P3DX-RevA.sflb.ashx.

[12] «Tutorial links y joints Gazebo,» [En línea]. Available: http://gazebosim.org/tutorials?tut=ros_urdf&cat=connect_ros.

[13] C. Santos, «Adaptative self-triggered control of a remotely operated P3DX robot,» *Simulation and experimentation, Robotics and Autonomous Systems,* 2014.

[14] Á. Salcedo , Control Basado en Lyapunov para Robot P3DX, Alcalá: Universidad Politécnica Superior, 2014.

[15] W. F. Arnold y A. J. Laub, «Generalizad eigenproblem algorithms and software for algebraic Riccati equations,» *Proceedings of the IEEE,* vol. 72, nº 12, pp. 1746-1754, 1984.

[16] S. S. Ge, «Lyapunov Design,» *(EOLSS) Control Systems, Robotics and Automation,* 2004.

[17] M. García y A. Barreiro, «Análisis de la Estabilidad según Lyapunov de un Control Borroso en Tiempo Discreto,» Escuela Superior de Ingenieros Industriales y de Minas, Vigo, España.

[18] W. Gay, Linux Socket Programming by Example, Macmillan Computer Publishing, 2000.

[19] «S-Function de Matlab,» 2016. [En línea]. Available: http://es.mathworks.com/help/simulink/create-cc-s-functions.html.

[20] Player, «Página Oficial de Player,» [En línea]. Available: http://playerstage.sourceforge.net/doc/Player-cvs/player/index.html.

[21] MobileRobots, Pioneer 3 Operations Manual v3 (ARCOS-based DX and AT).

[22] C. Santos, Propuesta de control descentralizado con solapamiento para guiado en convoy de unidades P3-DX, UAH, Depeca, 2010.

[23] L. Tang, S. Dian, G. Gu, K. Zhou, S. Wang y X. Feng, «A Novel Potencial Field Method for Obstacle Avoidance and Path Planning of Mobile Robot,» *Sichuan University, Chengdu, China.*

[24] J. Borestein y Y. Koren, «The Vector Field Histogram-fast Obstacle Avoidance for Mobile Robots,» *IEEE Transactions on Robotics,* vol. 7, nº 3, pp. 278-288, 2002.

[25] M. Zohaib, S. Mustafa Pasha, N. Javaid y J. Iqbal, «Intelligent Bug Algorithm (IBA): A Novel Strategy to Navigate Mobile Robots Autonomously».

[26] S. Hong Lian, «Fuzzy Logic Control of an Obstacle Avoidance Robot,» *Fuzzy Systems, Proceedings of the Fifth IEEE International Conference,* vol. 1, pp. 26-30, 1996.

[27] M. Zohaib, M. Pasha, R. A.Riaz, N. Javaid, M. Ilahi y R. D.Khan, «Control Strategies for Mobile Robot with Obstacle Avoidance».

[28] C. Santos, F. Espinosa, E. Santiso y M. Mazo Jr, «Aperiodic Linear Networked Control Considering Variable Channel Delays: Application to Robots Coordination,» *Sensors,* vol. 15, 2015.

[29] R. Ragel De la Torre, «Entornos de simulación,» de *Modelado, Control y Simulación de un quadrotor equipado con un brazo manipulador robótico*, PFC, pp. 65-118.

[30] V. García Diaz, «Desarrollo Robótico, Robot Operating System (ROS),» Universidad de Oviedo, 2012.

[31] M. Quigley, K. Conley, B. Gerkey, J. Faust, T. Foote, J. Leibs, R. Wheeler y A. Y Ng, «ROS: an open-source robot operating system,» *IEEE ICRA workshop on open source software,* vol. 3, nº 3.2, 2009.

[32] P. Corke, «integrating ROS and Matlab (ROS Topics),» *IEEE Robotics & Automation Magazine,* vol. 22, nº 2, pp. 18-200, 2015.

[33] R. Reid, A. Cann, C. Meiklejohn, L. Poli, A. Boeing y T. Braunl, «Cooperative multirobot navigation, exploration, mapping and object detection with ROS,» *IEEE Intelligent Vehicles Symp osiom,* vol. 4, nº June, pp. 23-26, 2013.

[34] C. Santos, M. Mazo Jr y F. Espinosa, «Lectures Notes in Computer Science. Advances in Autonomous Robotics,» de *Adaptative Self-triggered Control of a Remotely Operated Robot*, Springer-Verlag, 2012.

[35] C. Santos, M. Mazo Jr y F. Espinosa, «Adaptative Self-triggered Control of a Remotely Operated Robot P3-DX Robot: Simulation and Experimentation,» *Robotics and Autonomous Systems,* vol. Elsevier, nº March, 2014.

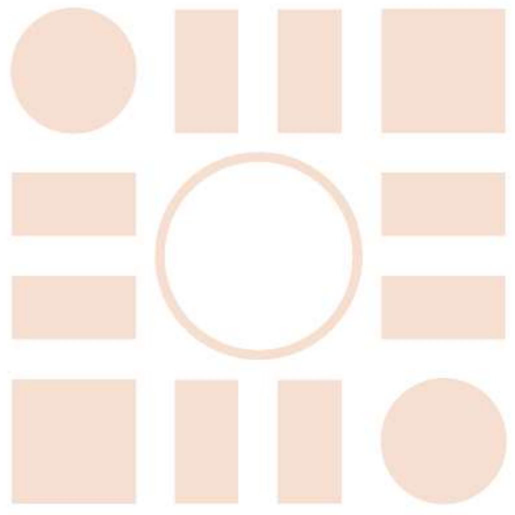
[36] B. Lacevic, J. Velagic y N. Osmic, «Design of Fuzzy Logic Based Mobile Robot Position Controller Using Genetic Algrithm,» *IEEE/ Asme Internation Conference on Advanced Intelligent Mechatronics,* pp. 1-6, 2007.

[37] j. Jassbi, P. Serra, R. Ribeiro y A. Donati, «A Comparison of Mandani and Sugeno Inference Systems for a Space Fault Detection Application,» *Word Automation Congress,* 2006.

[38] W. Levine, The control handbook, IEEE-Press, 1996.

Universidad de Alcalá

Escuela Politécnica Superior



ESCUELA POLITECNICA

SUPERIOR