UNIVERSIDAD
DE MÁLAGA

E.T.S.
INGENIERÍA
INFORMÁTICA

ESCUELA TÉCNICA SUPERIOR DE INGENIERÍA INFORMÁTICA

GRADO EN INGENIERÍA INFORMÁTICA

APRENDIZAJE POR REFUERZO PARA LA MANIPULACIÓN DE OBJETOS POR UN BRAZO ROBÓTICO

REINFORCEMENT LEARNING FOR OBJECT MANIPULATION BY A ROBOTIC ARM

Realizado por

Sergio González Muriel

Tutorizado por

Dr. Juan Antonio Fernández Madrigal

Dra. Ana Cruz Martín

Departamento

Ingeniería de Sistemas y Automática

UNIVERSIDAD DE MÁLAGA

MÁLAGA, SEPTIEMBRE DE 2019

Fdo. El/la Secretario/a del Tribunal

Resumen

El objetivo de este proyecto consiste en la implementación de un algoritmo de Aprendizaje por Refuerzo (AR) para un robot formado por una plataforma móvil y un brazo robótico, para realizar una comparación con otros modelos y con una programación explícita. Además, buscábamos la puesta a punto de una plataforma de experimentación, ya que el AR no se ha utilizado todavía en el brazo manipulador de dicho robot. Para ello, hemos utilizado el *framework* de desarrollo robótico *Robot Operating System* (ROS), con el lenguaje de programación C++.

En concreto, hemos implementado el algoritmo de AR *Q-Learning*, mostrando ventajas sobre la programación explícita. La principal ventaja que aporta este algoritmo es la amplitud de situaciones abarcadas por el robot sin tener que tenerlas en cuenta en la implementación.

También hemos puesto a punto una plataforma de experimentación con el simulador Gazebo y la del robot real (aunque finalmente solo se ha trabajado con la simulada, para no dañar el robot real durante el proceso de aprendizaje), así como ajustado los parámetros del algoritmo y realizado experimentos para un correcto aprendizaje.

Palabras clave: Aprendizaje por Refuerzo, robot, Q-Learning, C++, ROS

Abstract

The aim of this project has been the implementation of a Reinforcement Learning (RL) algorithm for a robotic platform with a robotic arm, so it can be compared to other models and also to explicit programming. In addition, we have tuned up a test-bed because the RL has not been tested on the manipulator arm of such robot. We have used the robotic development framework ROS, with the C++ programming language.

We have implemented the RL algorithm Q-Learning, which shows some advantages over the explicit programming. The main advantage is the amplitude of situations covered by the robot without the need of taking them into account in the implementation.

We have also refined a experimentation platform based on the Gazebo simulator and the real robot - although we have only worked with the simulated environment, in order to not damaging the real robot during the learning process - and we have also adjusted the algorithm parameters and performed experiments for a correct learning.

Keywords: Reinforcement Learning, robot, Q-Learning, C++, ROS

Índice general

Índice de tablas

Índice de figuras

Índice de acrónimos

AA Aprendizaje Automático

AP Aprendizaje Profundo

AR Aprendizaje por Refuerzo

CRUMB *Cognitive-Robotics-sUpporting Mobile Base*

FTDI *Future Technology Devices International*

HSV Matiz, Saturación, Valor

MCD Modelo Cinemático Directo

MCI Modelo Cinemático Inverso

LTS *Long Term Support*

OpenCV *Open Source Computer Vision Library*

ROS *Robot Operating System*

TFG Trabajo de Fin de Grado

TFM Trabajo de Fin de Máster

UMA Universidad de Málaga

USB *Universal Serial Bus*

V-REP *Virtual Robot Experimentation Platform*

Capítulo 1

Introducción

1.1. Motivación

La manipulación inteligente de objetos por parte de robots es un campo de investigación muy activo hoy en día y que tiene un enorme potencial todavía por descubrir. Además, tiene un gran número de aplicaciones posibles, desde ayudar a mejorar la calidad de vida a un disminuido físico hasta la manipulación de objetos peligrosos como explosivos, sin la necesidad de que el programador tenga que tener en cuenta todas las posibles situaciones.

El objetivo principal de la manipulación inteligente es, mediante técnicas de Aprendizaje Automático (AA), que el robot aprenda a realizar dicha manipulación mejorando sus parámetros de funcionamiento durante la misma, así como dotarlo de mayor adaptabilidad a situaciones imprevistas, cuya totalidad sería difícil de recoger en una programación explícita clásica.

Dentro de las diversas metodologías de AA disponibles, el AR [1] permite que el agente (el que realiza la acción), en este caso nuestro robot, desarrolle por sí mismo y en base a su propia experiencia y a una serie de recompensas que obtiene, la capacidad de operar en su entorno de forma cercana a la óptima. Al contrario que en otra metodología de aprendizaje también en auge, el Aprendizaje Profundo (AP), en el AR no es necesario disponer de ingentes cantidades de datos previamente tratados para el entrenamiento, y el agente puede seguir aprendiendo de manera mucho más natural y suave a lo largo de toda su vida útil.

Aunque el AR ha sido utilizado en multitud de disciplinas para la toma automática de decisiones (ingeniería civil, ecológica, finanzas y economía, por ejemplo [2]), en Robótica no está tan extendido, fundamentalmente debido a la interacción del agente con un entorno

físico en lugar de con uno puramente computacional [3]. Aunque se están realizando esfuerzos importantes para solventar estos problemas (p.ej., en el mismo grupo de los directores de este Trabajo de Fin de Grado (TFG) [4]), aún no se disponen de análisis científicos que establezcan de manera experimental y general las particularidades que hacen que el AR en Robótica sea exactamente igual que en otras disciplinas. Por ejemplo, se cree que en su aplicación a robots es más adecuado utilizar métodos basados en modelos (model-based), pues la alternativa (model-free) parece tener mayor tiempo de aprendizaje, pero no existen estudios enfocados en estas afirmaciones, sólo conclusiones fruto de experiencias particulares.

Es importante, por tanto, seguir investigando en la aplicación del AR en robots con el fin de, por un lado, dar solidez y generalidad a sus particularidades, y por otro, conseguir dispositivos más adaptables a largo plazo y a la diversidad de tareas que deben llevar a cabo, que abaratarían costes debido al menor esfuerzo de programación para ser puestos en marcha.

1.2. Estado del arte

En este TFG vamos a trabajar con un tipo de aprendizaje llamado AR. Este aprendizaje se basa en el aporte de refuerzo positivo o negativo a las acciones tomadas por el agente, según realice la tarea asignada o no.

Así, se distinguen dos tipos bien diferenciados:

- **Basados en modelos (*model-based*):** Estos métodos trabajan con un modelo del entorno en el que se sitúan, aprendiendo a construirlo para luego utilizarlo en su tarea. Esto le confiere más rapidez al método con respecto a los modelos libres, sin embargo, un modelo equivocado puede hacer que el aprendizaje sea fatal. [5]
- **Libres de modelos (*model-free*):** Estos otros modelos aprenden directamente de la experiencia, sin valerse de ningún otro modelo previamente formado. Debido a esto, son métodos más lentos que los anteriores, ya que tienen que ejecutar las acciones y obtener información de éstas, pero si se produce algún error en el aprendizaje, se puede autocorregir. [5]

El algoritmo utilizado en este proyecto es llamado *Q-Learning*, y es un método de AR libre de modelos (*model-free*). Este es uno de los métodos más simples de aprendizaje por refuerzo, el cual utiliza una matriz de búsqueda llamada *Q-Table* para guardar la calificación

que va obteniendo al realizar cada acción en un estado concreto. Las acciones en nuestro caso son girar la base del robot un poco a la izquierda, a la derecha, moverla hacia delante, hacia atrás o mover el brazo hacia el objeto. Nuestro estado está formado por valores discretos de la distancia, ángulo y altura del objeto con respecto al robot, si el objeto ha sido alcanzado o no y si el brazo manipulador está plegado.

Así, su éxito está sujeto a la elección precisa de los siguientes parámetros:

- **Recompensa:** la tarea a realizar por nuestro agente está definida por la correcta elección de recompensas en función del estado actual, la acción ejecutada y el nuevo estado.
- **Estrategia de exploración/explotación:** decide si el agente debe explotar la política ya aprendida o explorar nuevas acciones.
- **Ratio de aprendizaje (α):** establece cómo influencia lo ya aprendido al aprendizaje. Toma valores entre 0 y 1, significando 0 que no existe aprendizaje y 1 que solamente los nuevos conocimientos son tomados en cuenta.
- **Ratio de descuento (γ):** regula cómo influencia la recompensa a los valores de la *Q-Table* en siguientes pasos. Cuando este valor es 0, solamente se tienen en cuenta recompensas inmediatas, mientras que un valor cercano a 1 permite estrategias a más largo plazo, aunque también implicará un proceso más largo de aprendizaje para una política correcta [6].

La fórmula utilizada para actualizar los valores de la matriz es la siguiente:

$$Q_k(s,a) = (1-\alpha_k) * Q_{k-1}(s,a) + \alpha_k[R(s,a,s') + \gamma * V_{k-1}(s')] \tag{1.1}$$

siendo:

$Q_k(s,a)$: el nuevo valor de la matriz en el paso *k*, estando en el estado *s* y realizando la acción *a*.

$Q_{k-1}(s,a)$: el antiguo valor de la matriz en el paso *k-1*, estando en el estado *s* y realizando la acción *a*.

$R(s,a,s')$: la recompensa obtenida al transitar al estado *s'* estando en el estado *s* y ejecutando la acción *a*.

$V_{k-1}(s')$: equivalente a $max_{a'}Q_{k-1}(s',a')$, es decir, el valor máximo de la matriz en el siguiente estado.

1.3. Objetivos

El objetivo principal de este TFG es la implementación y configuración de métodos de AR para tareas de manipulación robóticas. Este objetivo se puede desglosar en los siguientes, planteados inicialmente:

1. Comparación del desempeño de métodos sin modelo (*model-free*) frente al de métodos basados en modelos (*model-based*), obteniendo medidas científicas de en qué situaciones y bajo qué circunstancias los segundos mejoran el proceso de aprendizaje.

2. Puesta a punto de una plataforma de experimentación simulada y real para algoritmos de AR. El grupo de los tutores del TFG dispone de un robot con manipulador previamente subvencionado por la Universidad de Málaga (UMA) [7], así como de algunos simuladores del mismo, pero no se ha utilizado aún el AR en el manipulador de a bordo.

3. Demostración, en tareas concretas, de las ventajas del AR en la manipulación robótica respecto a la programación explícita.

A continuación se detalla el nivel de cumplimiento de estos objetivos:

1. El primer objetivo no ha llegado a completarse debido a problemas encontrados durante la implementación de los algoritmos, por lo que solamente se ha podido realizar la implementación del algoritmo de AR libre de modelos (*model-free*) *Q-Learning*.

2. Este segundo objetivo sí ha sido completado con éxito, teniendo que realizar pequeñas modificaciones para el completo funcionamiento en la plataforma de experimentación real.

3. El algoritmo aprende en un grado observable, pero sería necesario un estudio más detallado para poder afirmar con certeza que el aprendizaje es correcto. Se han identificado, encontrado y resuelto multitud de problemas inesperados.

1.4. Estructura de la memoria

Este TFG se ha estructurado de la siguiente manera:

- **Capítulo 1:** en el capítulo actual, hemos presentado los motivos que nos han llevado a realizar este TFG, además del estado del arte del proyecto, explicando en qué consiste el modelo de aprendizaje utilizado. También se han introducido los objetivos del mismo.
- **Capítulo 2:** en este capítulo detallaremos las especificaciones del robot utilizado en nuestro TFG (el robot CRUMB), el *framework* de desarrollo utilizado (ROS) y una librería de C++ utilizada para la localización y detección del objeto (OpenCV).
- **Capítulo 3:** en este capítulo se describirá todo el proceso seguido para la implementación del aprendizaje, desde comprobaciones iniciales hasta la elección de los valores de los parámetros de aprendizaje, pasando por algunos problemas planteados.
- **Capítulo 4:** en este capítulo realizaremos una serie de simulaciones con diferentes recompensas para mostrar el efecto de esas modificaciones en el aprendizaje, además de un análisis básico de la política aprendida.
- **Capítulo 5:** en este último capítulo se mostrarán las conclusiones del trabajo, así como los posibles trabajos futuros.
- **Apéndice A. Manual de instalación:** en este primer anexo especificaremos cómo instalar un entorno propicio para la ejecución del algoritmo de aprendizaje implementado, pasando por la instalación de ROS y Gazebo en Ubuntu, además de cómo solventar algunos errores que nos podrían aparecer durante dichas instalaciones.
- **Apéndice B. Código C++:** en este último anexo se mostrarán los códigos desarrollados en C++ para la detección del objeto y de su profundidad, el cálculo del MCI del robot, el cuerpo del aprendizaje y la generación automática de simulaciones. Se han incluido porque son los códigos más relevantes de nuestro trabajo.

Capítulo 2

Herramientas utilizadas

En este capítulo se presentarán las herramientas utilizadas más importantes en la implementación de nuestro algoritmo.

Comenzaremos presentando el robot CRUMB [8], así como algunas de sus características técnicas. Este robot se compone de una base móvil *Turtlebot 2* [9] y un brazo manipulador *Widow-X* [10].

A continuación, explicaremos qué es ROS [11], así como sus ventajas y funcionamiento interno, haciendo hincapié en aquellos puntos que lo hacen idóneo para el campo de la robótica. También se explicará en este apartado el simulador 3D utilizado.

Para finalizar, se presentará una librería de C++ utilizada para la detección del objeto a manipular mediante procesado de imágenes, llamada *Open Source Computer Vision Library* (OpenCV) [12].

2.1. *Cognitive-Robotics-sUpporting Mobile Base* (CRUMB)

Se compone de dos partes: una plataforma móvil *Turtlebot 2* [9] y un brazo manipulador *Widow-X* [10] (Figura 2.1).

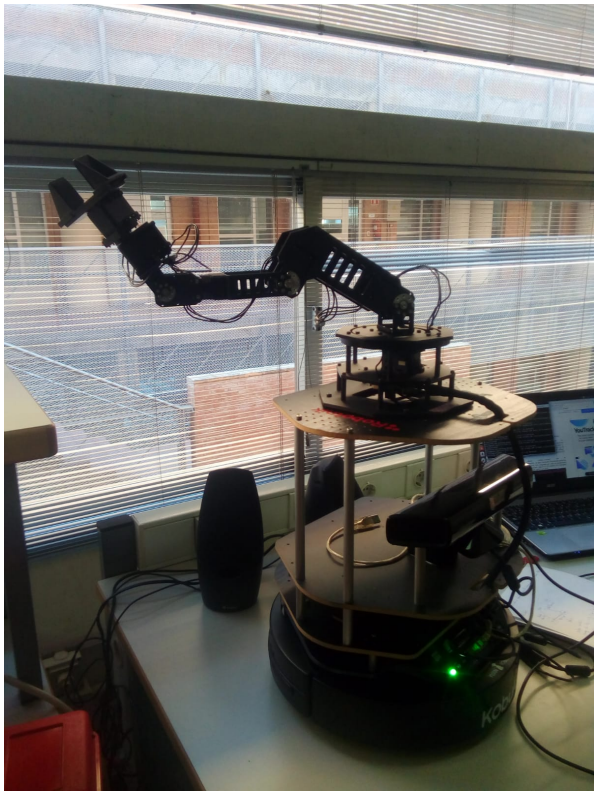


Figura 2.1: Robot CRUMB.

El robot *Cognitive-Robotics-sUpporting Mobile Base* (CRUMB) [8] forma parte de un proyecto de investigación relativo a AA desarrollado en el Departamento de Ingeniería de Sistemas y Automática de la UMA.

2.1.1. Plataforma móvil *Turtlebot*

Turtlebot es una plataforma móvil de bajo coste y *software* de código libre totalmente compatible con ROS [13].

El diseño original está equipado con una cámara *Kinect* [14] destinada a la navegación y el mapeado de interiores (Figura 2.2). Sin embargo, a nuestro modelo se le ha añadido también un láser *hokuyo* [15] para dar soporte a la cámara.

Figura 2.2: Turtlebot [16].

Se puede montar de diversas maneras, consiguiendo diferentes configuraciones. Sin embargo, la más usual es la mostrada en la Figura 2.3

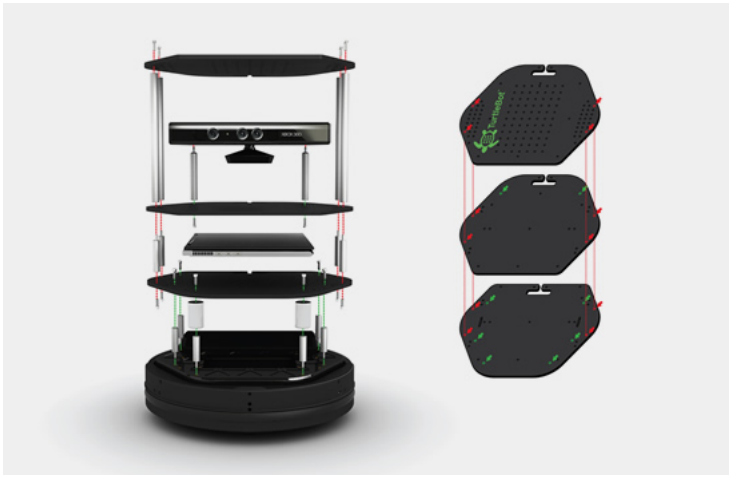


Figura 2.3: Montaje de Turtlebot [16].

Tiene unas dimensiones de 31.5 x 43 x 34.7 cm y un peso de 5 kg, además de una capacidad de carga de 5 kg en suelo rígido y 4 kg en alfombra. También soporta velocidades lineales de 0.7 m/s (no recomendándose su uso sobrepasando los 0.2 m/s por razones de seguridad) y velocidades angulares de 180 grados/s [17].

Esta plataforma está montada sobre una plataforma diferencial Kobuki (Figura 2.4), la cual contiene los siguientes sensores:

- 3 detectores de precipicios.
- 3 sensores de choque o *bumpers*.
- 1 giróscopo para odometría en un eje.
- 1 luz LED para indicar el estado de la batería.
- 2 LEDs programables.

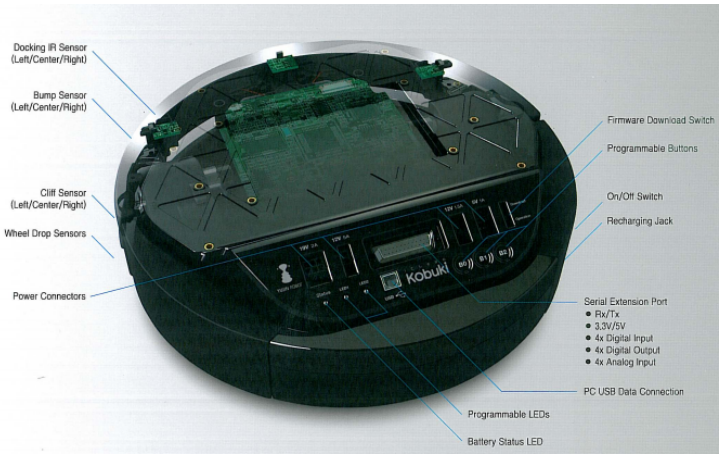


Figura 2.4: Base Kobuki [16].

La batería es una batería de Ión-Litio que proporciona 14,8 V y 4400 mAh. Además, la base contiene botones de encendido-apagado y varios puertos de conexión.

2.1.2. Brazo manipulador Widow-X

El brazo manipulador *Widow-X* es un brazo manipulador con 5 grados de libertad, una pinza paralela como efector final y una placa de Arduino como controlador [10].

Las articulaciones (joints) del brazo son de revolución y su rango comprende desde $-\pi/2$ a $\pi/2$. En la Figura 2.5 se muestra el sentido de giro de estas.

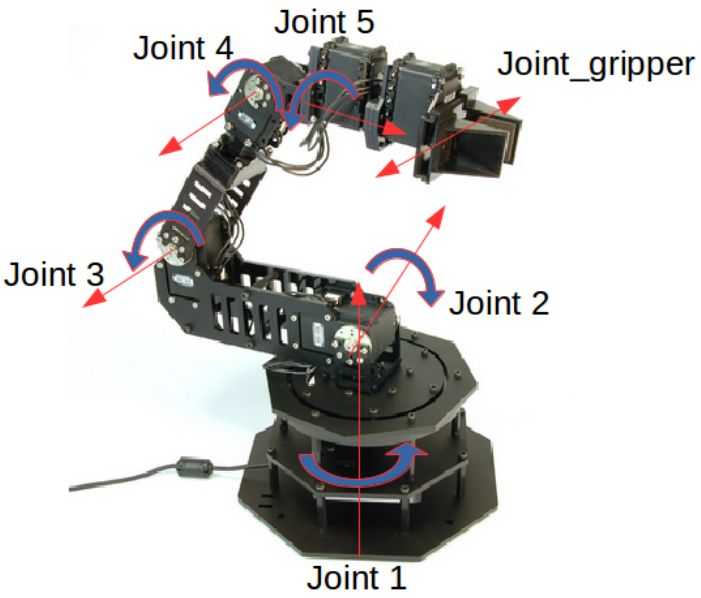


Figura 2.5: Sentido de giro de las articulaciones del brazo Widow-X. [18]

Los actuadores Dynamixel de estas articulaciones se muestran en la Figura 2.6, asociados a cada articulación.

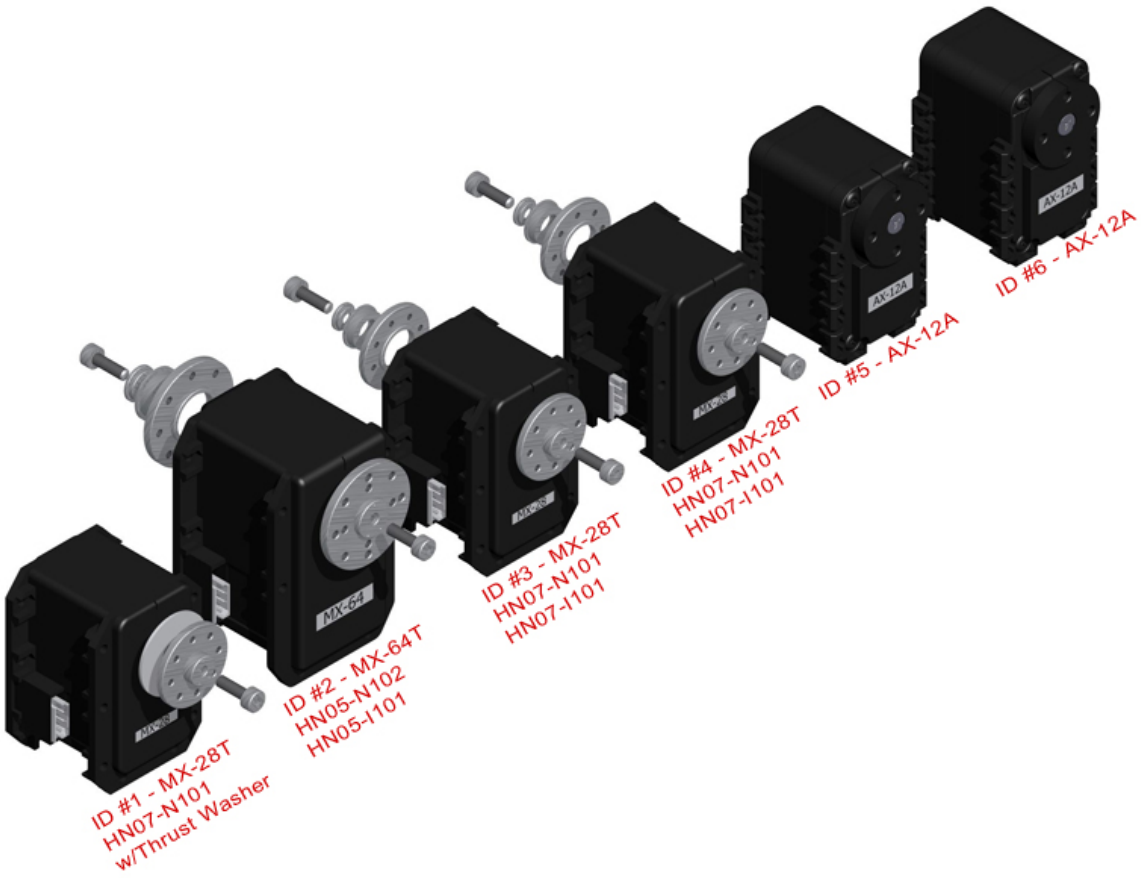


Figura 2.6: Actuadores DynaMixel del brazo Widow-X. [19]

Estos actuadores se diferencian en el par y velocidad máxima que alcanzan. En la Tabla 2.1 se muestran las características de cada motor.

	Articulaciones	Par(N*m)	Velocidad sin carga (rpm)	Velocidad máxima (rad/s)
AX-12	5 y pinza	1.5	59	1.7453
MX-28	1, 3 y 4	2.5	55	1.7453
MX-64	2	6	63	1.7453

Tabla 2.1: Características motores DynaMixel. [18]

La pinza, como muestra la Tabla 2.1, se mueve con un único actuador que controla ambas partes de esta. Dependiendo de la presencia o ausencia de almohadillas, la pinza puede tener diferentes distancias de apertura, mostradas en la Figura 2.7. En nuestro proyecto, la simulación trabaja con la pinza sin almohadillas.

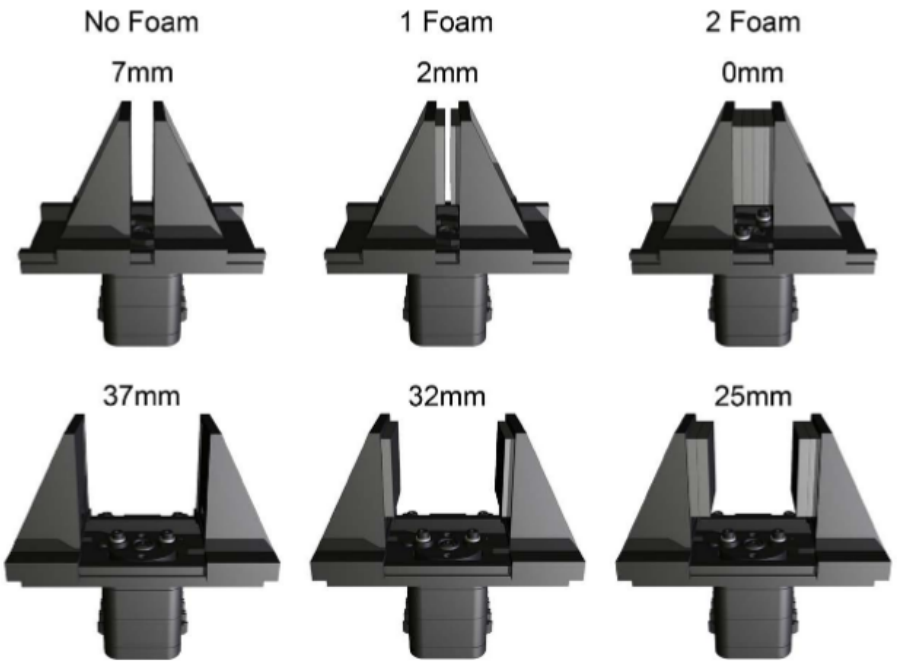


Figura 2.7: Apertura de la pinza. [20]

En cuanto a características técnicas, el brazo Widow-X cuenta con un peso de 1.33 kg, teniendo un alcance horizontal máximo de 41 cm con la pinza en horizontal y 29 cm con ella en vertical. El peso que soporta se muestra en la Figura 2.8, donde el eje horizontal indica la distancia de la pinza a la base del brazo, mientras que el vertical, la fuerza de agarre de la pinza, en gramos. En la Figura 2.8 se aprecia que el brazo puede levantar desde 800 a 400 g dependiendo de la distancia a la base. La pinza tiene una fuerza de agarre máxima de 500 g, pudiendo mover horizontalmente un objeto de 400 g.

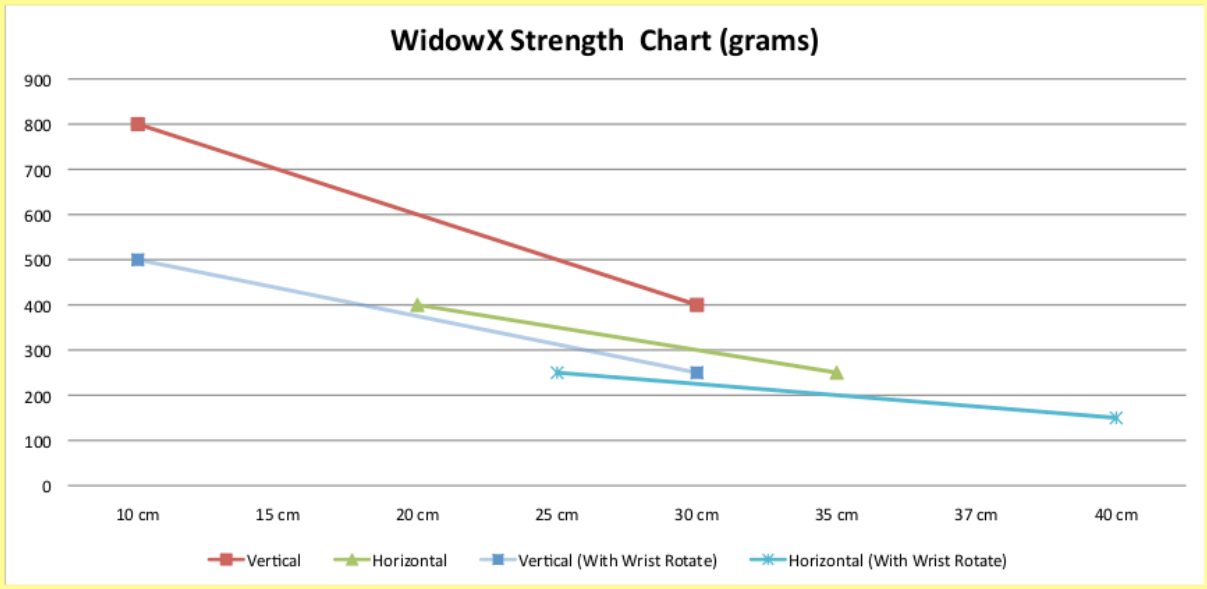


Figura 2.8: Fuerza del brazo Widow-X [20].

2.2. *Robot Operating System*

Robot Operating System (ROS) [11] es un *framework* de desarrollo de software que provee librerías y herramientas de ayuda para el desarrollo robótico. Programar aplicaciones robóticas de carácter general es muy complicado debido a sus múltiples variantes, como pueden ser el número de articulaciones del robot, su fisionomía o el espacio de trabajo. Estas variantes pueden provocar que un simple cambio en el entorno o en el propósito de la tarea, produzca un cambio completo del problema. Por esta razón, es muy complicado que una sola persona, o incluso un solo grupo, pueda implementar una tarea compleja desde cero.

Aquí es donde entra en juego ROS, ya que permite realizar *software* de una forma colaborativa gracias a su gran comunidad de usuarios que, mediante foros [21], blogs [22], *wikis* [11] y tutoriales [23] comparten conocimiento y aplicaciones desarrolladas, proporcionando las siguientes características:

- Abstracción de hardware.
- Controladores de dispositivos.
- Herramientas de visualización.

- Comunicación por mensajes.
- Administración de paquetes.

Hasta la fecha existen 12 distribuciones de ROS. En nuestro TFG hemos trabajado con ROS Indigo Igloo, el único compatible con *Turtlebot 2*. En la sección 3.1 se hablará sobre los problemas que encontramos al intentar usar otra versión de Ubuntu y de ROS, en especial Ubuntu 17.04 y ROS Melodic.

2.2.1. *Topics* y servicios

La programación en ROS se basa en programas interconectados entre sí llamados nodos (desarrollados en C++ o Python). Para que estos nodos se comuniquen entre sí, es necesario que se conecten a través de *topics* o de servicios:

- ***Topics:*** Son canales de comunicación unidireccionales, en los que uno o varios nodos publican y otros nodos pueden subscribirse para recibir dicha información. En la figura 2.9 se muestra un esquema de su funcionamiento.

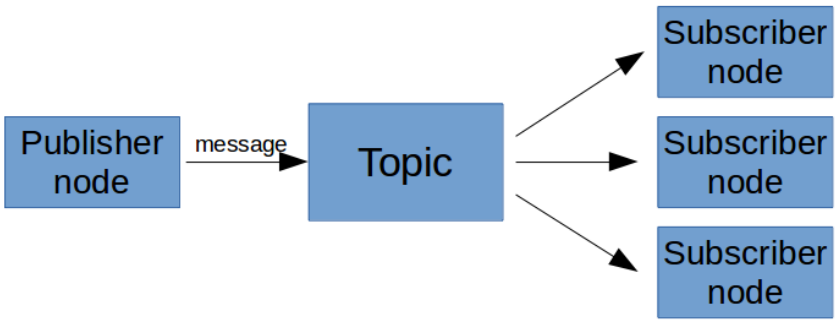


Figura 2.9: Esquema de funcionamiento de un *topic* ROS. [18]

- **Servicios:** Canal multidireccional en el que un nodo Cliente emite una petición y el nodo Servidor la realiza y devuelve el resultado. Esta información se envía y recibe de manera síncrona. En la Figura 2.10 se muestra un esquema de su funcionamiento.

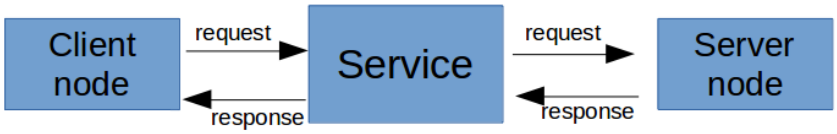


Figura 2.10: Esquema de funcionamiento de un servicio ROS. [18]

2.2.2. Sistema de ficheros en ROS

Para trabajar con ROS, necesitamos tener los ficheros organizados de una manera determinada, siendo la unidad mínima los paquetes. Estos son una colección de archivos, tanto ejecutables como de soporte, que sirven un propósito específico y pueden funcionar por sí mismos, pero también ser agrupados dentro de un *stack*, que sólo es un grupo de paquetes (Figura 2.11)

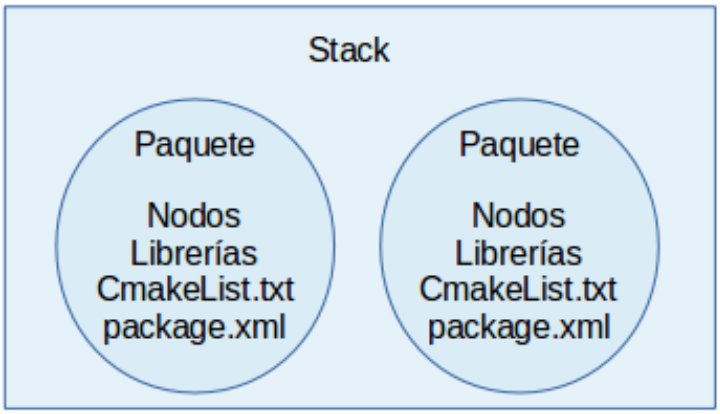


Figura 2.11: Sistema de ficheros de ROS.

Existen multitud de repositorios en los que se pueden encontrar paquetes y *stacks* desarrollados por otros usuarios. Uno de los repositorios más conocidos es GitHub [24], de donde se han extraído todos los paquetes principales para trabajar con este proyecto [8], desarrollados previamente por el grupo de investigación del Departamento de Ingeniería en Sistemas y Automática.

2.2.3. Gazebo

Los simuladores son una herramienta fundamental en Robótica. Además de permitir el acceso a usuarios que no disponen de medios económicos para adquirir el sistema robótico real, facilitan el proceso de aprendizaje y el trabajo en grupo cuando el robot físico es único. Permiten lanzar varias simulaciones al mismo tiempo en entornos muy diversos y controlados, sin dañar los motores o articulaciones del robot.

Los motores y articulaciones de un robot realizan mucho esfuerzo al ejecutar un algoritmo de aprendizaje, sobre todo en el tipo de aprendizaje utilizado en este proyecto, ya que el AR se basa en el método de ensayo-error. Además, sigue siendo necesario un espacio de trabajo y la carga de la batería.

Con un simulador, podremos probar estos algoritmos sin que el robot realice un esfuerzo mecánico y sin la necesidad de tenerlo físicamente hasta que se hayan corregido todos los errores. Por ello, ROS dispone de multitud de simuladores gratuitos 2D y 3D. Sin embargo, en nuestro proyecto vamos a trabajar con Gazebo. [25]

Este simulador 3D permite realizar aplicaciones de navegación y manipulación, incluyendo incluso variables físicas que proporcionan más realismo a la simulación.

La integración con ROS es plena mediante el paquete *gazebo_ros* y utiliza varios *plugins* para comunicaciones bidireccionales entre el simulador y ROS. Así, se pueden ejecutar los programas de forma idéntica en el robot real y el simulado.

En él se pueden incluir diferentes modelos 3D e incluso varios robots. La Figura 2.12 muestra un ejemplo de simulación en Gazebo.

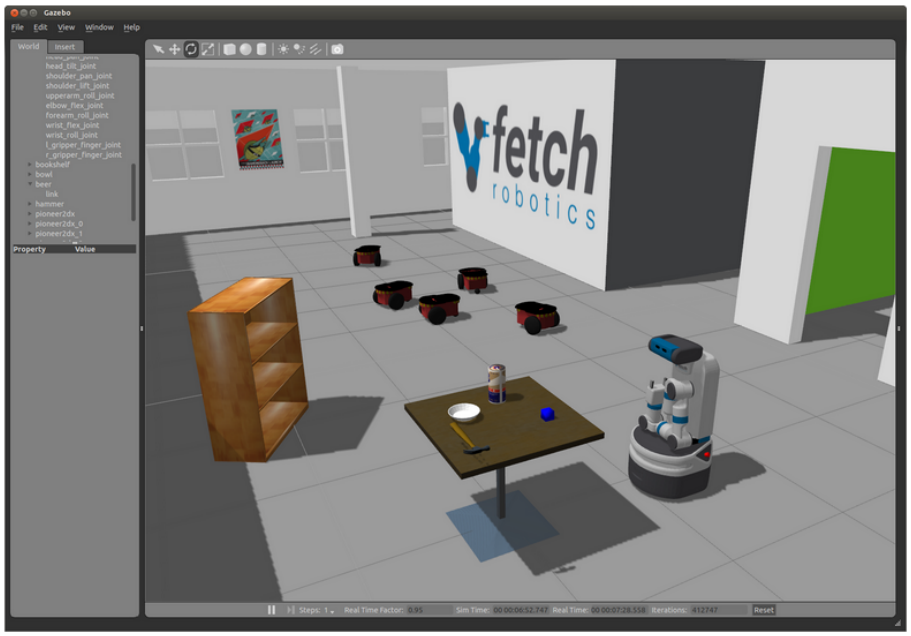


Figura 2.12: Ejemplo de simulación en Gazebo [26].

Se planteó la posibilidad de trabajar con el simulador *Virtual Robot Experimentation Platform* (V-REP) [27], ya que presenta las mismas funcionalidades que Gazebo, además de integración directa con mayor número de sistemas operativos. Sin embargo, para su uso se precisaba de un lenguaje de programación diferente a C++ con ROS y resultaría complicada su integración con programas orientados a procesos necesarios para la tarea a realizar, como visión por computador, además de un retraso debido al aprendizaje de este nuevo lenguaje. Por ello es mencionado en diversos capítulos del TFG, aunque no es utilizado posteriormente en él, ya que se ha realizado un esfuerzo en intentar incluirlo en el proceso de aprendizaje.

2.3. *Open Source Computer Vision Library*

Open Source Computer Vision Library (OpenCV) [12] es una librería de código abierto que incluye varios cientos de algoritmos de visión por computador, cuya versión utilizada en este proyecto es la API de C++ OpenCV 2.x. Esta librería tiene una estructura modular, que se divide en los siguientes módulos [28]:

- **core:** un módulo compacto que define las estructuras de datos básicas usadas por todos los demás módulos.
- **imgproc:** módulo de procesamiento de imágenes que incluye filtros lineales y no lineales, transformaciones geométricas, conversiones del espacio de color, histogramas, etc.
- **video:** módulo de análisis de vídeo.
- **calib3d:** módulo de reconstrucción 3D y calibración de la cámara.
- **objdetect:** detectores de características, descriptores y enlazadores de descriptores.
- **highgui:** detección de objetos e instancias de clases predefinidas.
- **videoio:** interfaz de fácil uso para capacidades de interfaz de usuario.
- Otros módulos suplementarios.

Los módulos utilizados en nuestra implementación han sido fundamentalmente *core*, *imgproc* y *highgui*.

Capítulo 3

Aprendizaje de una tarea: alcance de objeto

3.1. Comprobaciones previas

Antes de empezar con el trabajo, fue necesario comprobar qué combinación de sistema operativo, ROS y simuladores permitía controlar el robot CRUMB con el ordenador portátil del que se disponía.

3.1.1. Windows 8.1

Empezamos comprobando su funcionamiento en Windows 8.1, ya que era la distribución principal del ordenador en el que se trabajaría, investigando su compatibilidad con los simuladores que se podrían utilizar. Al examinar esta con el entorno de simulación robótica V-REP, se verificó que V-REP tiene versión para Windows [29].

Para manejar CRUMB necesitamos usar el *plugin* incluido en V-REP *RosInterfaces* [30], precisando instalar ROS en el equipo. Sin embargo, como podemos apreciar en la Figura 3.1, Windows 8.1 no es contemplado en la última versión de ROS, Melodic [31].

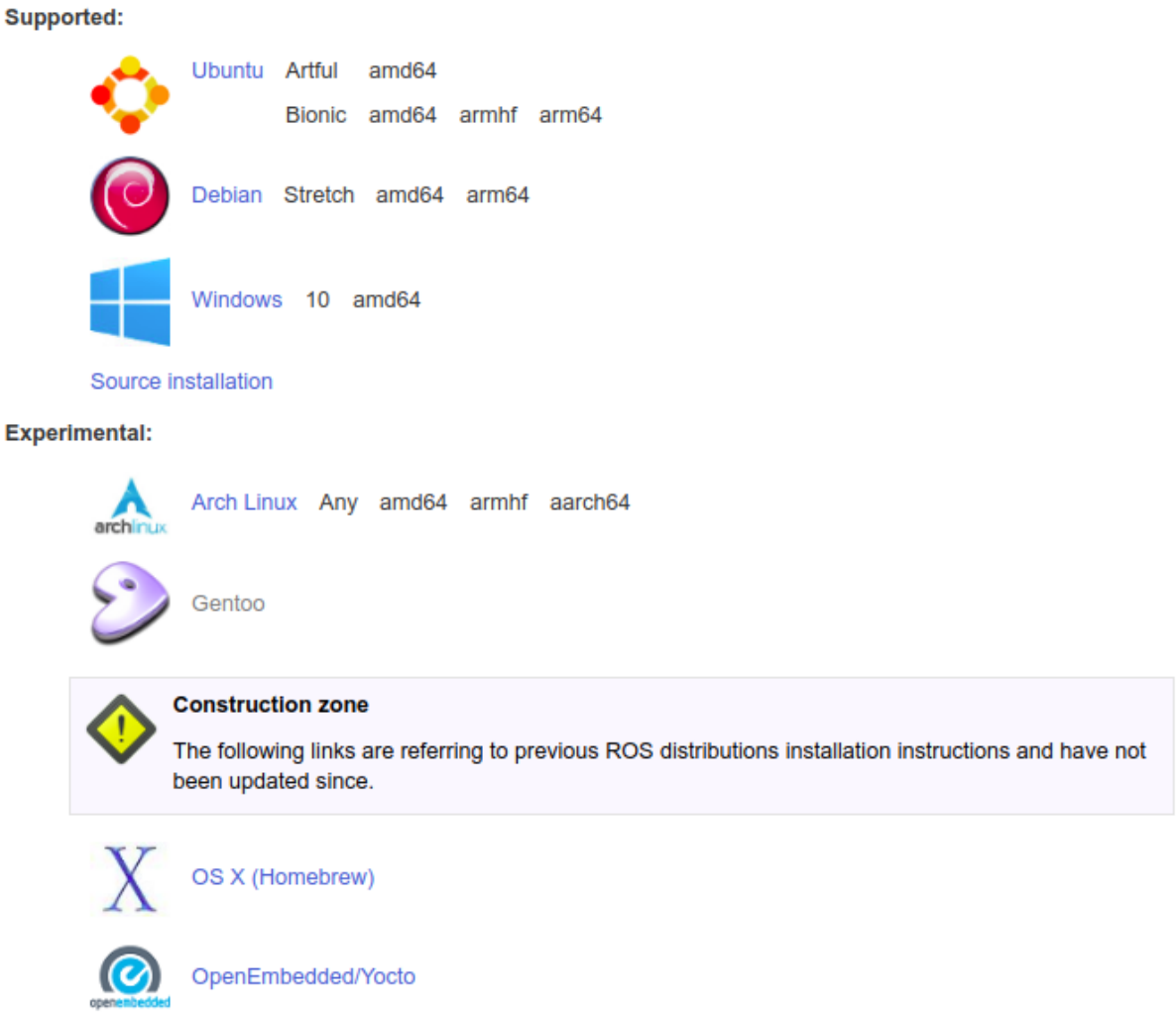


Figura 3.1: Plataformas soportadas por ROS Melodic [31].

Debido a esto, se tuvo que probar con otra partición disponible en el portátil, con Ubuntu 17.04 *Long Term Support* (LTS) instalado.

3.1.2. Ubuntu 17.04 LTS

En esta partición se pudieron instalar los simuladores Gazebo y V-REP sin problemas, además de ROS Melodic, la última versión disponible de este *framework* de desarrollo [31].

Sin embargo, CRUMB y en especial *Turtlebot 2*, están configurados para funcionar con ROS Indigo. Es decir, todos sus paquetes están programados para funcionar con esta versión de ROS.

Por lo tanto, se procedió a comprobar la compatibilidad entre estas dos versiones de ROS, buscando versiones de esos paquetes compatibles con ROS Melodic, instalando paquetes auxiliares e intentando realizar pequeñas modificaciones en los paquetes para crear dichas compatibilidades. Sin embargo, la versión de C++ usada por ROS Indigo es C++03, mientras que ROS Melodic utiliza C++14 [32]. De este modo las diferencias eran tales que, para hacer los paquetes completamente funcionales, habría que crearlos desde cero, suponiendo un esfuerzo fuera del ámbito del TFG.

3.1.3. Ubuntu 14.04 LTS

Finalmente, se comprobó que sí era posible trabajar en una partición de Ubuntu 14.04 LTS con ROS Indigo. Para ello seguimos el manual de instalación del Trabajo de Fin de Máster (TFM) de Marina Aguilar Moreno [18]; algunos problemas se han recogido en el Manual de instalación presente en el Apéndice A.

Cabe destacar el problema que tuvimos con el brazo manipulador Widow-X (Sección A.5), para el cual tuvimos que configurar los actuadores de cada articulación independientemente, ya que solamente detectaba dos de los 6 actuadores.

Para ello tuvimos que ir desconectando todos los actuadores y conectándolos uno a uno a la placa *Arduino* de la base del brazo. A continuación, se tuvo que ejecutar un código *Arduino* proporcionado por el fabricante (*Trossen Robotics*) [19], junto con el programa de configuración de los motores DynaManager, para configurarlos manualmente.

Además, también tuvimos problemas con algunos paquetes ROS (Sección A.3), el uso del láser (Sección A.6) y el uso de V-REP, ya que debíamos instalar una versión más actualizada de *cmake* (Sección A.2).

Finalmente no se usó V-REP por cuestiones de tiempo, ya que la incorporación de ROS a V-REP requería crear código usando el *plugin RosInterface* [30], bastante diferente a código C++ con ROS.

3.2. Implementación sin aprendizaje

Antes de comenzar la implementación del algoritmo de aprendizaje para alcanzar un objeto con el robot CRUMB, fue necesario comprobar si las actividades involucradas eran posibles para el robot. Dichas tareas son la detección del objeto y el movimiento del brazo manipulador para el alcance del objeto. De este modo, si dichas acciones no eran posibles, no tendría sentido el aprendizaje, y debería haberse cambiado el objetivo. Para ello se realizó un nodo ROS ejecutable en simulación de forma que si el robot estaba enfrente del objeto, este se aproximaba y lo cogía.

Antes de hablar de las comprobaciones realizadas con dichas funciones, tenemos que hablar de las diferencias existentes entre el robot real y el simulado y por qué hemos decidido trabajar con uno y no con otro.

Para comenzar, el robot real produce más *topics* que el robot simulado, por lo que se tuvo que hacer un estudio de los que eran generados por ambos entornos, para que el código pueda ser adaptado a ambos sin gran esfuerzo. Además, no todos los *topics* devuelven los mismos datos, siendo destacable el que nos devuelve el estado de las articulaciones del brazo, el cual en el robot real nos devuelve la posición angular de la articulación o la apertura de la pinza, mientras que en la simulación no.

Finalmente, se decidió utilizar solamente la simulación, ya que es un entorno más controlado en términos de iluminación o reflejos, por ejemplo, lo que nos facilita la detección del objeto, y también porque el robot no tiene que realizar esfuerzos mecánicos que puedan dañar sus componentes.

A continuación se presentan los principales problemas que se tuvieron que solventar antes de la implementación del aprendizaje.

3.2.1. Detección del objeto

Nuestra primera tarea a realizar era la detección objetos utilizando la información proporcionada por la cámara *Kinect*. Los problemas principales de nuestro TFG en este sentido se debieron a la poca información encontrada de la cámara *Kinect* [14] y a algunos problemas extra debidos a la naturaleza del entorno.

Para simplificar el problema, decidimos utilizar un único objeto de color rojo, ya que destacaría en el entorno de pruebas del laboratorio en el que se ha trabajado. Para detectarlo, se creó un filtro para hallar objetos rojos en la imagen, a partir del cual se obtenía la posición del punto central de este. Para ello tuvimos que transformar la imagen al modelo de color Matiz, Saturación, Valor (HSV) (Figura 3.2), ya que la librería OpenCV necesita de la imagen en este modelo de color.

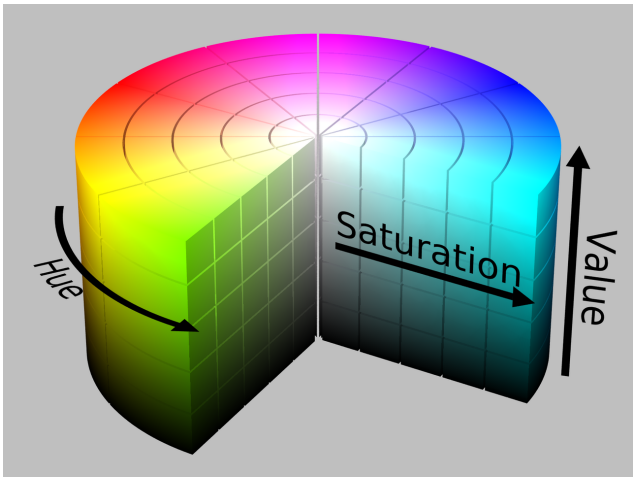


Figura 3.2: Modelo de color HSV. [33]

La detección de un objeto de un color determinado con la librería OpenCV, se realiza a partir de unos filtros de color en el modelo HSV. En este modelo, el color rojo en toda su gama se encuentra entre los valores de matiz 350 y 10, por lo que harían falta dos filtros para su correcta detección. Debido a esto, a partir de la imagen en modelo HSV obtenida, tuvimos que invertir su mapa de color (Figura 3.3) de forma que solo debemos usar un filtro para detectar el objeto.

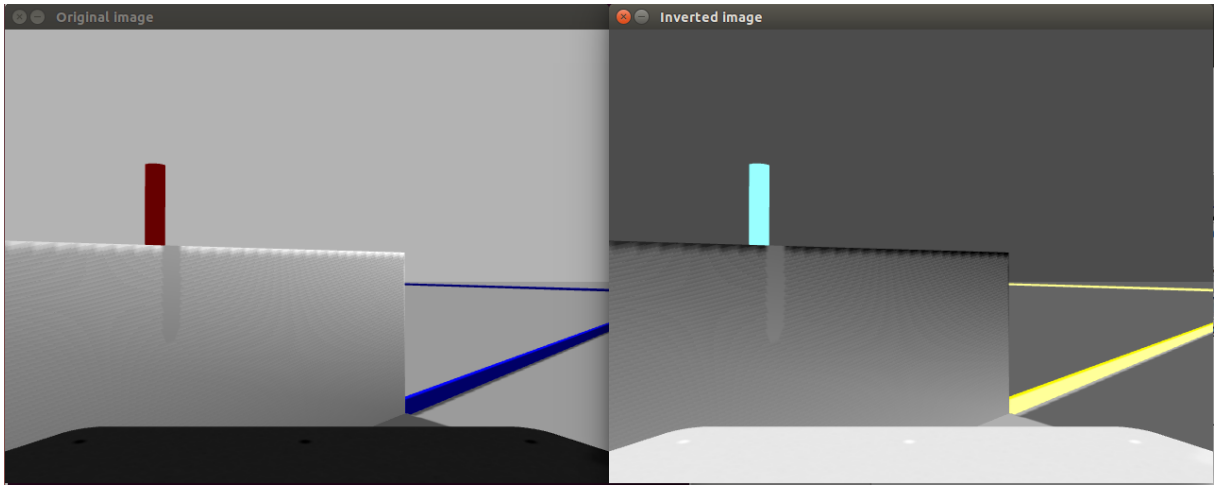


Figura 3.3: Imagen real - Imagen invertida.

Una vez obtenida la imagen con su mapa de color invertido, aplicamos un detector de color cian mediante filtro utilizando la librería OpenCV, quedando como se muestra en la Figura 3.4:

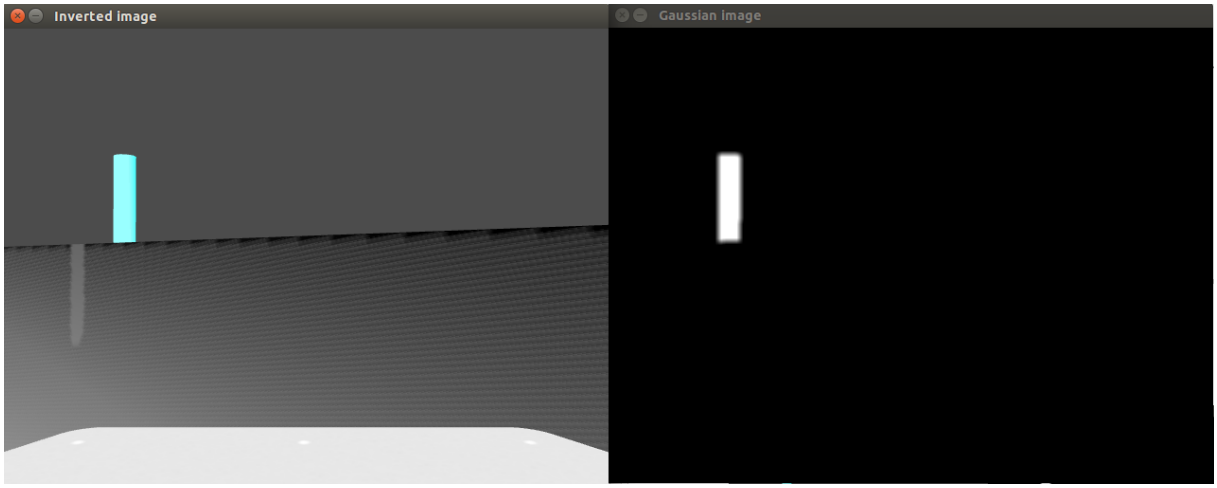


Figura 3.4: Objeto detectado.

Como se puede apreciar en las Figuras 3.3 y 3.4, la cámara *Kinect* se encuentra a menor altura que el objeto, ya que se posiciona en la base de la plataforma móvil del robot. Esto influye en la detección del objeto ya que, cuando el robot está muy cerca de este, la mesa oculta parte del objeto a la cámara, proporcionando unas medidas en altura del objeto erróneas.

Aquí aparece el mayor problema de la detección del objeto: la distancia a la que se encuentra del robot. Aparte de la cámara *Kinect* [14], el robot dispone de un láser *hokuyo* para hallar la profundidad a la que se encuentra el objeto, como se ha comentado anteriormente en la sección 2.1.

Comenzamos probando con el láser [15], pero este solamente nos proporciona la profundidad en un plano y el objeto puede estar a diferentes alturas, por lo que no nos sirve en la mayoría de los casos.

Por tanto, utilizar la imagen de profundidad de la cámara *Kinect* [34] parecía la opción correcta, ya que en la simulación nos proporciona la profundidad prácticamente sin errores. Sin embargo, al tener diferente resolución que la cámara RGB [35] (1024V * 1280H frente a 480V * 640H píxeles) resultaba complicado localizar el objeto en la imagen de profundidad.

Además queremos que el TFG sea adaptable fácilmente a un entorno real y en este, debido a los reflejos provocados por los objetos del entorno, la cámara no nos proporciona una profundidad fiable. Esto se muestra en la Figura 3.5, donde se aprecia cómo los reflejos son detectados como objetos rojos (parte derecha de la Figura), provocando errores en la localización del objeto real. Además, no existe la posibilidad de aplicarle filtros a la imagen de profundidad que la suavicen y eliminen dichos reflejos.

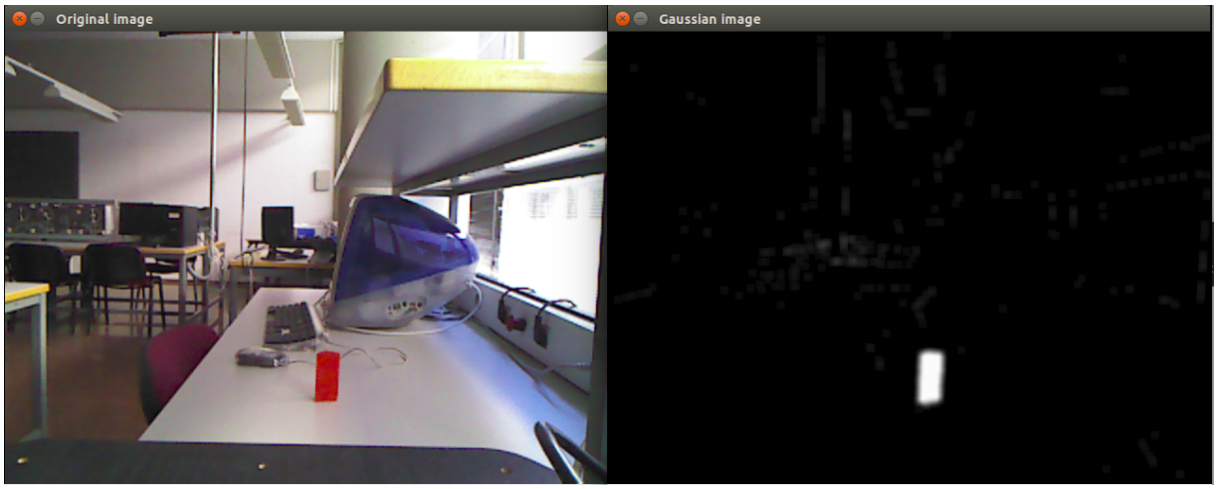


Figura 3.5: Reflejos en un entorno real.

Así pues, tenemos que utilizar la imagen RGB proporcionada por la cámara y, mediante un sistema de triangulación, calcular la distancia a la que se encuentra el objeto. Para ello se obtiene el punto más a la izquierda y más a la derecha del objeto (debido a que si cogemos la altura, cuando el robot estuviera demasiado cerca, la cámara quedaría debajo del objeto, cortando a este en alto pero no en ancho, debido a la posición de la cámara *Kinect* en la base de la plataforma móvil, como se ha explicado anteriormente). Para evitar que el ancho del objeto sea erróneo (porque no aparezca en pantalla completamente) creamos unas fronteras de seguridad de 42 píxeles de ancho donde, si se encuentra algún píxel del objeto, sería como si el robot no detectara el objeto.

Se calcula ese ancho, transformándolo de píxeles a metros mediante una matriz de transformación (Matriz 3.1) y la matriz de proyección de la cámara (Matriz 3.3) y realizando una triangulación con el ancho real del objeto y la distancia focal de la cámara, obteniendo la distancia real hasta la cámara del robot, en su sistema de referencia (Ecuación 3.6). A continuación se haya el punto central del objeto (en coordenadas homogéneas) en el sistema de referencia del sensor y se multiplica por esa distancia.

La matriz de transformación utilizada transforma un punto en píxeles del sistema de coordenadas de la imagen a la del plano de proyección y es:

$$T_P^I = \begin{bmatrix} 1 & 0 & 0 \\ 0 & -1 & 481 \\ 0 & 0 & 1 \end{bmatrix} \tag{3.1}$$

Además, también se utiliza la matriz de proyección de la imagen RGB de la cámara de tamaño 3x4:

$$P = \begin{bmatrix} fx & 0 & cx & Tx \\ 0 & fy & cy & Ty \\ 0 & 0 & 1 & 0 \end{bmatrix} \tag{3.2}$$

Siendo:

(fx, fy): distancia focal

(cx, cy): centro óptico de la cámara

(Tx, Ty): desplazamiento de la imagen con respecto al mundo real

Quedando como sigue:

$$P = \begin{bmatrix} 554{,}25 & 0 & 320{,}5 & 0 \\ 0 & 554{,}25 & 240{,}5 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} \tag{3.3}$$

En las Ecuaciones 3.4 y 3.5, $pixel\_pos = \begin{bmatrix} xo & yo & 1 \end{bmatrix}^t$ nos indica la posición en píxeles del objeto, siendo (xo, yo) un punto del objeto en píxeles (en nuestro caso los puntos laterales del objeto o el centro de este) y el vector *e* muestra la posición en píxeles del objeto en el plano proyección de la cámara (el cual es de 3 dimensiones y, por tanto, de 4 en coordenadas homogéneas). El acceso a los índices de la matriz lo denotamos con P[x,y] o e[z], denotando al índice de la fila "x" y columna "y" de P y al valor número "z" de e (tomando los índices de 0 al número de elementos del vector fila o columna menos 1).

El valor de la posición en píxeles del objeto en metros desde el sistema de referencia del sensor ($meter\_pos$) se calcula de la siguiente forma:

$$e = T_P^I * pixel\_pos \tag{3.4}$$

$$meter\_pos = \begin{bmatrix} \frac{(e[0]-P[0,2]*e[2]+P[2,3]*P[0,2])}{P[0,0]} \\ \frac{(e[1]-P[1,2]*e[2]+P[2,3]*P[1,2])}{P[1,1]} \\ e[2] - P[2,3] \\ 0 \end{bmatrix} \tag{3.5}$$

Para hallar el valor de los píxeles más a la izquierda y a la derecha del objeto, este $meter\_pos$ se multiplica por la distancia focal (f) y para hallar el valor del centro del objeto, dividimos $meter\_pos$ por la norma y después multiplicamos por la distancia hallada por triangulación de la siguiente manera:

$$distancia = ((f * OBJECT\_WIDTH)/width)/1000 \tag{3.6}$$

donde $OBJECT\_WIDTH$ es el ancho real del objeto (en mm), conocido previamente, y $width$ es el ancho del objeto en la imagen, ya transformado a metros y en el sistema de referencia del sensor. Se divide por 1000 para que la distancia esté en metros.

Después de realizar esto nos dimos cuenta de que el problema de los reflejos seguía estando presente. Así, para solucionarlo completamente decidimos aplicarle un par de filtros Gaussianos para suavizar la imagen, de forma que el primer filtro reduce los falsos positivos proporcionados por el error intrínseco de la cámara y el segundo, los reflejos detectados en el entorno.

Debido a estos filtros y a la baja calidad de la cámara (480 * 640 píxeles) [35], se produce un pequeño error en la lectura de dicha distancia, la cual se acentúa cuando esta aumenta. Así pues, tuvimos que hacer comprobaciones de hasta qué distancia el error no era considerable, llegando a la conclusión de que, para que el robot aprendiera bien, el objeto debería de estar a un máximo de 1 metro del robot, dándonos un error máximo de 5 cm y poca oscilación en el mismo, como podemos apreciar en la Figura 3.6.

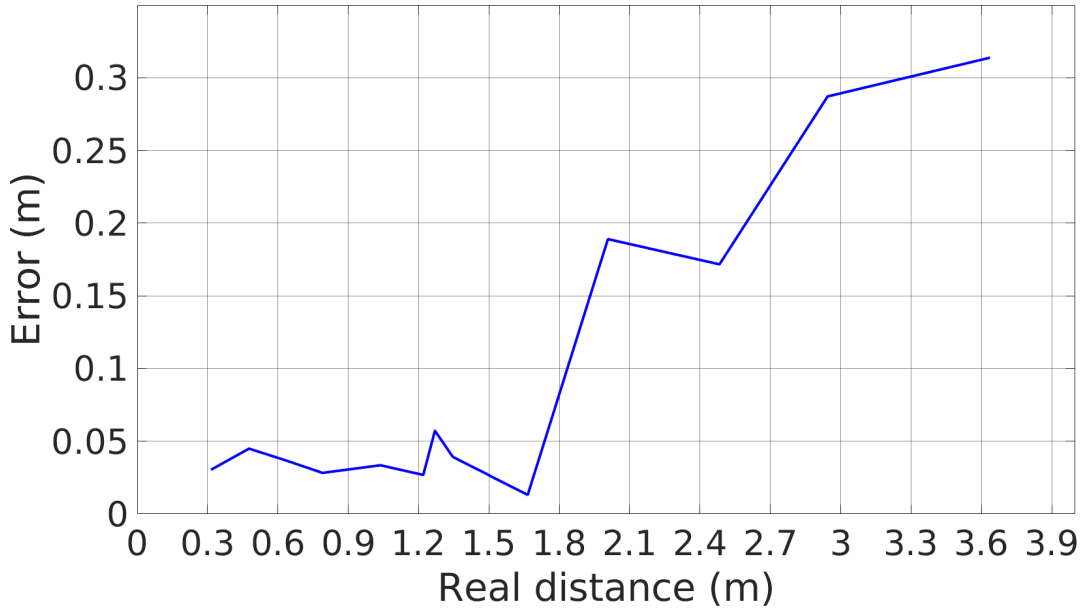


Figura 3.6: Evolución del error según distancia al objeto.

Por último, comenzamos los experimentos con objetos prismáticos de cuatro caras. Esto nos acarreaba otro problema: si el robot ve el objeto por su esquina (con lo cual es posible que aprecie el objeto más ancho de lo que realmente es), al estar calculando la profundidad mediante el ancho que se aprecia de este, nos puede proporcionar una distancia errónea. Debido a esto, utilizamos un objeto cilíndrico.

En el Anexo B se muestra el código para localizar el objeto y calcular su distancia.

3.2.2. Movimiento del brazo Widow-X

Para el correcto movimiento del brazo, tuvimos que realizar un estudio de los posibles movimientos que podía realizar el mismo sin producir una singularidad, que es una posición en el espacio donde el brazo manipulador no llega o tendría que moverse a una velocidad infinita para alcanzarla, por lo que se podría dañar. De esta forma comprobamos, como se especifica en las características técnicas del brazo [20], que la distancia horizontal máxima de este es de 41 cm, mientras que en vertical es de 33 cm hacia abajo y de 30 cm hacia arriba aproximadamente (con la pinza en horizontal).

Además, tuvimos que trabajar con el Modelo Cinemático Directo (MCD) y el Modelo Cinemático Inverso (MCI), fuente del TFM de Marina Aguilar Moreno [18], así como matrices de transformación para transformar la posición del objeto del sistema de referencia del sensor al sistema de referencia de la base del brazo manipulador.

El MCD se encarga de, dados los ángulos de cada una de las articulaciones del brazo, darnos la *pose* de la pinza (la cual se compone de su posición y ángulo). El MCI realiza justamente el proceso contrario: dada la *pose* de la pinza y la matriz de transformación de la base del brazo a la pinza (llamada también matriz de paso), nos da el ángulo de las articulaciones.

3.2.2.1. Modelo Cinemático Directo

El MCD se ha utilizado en el TFG porque el MCI utiliza este modelo para calcular el ángulo de giro de las articulaciones para mover la pinza a la posición deseada (en nuestro caso, al objeto). El código en C++, adaptado del de Matlab proporcionado en el TFM de Marina Aguilar Moreno [18], se muestra en el Apéndice B.

Para hallar el MCD primero se precisa calcular los parámetros geométricos de Denavit-Hartenberg. Estos parámetros se dividen en cuatro [36]:

- a_i : Distancia desde el eje Z_i hasta el eje Z_{i+1} medida a través de X_i.
- α_i : Ángulo que forman los ejes Z_i y Z_{i+1} medido sobre X_i.
- d_i : Distancia desde el eje X_{i-1} hasta el eje X_i medida a través de Z_i.
- θ_i : Ángulo que forman los ejes X_{i-1} y X_i medido sobre Z_i. Estas son las variables articulares de nuestro brazo manipulador, es decir, las variables dinámicas, debido a que todas las articulaciones son de revolución.

Para obtener dichos parámetros, primero hay que definir los sistemas de coordenadas de cada articulación de la siguiente forma:

1. Los ejes Z se colocan según la dirección y sentido de giro de las articulaciones.

2. El centro de coordenadas de cada articulación se sitúa en la intersección de la perpendicular común entre los ejes.

3. El eje X_i se sitúa formando una perpendicular común o punto de corte entre los ejes Z_i y Z_{i+1}.

4. El sistema de coordenadas 0 se sitúa coincidente con el 1 cuando no esté rotado.

5. La última articulación se sitúa coincidente con la 4, alineada con ella cuando θ sea 0, colocando Z_5 en función del giro de la articulación.

6. Los ejes Y se posicionan según la regla de la mano derecha.

Además, hay que añadir una serie de dimensiones. El resultado se muestra en la figura 3.7:

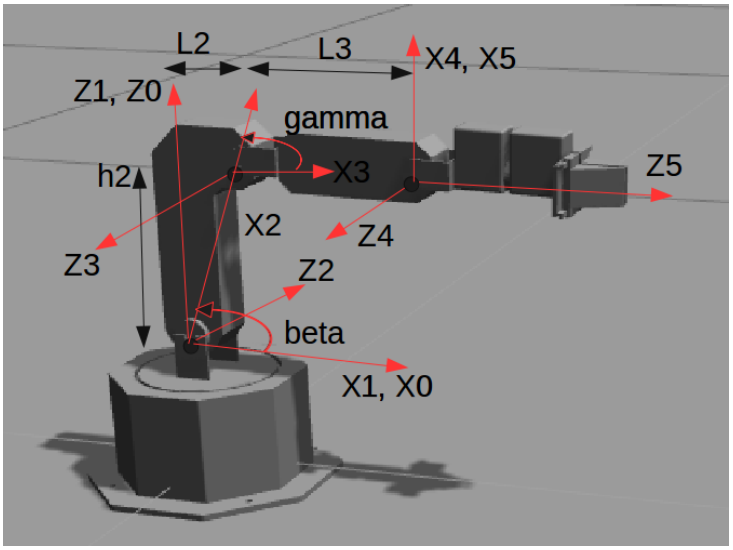


Figura 3.7: Ejes y dimensiones para el cálculo de los parámetros Denavit-Hartenberg. [18]

Estas dimensiones son:

$$h_2 = 0{,}1450m \tag{3.7}$$

$$L_2 = 0{,}04825m \tag{3.8}$$

$$L_3 = 0{,}1423m \tag{3.9}$$

$$\beta = Atan(\frac{h_2}{L_2}) = 1{,}2496rad \tag{3.10}$$

$$\gamma = \beta = Atan(\frac{h_2}{L_2}) = 1{,}2496rad \tag{3.11}$$

$$d_2 = \sqrt{L_2^2 + h_2^2} = 0{,}1528m \tag{3.12}$$

En la Tabla 3.1 se muestran los parámetros Denavit-Hartenberg del brazo WidowX.

Articulaciones	a_{i-1}	α_{i-1}	d_i	θ_i
1	0	0	0	θ_1
2	0	$-\frac{\pi}{2}$	0	$\theta_2 - \beta$
3	d_2	π	0	$\theta_3 - \gamma$
4	L_3	0	0	$\theta_4 + \frac{\pi}{2}$
5	0	$\frac{\pi}{2}$	0	θ_5

Tabla 3.1: Parámetros Denavit-Hartenberg del manipulador WidowX. [18]

Una vez obtenidos los parámetros, se puede calcular el MCD, que queda como sigue:

$$T_5^0 = \begin{bmatrix} r_{11} & r_{12} & r_{13} & p_x \\ r_{21} & r_{22} & r_{23} & p_y \\ r_{31} & r_{32} & r_{33} & p_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{3.13}$$

donde:

$$r_{11} = \sin\theta_1 \sin\theta_5 - \sin(\theta_3 - \theta_2 + \theta_4)\cos\theta_1 \cos\theta_5 \tag{3.13a}$$

$$r_{12} = \cos\theta_5 \sin\theta_1 + \sin(\theta_3 - \theta_2 + \theta_4)\cos\theta_1 \sin\theta_5 \tag{3.13b}$$

$$r_{13} = \frac{\cos(\theta_1 + \theta_2 - \theta_3 - \theta_4)}{2} + \frac{\cos(\theta_1 + \theta_2 + \theta_3 + \theta_4)}{2} \tag{3.13c}$$

$$r_{21} = -\cos\theta_1 \sin\theta_5 - \sin(\theta_3 - \theta_2 + \theta_4)\cos\theta_5 \sin\theta_1 \tag{3.13d}$$

$$r_{22} = \sin(\theta_3 - \theta_2 + \theta_4)\sin\theta_1 \sin\theta_5 - \cos\theta_1 \cos\theta_5 \tag{3.13e}$$

$$r_{23} = \frac{\sin(\theta_1 + \theta_2 - \theta_3 - \theta_4)}{2} + \frac{\sin(\theta_1 + \theta_2 + \theta_3 + \theta_4)}{2} \tag{3.13f}$$

$$r_{31} = \frac{\cos(\theta_2 - \theta_3 - \theta_4 + \theta_5)}{2} + \frac{\cos(\theta_3 - \theta_2 + \theta_4 + \theta_5)}{2} \tag{3.13g}$$

$$r_{32} = -\frac{\sin(\theta_2 - \theta_3 - \theta_4 + \theta_5)}{2} - \frac{\sin(\theta_3 - \theta_2 + \theta_4 + \theta_5)}{2} \tag{3.13h}$$

$$r_{33} = \sin(\theta_3 - \theta_2 + \theta_4) \tag{3.13i}$$

$$p_x = \cos\theta_1(L_3\cos(\theta - \theta_3) + d_2 * \cos(\theta_2 - \beta)) \tag{3.13j}$$

$$p_y = \sin\theta_1(L_3\cos(\theta - \theta_3) + d_2 * \cos(\theta_2 - \beta)) \tag{3.13k}$$

$$p_z = -d_2\sin(\theta_2 - \beta) - L_3\sin(\theta_2 - \theta_3) \tag{3.13l}$$

3.2.2.2. Modelo Cinemático Inverso

El método utilizado por Marina Aguilar Moreno en su TFM [18] para hallar el MCI es el modelo algebraico, que utiliza el MCD calculado en el apartado anterior para hallar el ángulo de giro de cada articulación. Este modelo se ha usado en el TFG para, una vez localizado el objeto en el espacio, mover el brazo hasta dicha posición. Para ello se ha adaptado el código Matlab presente en el TFM de Marina Aguilar Moreno [18] a código de C++, mostrado en el Apéndice B.

Así, siguiendo el procedimiento de dicho TFM [18], conseguimos las ecuaciones del MCI:

$$\theta_1 = Atan(\frac{p_y}{p_x}) \tag{3.14}$$

$$\theta_2 = \beta + Atan(\frac{k_1}{k_2}) - Atan(\frac{k}{\pm\sqrt{k_1^2 + k_2^2 + k^2}}) \tag{3.15}$$

donde

$$k = p_z^2 + d_2^2 + (p_x \cos\theta_1 + p_y \sin\theta_1)^2 - L_3^2 \tag{3.15a}$$

$$k_1 = 2p_x \cos\theta_1 d_2 + 2p_y \sin\theta_1 d_2 \tag{3.15b}$$

$$k_2 = 2p_z d_2 \tag{3.15c}$$

$$\theta_3 = \theta_2 - Asin(\frac{-p_z - d_2 \sin\beta}{L_3}) \tag{3.16}$$

$$\theta_4 = Acos(-k_3) - \frac{\pi}{2} \tag{3.17}$$

siendo

$$k_3 = a_z \cos(\theta_2 - \theta_3) + a_x \sin(\theta_2 - \theta_3)\cos\theta_1 + a_y \sin(\theta_2 - \theta_3)\sin\theta_1 \tag{3.17a}$$

$$\theta_5 = Asin(n_x \sin\theta_1 - n_y \cos\theta_1) \tag{3.18}$$

Los valores L_3, d_2 y β son dimensiones definidas en el apartado anterior, el vector columna $\begin{bmatrix} p_x & p_y & p_z \end{bmatrix}^t$, un punto genérico y los vectores columna $\begin{bmatrix} n_x & n_y & n_z \end{bmatrix}^t$ y $\begin{bmatrix} a_x & a_y & a \end{bmatrix}^t$ la rotación de la pinza en los ejes X y Z respectivamente. De esta forma, con las ecuaciones 3.14, 3.15, 3.16, 3.17 y 3.18 obtenemos el MCI del brazo manipulador Widow-X.

Una vez hallada la profundidad del objeto en el sistema de referencia del sensor y utilizando la matriz de transformación de este sistema al de la base del brazo (Ecuación 3.19), utilizamos el MCI para llevar la pinza hasta dicho objeto.

$$T_B^S = \begin{bmatrix} 0 & 0 & 1 & -0{,}0734 \\ -1 & 0 & 0 & -0{,}0125 \\ 0 & 1 & 0 & -0{,}2448 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{3.19}$$

3.3. Implementación con aprendizaje

Una vez solucionados todos los problemas planteados y comprobado que el robot es capaz de realizar la tarea destinada, era momento de empezar con la implementación del algoritmo de aprendizaje.

3.3.1. Estados y acciones

Como indicamos en el Capítulo 1.2, el algoritmo implementado (*Q-Learning*) hace uso de una matriz llamada *Q-Table* formada por los pares (estado, acción). Para empezar, tenemos que definir el conjunto de acciones y estados a utilizar. Así, las acciones posibles para el robot son girar la base hacia la izquierda, hacia la derecha, moverla hacia delante, hacia atrás o mover el brazo hacia el objeto.

En cuanto a los estados, están formados por valores discretos de la distancia, el ángulo y la altura al objeto, además de dos valores binarios que indican si se ha alcanzado el objeto y si el brazo manipulador está plegado o no. En las Tablas 3.2, 3.3 y 3.4 se indica el significado de cada valor de discretización para la distancia, el ángulo y la altura del objeto.

Valor de discretización	Significado
0	No se ve el objeto
1-9	El objeto está a una distancia de $\frac{Valor\ de\ discretización}{9}$ m del sensor
10	El objeto se encuentra a más de 1 m del sensor

Tabla 3.2: Significado de los valores de discretización para la distancia.

Valor de discretización	Significado
0	No se ve el objeto
1-9	El objeto está más a la izquierda o a la derecha del sensor

Tabla 3.3: Significado de los valores de discretización para el ángulo.

Valor de discretización	Significado
0	No se ve el objeto
1-9	El objeto está más arriba o abajo del sensor

Tabla 3.4: Significado de los valores de discretización para la altura.

Con estas definiciones se genera un total de 4400 estados distintos, con 5 acciones posibles, lo que nos da 22000 pares (estado, acción) generados. La inmensa cantidad de pares (estado, acción) que se generan para la *Q-Table* es uno de los problemas principales del algoritmo *Q-Learning* en Robótica, el cual ha tenido que ser estudiado con anterioridad a la definición de los estados y las acciones.

El nivel de discretización se ha escogido teniendo en cuenta que la distancia mínima a la que el robot puede estar del objeto (tocando la mesa) es de 22.3 cm aproximadamente (sin contar la distancia del sensor al centro del robot, que es de 8.7 cm). Además, la distancia máxima a la que el robot puede detectar la distancia correctamente es 1 m (con un error máximo de 5 cm) y el brazo desplegado completamente puede llegar hasta 41 cm en horizontal.

Para poder acceder a la matriz Q necesitamos transformar esos estados a índices, utilizando para ello la siguiente fórmula:

$$state\_idx = dist\_d * 10^2 * 2^2 + ang\_d * 10 * 2^2 + height\_d * 2^2 + object\_picked * 2 + folded\_arm \tag{3.20}$$

donde:

- **state_idx:** índice correspondiente al estado.
- **dist_d, ang_d, height_d:** valores discretos de la distancia, ángulo y altura respectivamente.
- **object_picked:** indicador de si se ha alcanzado el objeto.
- **folded_arm:** indicador de si el brazo está plegado.

Para obtener de nuevo el estado a partir del índice, se realiza la operación inversa (teniendo en cuenta que el símbolo % corresponde a la operación módulo y los símbolos $\lfloor$ y $\rfloor$, a la operación suelo):

$$dist\_d = \lfloor \frac{state\_idx}{10^2 * 2^2} \rfloor \tag{3.21}$$

$$ang\_d = \lfloor \frac{state\_idx \,\%(10^2 * 2^2)}{10 * 2^2} \rfloor \tag{3.22}$$

$$height\_d = \lfloor \frac{state\_idx \,\%(10^2 * 2^2) \,\%(10 * 2^2)}{2^2} \rfloor \tag{3.23}$$

$$object\_picked = \lfloor \frac{state\_idx \,\%(10^2 * 2^2) \,\%(10 * 2^2) \,\%2^2}{2} \rfloor \tag{3.24}$$

$$folded\_arm = state\_idx \,\%(10^2 * 2^2) \,\%(10 * 2^2) \,\%2^2 \,\%2 \tag{3.25}$$

Para mayor claridad, a continuación mostramos unos ejemplos del proceso para el estado (1,2,3,0,1):

1. Calculamos su valor en índice usando la Ecuación 3.20:

$$state\_idx = 1 * 10^2 * 2^2 + 2 * 10 * 2^2 + 3 * 2^2 + 0 * 2 + 1 = 493 \tag{3.26}$$

2. Hacemos el proceso inverso utilizando las Ecuaciones 3.21, 3.22, 3.23, 3.24 y 3.25:

$$dist\_d = \lfloor \frac{493}{10^2 * 2^2} \rfloor = \lfloor \frac{493}{400} \rfloor = 1 \tag{3.27}$$

$$ang\_d = \lfloor \frac{493 \,\%(10^2 * 2^2)}{10 * 2^2} \rfloor = \lfloor \frac{93}{40} \rfloor = 2 \tag{3.28}$$

$$height\_d = \lfloor \frac{493 \,\%(10^2 * 2^2) \,\%(10 * 2^2)}{2^2} \rfloor = \lfloor \frac{13}{4} \rfloor = 3 \tag{3.29}$$

$$object\_picked = \lfloor \frac{493 \,\%(10^2 * 2^2) \,\%(10 * 2^2) \,\%2^2}{2} \rfloor = \lfloor \frac{1}{2} \rfloor = 0 \tag{3.30}$$

$$folded\_arm = 493 \,\%(10^2 * 2^2) \,\%(10 * 2^2) \,\%2^2 \,\%2 = 1 \tag{3.31}$$

3.3.2. Elementos del aprendizaje

Después de decidir las acciones y los estados, tenemos que elegir el valor de los parámetros que afectan al aprendizaje. El propósito de estos parámetros se explicó en el apartado 1.2. Estos son los valores escogidos para cada uno de ellos:

- **Ratio de aprendizaje (α):** Lo hemos definido como $\frac{1}{visitas(estado,acción)^{0,5}}$, teniendo en cuenta que las visitas de una pareja (estado, acción) se definen como el número de veces que se ha realizado dicha acción en ese estado concreto. Se ha elegido este α por las siguientes razones:

  1. Al usar el número de visitas del par (estado, acción), el aprendizaje no se ve influenciado por el orden de las acciones tomadas en un estado determinado.
  2. Al utilizar un valor decreciente que converge a 0, cuando el robot haya realizado las suficientes iteraciones y haya aprendido lo suficiente, dejará de aprender. Esto es necesario porque, si no sucede, puede aprender incorrectamente lo que ya había aprendido correctamente.
  3. El número de visitas está elevado a 0.5 con el objetivo de que α no descienda demasiado rápido, pero tampoco demasiado lento, como se muestra en la Figura 3.8.

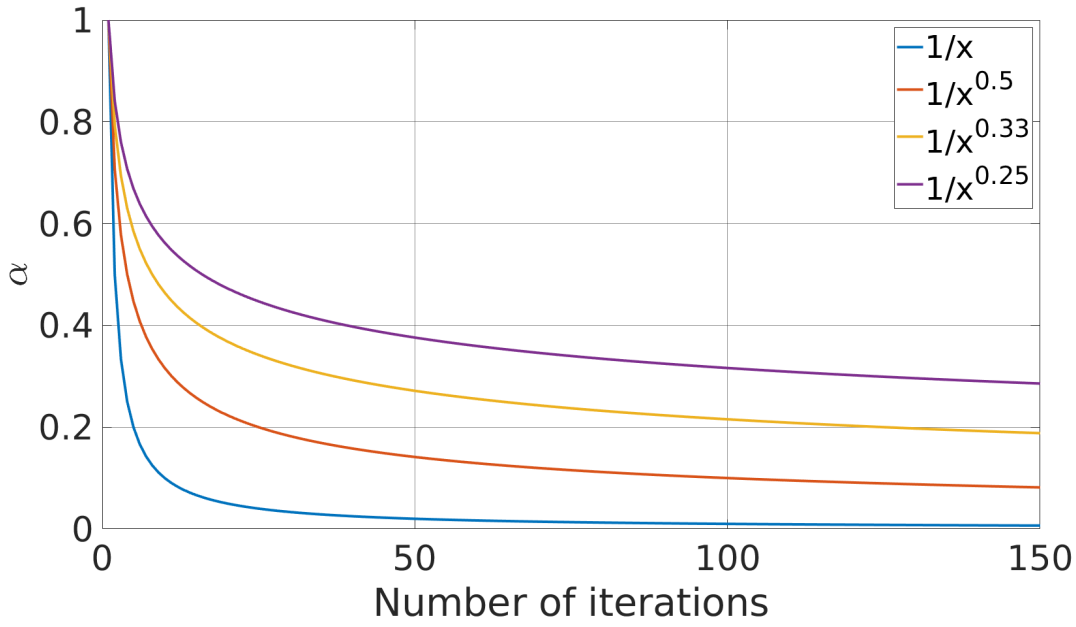


Figura 3.8: Evolución de α, donde X es el número de iteraciones.

- **Ratio de descuento (γ):** Para este valor hemos escogido un valor estático de 0.99, ya que queremos que tenga en cuenta el estado al que llega de manera notable. Aún así, como en la Ecuación 1.1 aparece multiplicado por α, al disminuir este último, γ también tiene menos efecto sobre el aprendizaje.
- **Estrategia de exploración/explotación:** Para ello hemos elegido una estrategia en el que el 30 % de las veces el robot realiza una acción aleatoria de las disponibles (explora). Sin embargo, el otro 70 % de las veces realiza la acción que tenga mayor valor en la *Q-Table* (explota).

  Si hay más de una acción que tiene la máxima puntuación para el estado actual, escogemos una de ellas al azar y, si hemos pasado a un estado en el que todavía no se ha explorado ninguna acción, exploramos.

Además de estos parámetros propios del *Q-Learning*, existen otros elementos destacables:

- El vector V, que incluye la calificación máxima en cada estado.
- El vector P, también llamado política del aprendizaje, que contiene la primera de las acciones con la calificación más alta en la *Q-Table*, es decir, la acción que ha aprendido el robot que es más favorable en cada estado.
- La matriz de visitas ($visitas(estado, acción)$) que representa el número de veces que se ha realizado dicha acción en ese estado concreto.

Cada experimento se realiza poniendo en marcha una simulación en Gazebo, la cual es un conjunto de varios episodios. Un episodio se define como un número de pasos de simulación (con un número máximo definido en cada experimento). Cuando el robot ha alcanzado el objeto o ha realizado el número de pasos máximos indicado en el experimento (en nuestro caso lo dejamos en 200 para el caso satisfactorio), se termina un episodio y se lanza uno nuevo. Todas las matrices utilizadas en el aprendizaje empiezan con sus valores inicializados a 0 al inicio del experimento. Para que el aprendizaje sea correcto, ninguna de dichas matrices se debe reiniciar después de cada episodio, o sería como si el aprendizaje empezara desde el principio en cada iteración.

Durante el experimento, se van rellenando unos archivos de *log* con la posición del robot en cada paso, del objeto en cada episodio, diferencias entre los vectores V de cada paso y episodio, y datos sobre el estado actual y anterior, acción realizada, recompensa y nuevos valores en las matrices del aprendizaje. Estos archivos sirven para un posterior análisis de los datos mediante código Matlab implementado, para poder visualizar el aprendizaje realizado por el robot y si este tiene sentido, como se muestra en la Sección 4.

3.3.3. Elección de las recompensas

Un elemento principal del AR son las recompensas. Sin ellas, el agente (en nuestro caso el robot CRUMB) no podría aprender, ya que no tendría ninguna referencia de si se está aproximando a su objetivo. Aún así, hay que tener un especial cuidado a la hora de definir las recompensas porque si se definen escasamente, el agente no es capaz de aprender correctamente, y si se definen abundantemente, no estaríamos aprovechando el potencial del AR, ya que estaríamos programando explícitamente las acciones a realizar.

De esta manera, se han probado distintas recompensas, algunas de las cuales se ejemplificarán en la Sección 4 destinada a los experimentos realizados y sus resultados. En la Tabla 3.5 se muestra el valor de todas las recompensas, junto con la acción que la desencadena y su representación en el estado, no siendo acumulables excepto la de "Alcanzar el objeto".

Acción	Representación en el estado	Recompensa
Alcanzar el objeto	El robot está en un estado donde la distancia discretizada es igual o menor a 3, el ángulo entre 3 y 5 y ha realizado la acción de mover el brazo	+100
Localizar el objeto	El robot ha cambiado de un estado donde los valores discretos de la distancia, ángulo y altura era 0 a uno en el que son mayores a 0	+3
Perder de vista el objeto	El robot ha cambiado de un estado donde los valores discretos de la distancia, ángulo y altura eran mayores a 0 a uno en el que todos son 0	-3
Acercarse al objeto	El valor de la distancia del estado anterior era mayor que el actual, siempre que le actual sea mayor que 0	+5
Alejarse del objeto	El valor de la distancia del estado anterior era menor que el actual, siempre que el anterior sea mayor que 0	-5
Centrar el objeto	El valor del ángulo del estado actual es más cercano a 5 que el del estado anterior	+(Ángulo discretizado en el estado anterior)

Descentrar el objeto	El valor del ángulo del estado actual es más lejano a 5 que el del estado anterior	-(Ángulo discretizado en el estado actual)

Tabla 3.5: Valor de las recompensas en el aprendizaje.

Se han probado los diferentes conjuntos de recompensas:

- **Recompensa positiva solamente por alcanzar el objeto:**

  Al aplicar solamente esta recompensa, el robot era totalmente incapaz de alcanzarlo debido a que, al no recibir recompensas por otra acción que no fuera esa, realizaba acciones aleatorias hasta alcanzar el objeto.

- **Recompensa positiva por localizar y alcanzar el objeto:**

  En este caso, logramos que el robot mantuviese localizado el objeto, pero no era capaz de acercarse a él, ya que no le dábamos ninguna pista de que tuviera que hacerlo.

- **Recompensa positiva por localizar, acercarse y alcanzar el objeto:**

  En el AR, siempre actualizamos las puntuaciones de una pareja (estado, acción) teniendo en cuenta el valor actual y el valor máximo del estado al que llegamos, por lo que puede ocurrir que el agente realice una acción a priori mala para llegar a un estado en el que pueda realizar una acción que le dé gran puntuación.

  En nuestro caso, al acercarse, si posteriormente se realizaba la acción de retroceder, aprendía que yendo en esa dirección después podría acercarse de nuevo para conseguir más recompensa, entrando en un bucle infinito.

- **Recompensa positiva en el caso anterior y negativa en modo espejo:**

  De esta forma, el robot es capaz de acercarse al objeto perfectamente, ya que tiene suficiente información. Sin embargo, el robot solamente puede alcanzar correctamente un objeto que esté centrado en su campo de visión y esto no se lo indicábamos, por lo que no llega a alcanzar el objeto correctamente con mucha frecuencia.

- **Todas las recompensas anteriores y recompensa según grado de centralización del objeto:**

  Para que el robot supiera que se tenía que aproximar al objeto con este centrado, se añade recompensa positiva si lo va centrando y negativa si lo va descentrando. Esto se realiza de una forma gradual, dándole más recompensa cuanto más descentrado estuviera, y quitándole más cuanto más lo descentre.

  De esta forma, el robot es capaz de aprender rápidamente a alcanzar el objeto, incluso en situaciones complejas, debido a que tiene suficiente información para saber cómo acercarse al objeto para alcanzarlo. Por lo tanto, nos quedamos con esta última recompensa.

Para entender mejor todo esto, incluimos el pseudocódigo del algoritmo *Q-Learning* en la Figura 3.9:

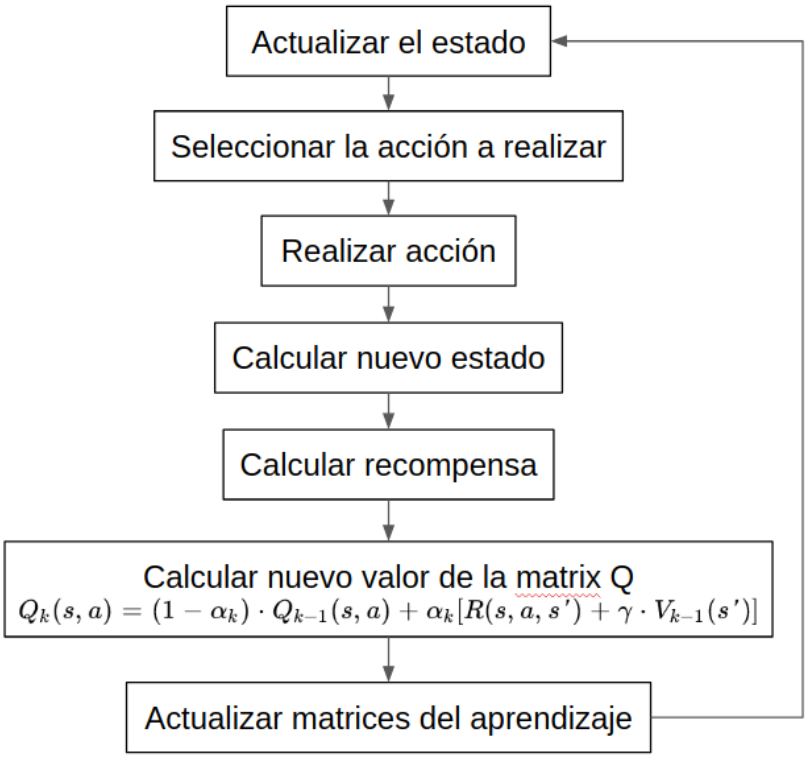


Figura 3.9: Pseudocódigo del algoritmo *Q-Learning*.

3.3.4. Entorno de simulación

Nuestro entorno de simulación consta de 4 paredes bajas, para acotar la zona de trabajo del robot y simular un entorno real (donde el espacio de trabajo no es infinito), una mesa con un objeto rojo encima y el robot CRUMB. En la Figura 3.10 se muestra dicho entorno.

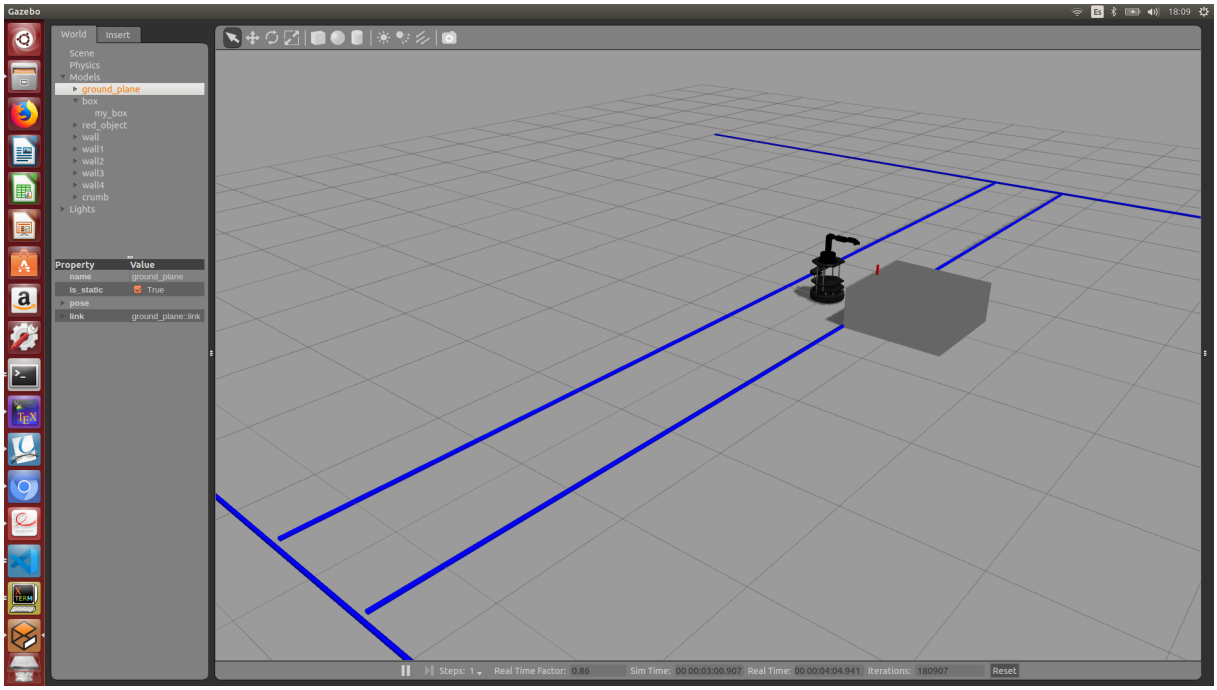


Figura 3.10: Entorno de simulación en Gazebo.

Cabe destacar que en simulación generamos las paredes bajas de forma automática y aleatoria mediante código, así como la mesa se posiciona entre unos rangos mostrados en la Tabla 3.6 y el objeto cilíndrico de 1.2 cm de radio y 10 cm de altura, según la posición $\begin{bmatrix} x & y & z \end{bmatrix}^t$ de la mesa, se coloca siguiendo las ecuaciones 3.32, 3.33 y 3.34.

Parámetros	Mínimo(m)	Máximo(m)
X (profundidad)	0.85	1.5
Y (derecha e izquierda)	-0.8	0.8
Z (posición en altura)	0.15	0.25
Altura	0.3	0.5

Tabla 3.6: Rangos de parámetros de la mesa.

$$X = x - 0{,}45 \tag{3.32}$$

$$Y = y \tag{3.33}$$

$$Z = 2 * z + 0{,}05 \tag{3.34}$$

Se ha optado por una situación aleatoria de la mesa y el objeto para que el robot, durante su aprendizaje, aprenda en todos los entornos posibles y no se limite a una sola situación. El código para la generación automática de simulaciones se muestra en el apéndice B.

Cabe destacar que el modelo simulado del robot CRUMB contiene un efector final hueco (la pinza del brazo manipulador solamente es maciza en su parte externa). Por ello, se tuvo que estudiar en qué posiciones alcanzaba correctamente el objeto para incluirlos como estados en los que lo conseguía, independientemente de los datos proporcionados por algunos nodos de interés para el robot real, siendo estos erróneos en simulación.

Además, hay que tener en cuenta que el robot no puede alcanzar el objeto a no ser que dicho objeto esté a menos de 33 cm del sensor y lo tenga centrado, además de que la mesa detiene el avance del robot a unos 22.3 cm de distancia del objeto (sin contar la distancia del sensor al centro de la base, que es de 8.7 cm).

Para configurar una simulación, en el archivo de C++ *learning.h* desarrollado, hay definidos varios elementos modificables, como tamaño real del objeto, posición y tamaño de la mesa o parámetros del aprendizaje como el ratio de exploración o γ. En el archivo *learning.cpp*, dentro del método *learning*, se encuentra la definición de α y del número máximo de pasos de un episodio.

Durante la simulación se rellenan automáticamente unos archivos (*logs*) con información muy diversa para su posterior estudio en Matlab. Antes de empezar la simulación es posible configurar el nombre de los archivos que almacenarán la información de *log*, así como si quieres sobrescribirlos, además de configurar otros parámetros como la ausencia de exploración o la visualización de la simulación en pantalla.

El análisis de datos posterior es de vital importancia en el campo de la Robótica debido a que, como se comentó en la Sección 1.2, en dicho campo todavía no se disponen de análisis científicos que establezcan, de manera experimental y general, las particularidades que hacen que el AR en Robótica sea exactamente igual que en otras disciplinas. Así, en la Sección 4, se muestra una serie de experimentos para demostrar qué ocurre conforme el robot va aprendiendo, y que la definición de las recompensas depende de la tarea a realizar, influyendo de manera notable en el resultado. Para la realización de este análisis se ha desarrollado código Matlab adjunto en el CD del TFG.

En el Anexo B se muestra el cuerpo de dicha implementación.

Capítulo 4

Experimentos y resultados

En este capítulo vamos a mostrar una serie de experimentos realizados, presentando diferentes situaciones y el porqué de su comportamiento. También vamos a demostrar que se necesita un cierto número de pasos de simulación para que el robot aprenda de forma eficaz y correcta.

Definimos un experimento o prueba como el lanzamiento de una simulación en Gazebo, lo cual no es más que un conjunto de episodios. Un episodio es un número de pasos de simulación (con un máximo definido que se detalla en cada uno de los experimentos presentados).

Para estos experimentos se ha utilizado un ordenador portátil una CPU Intel i3-6006U con 2 GHz, 8 GB de RAM y gráfica integrada, tardando 42 segundos en instanciar un nuevo entorno, entre 3 y 24 segundos por paso de simulación y 6 segundos más en eliminar esa instancia de simulación. Este rango tan amplio en el tiempo de cada paso de simulación se debe a que, si el robot no ve el objeto y realiza las acciones de girar o moverse, las ejecuta durante 1 segundo, pero si lo ve, al seleccionar estas acciones, las efectúa hasta que cambia el valor de la distancia o del ángulo (según se mueva o gire, respectivamente) hasta un máximo 7 veces consecutivas (contando como un único paso de simulación), esperando 3 segundos entre cada acción. En el caso de que el algoritmo elija la acción de mover el brazo y el objeto no sea alcanzable, no lo traslada, pero si lo es, tarda 13 segundos en desplazar el brazo a la posición indicada. Realice la acción que realice, espera 3 segundos más para estabilizar la imagen.

Vamos a dividir los experimentos en dos casos diferentes, uno satisfactorio y otro no satisfactorio. Estos experimentos se diferencian únicamente en las recompensas que recibe el robot, enumeradas antes de mostrar los resultados.

4.1. Caso satisfactorio

Las recompensas utilizadas en este caso son todas las definidas en la Sección 3.3.3 en la Tabla 3.5, donde se puede observar en qué consiste la acción que produce la recompensa y su valor. Para este caso se han realizado 3 pruebas de aprendizaje. En la primera, se han realizado 129 episodios aleatorios (es decir, apareciendo el objeto y la mesa en posiciones distintas en cada episodio) de 200 pasos máximos cada uno, alcanzando el objeto en un $86{,}82\,\%$ de los episodios.

En las pruebas 2 y 3, se han realizado 55 episodios con el objeto saliendo siempre en la misma posición (justo enfrente a 1.05 m y a esa distancia desplazado 0.8 m a la izquierda del robot, respectivamente). En estos experimentos se han realizado menos episodios ya que son situaciones más simples de aprender, debido a la localización fija del objeto y la mesa entre episodios. En estas pruebas, el robot ha alcanzado el objeto el $96{,}36\,\%$, y $92{,}73\,\%$ de las veces, respectivamente. En todas ellas, el robot realiza movimientos bastante aleatorios en el primer episodio, como se puede apreciar, por ejemplo, en la Figura 4.1, que muestra la trayectoria del robot en el plano X-Y durante el episodio 1 del experimento 1. Por regla general, no consigue alcanzar el objeto en el primer episodio de un experimento, aunque debido a la aleatoriedad del aprendizaje, a veces es posible que lo logre.

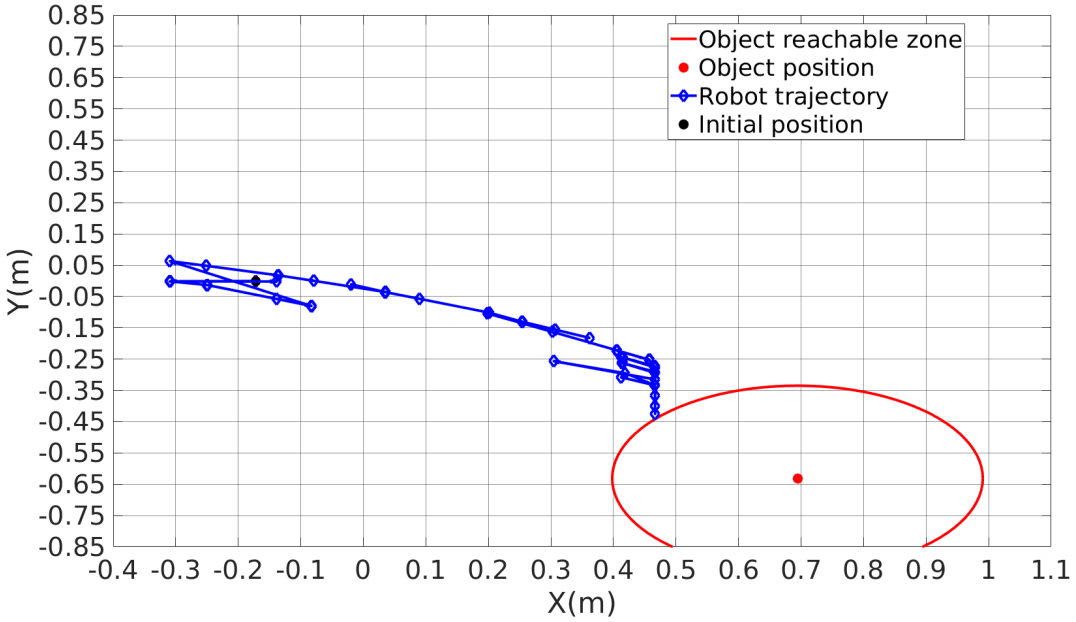


Figura 4.1: Ejemplo de primer episodio para el experimento 1 del caso satisfactorio.

Sin embargo, conforme se realizan episodios y el robot va aprendiendo, los movimientos suelen ser más lógicos para acercarse y alcanzar el objeto (Figura 4.2), donde los últimos movimientos los realiza para rectificar el ángulo con el que encara el objeto.

Este comportamiento de rectificación del ángulo sería complicado de implementar en una programación explícita, ya que se deberían tener en cuenta todas las posibles posiciones en las que tiene que realizarlo, así como las acciones necesarias para ejecutarlo (que en este caso consiste en retirarse del objeto, girar hacia él y acercarse). Sin embargo, utilizando este algoritmo de AR, el robot ha aprendido por su propia cuenta a llevar a cabo esta conducta.

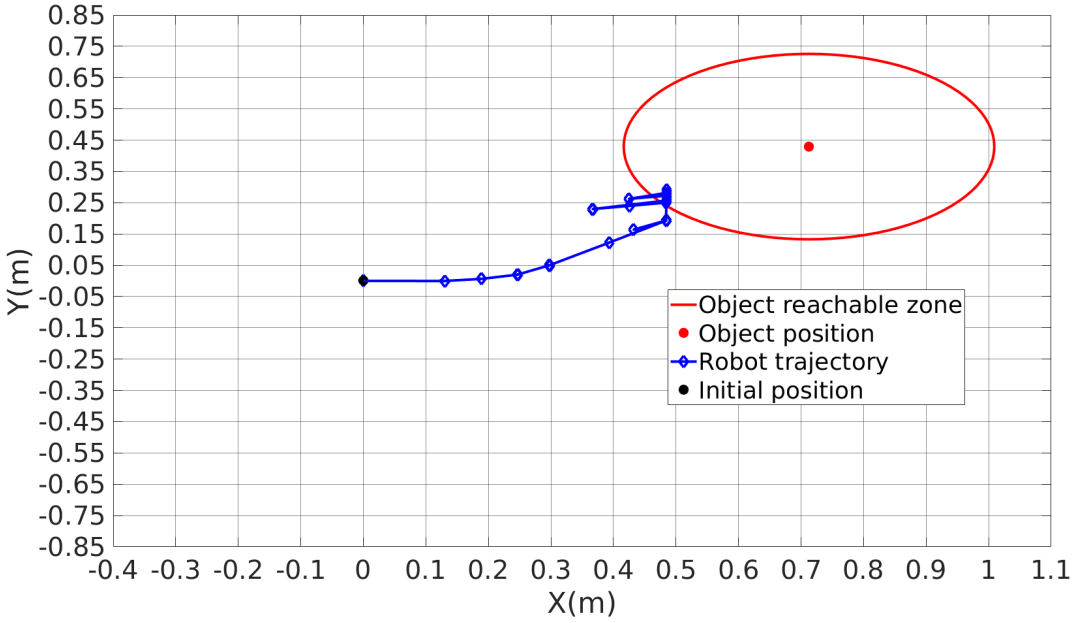


Figura 4.2: Episodio 128 para el experimento 1 del caso satisfactorio.

También puede que realice una serie de movimientos menos lógicos cuando pierde de vista el objeto, como se puede contemplar en la Figura 4.3. Esto se debe a que este estado (no ver el objeto) es muy amplio, debido a que considera todas las posibles poses (posiciones y ángulos) del robot en las que no ve el objeto, y, aunque en esta ocasión no deba girar a la izquierda para localizar el objeto, estará varios pasos girando en esta dirección, con un 30 % de posibilidades de explorar, realizando una acción aleatoria. Aún así sigue alcanzando el objeto, aunque en bastantes más pasos.

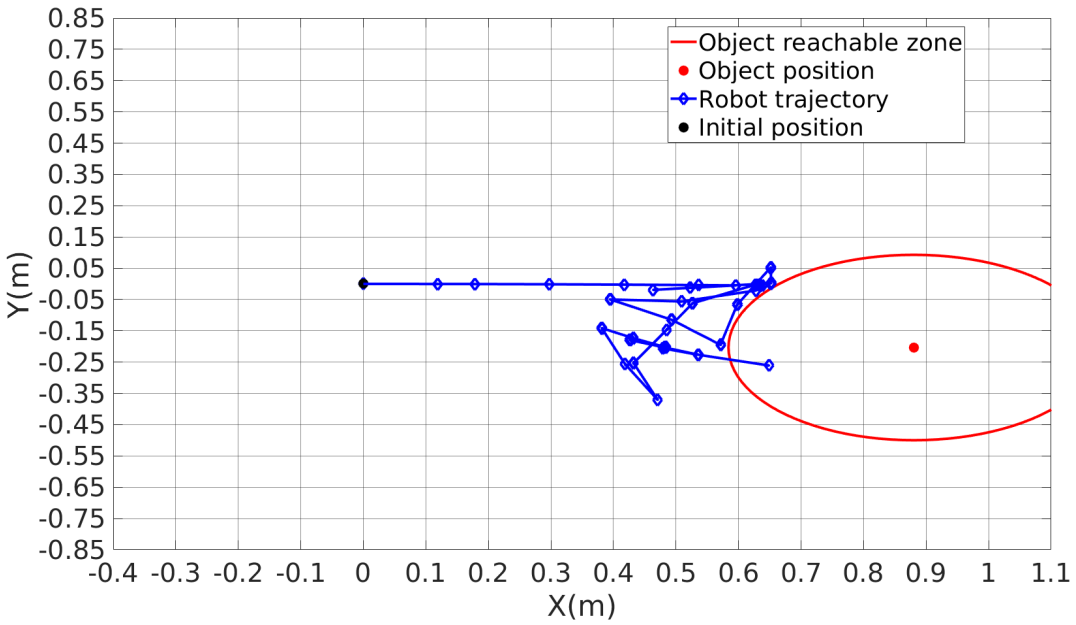


Figura 4.3: Episodio 129 para el experimento 1 del caso satisfactorio.

En las pruebas 2 y 3 se ha querido demostrar que, como mesa y objeto no varían su posición y, por tanto, se está aprendiendo en el mismo entorno, el número de pasos necesarios para alcanzar el objeto se observa prácticamente constante (Figuras 4.5 y 4.6). Sin embargo, en la Figura 4.4, que representa el número de pasos utilizados en cada episodio del experimento 1, debido a la aleatoriedad de la posición del objeto y la mesa, este número de pasos es más variable.

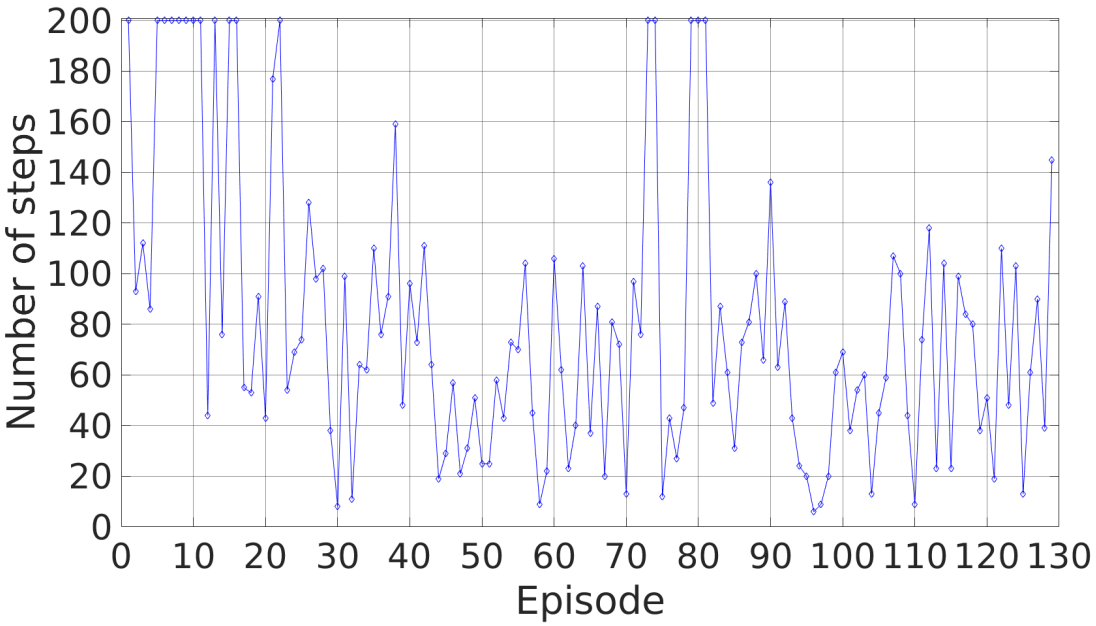


Figura 4.4: Número de pasos por episodio en el experimento 1 del caso satisfactorio.

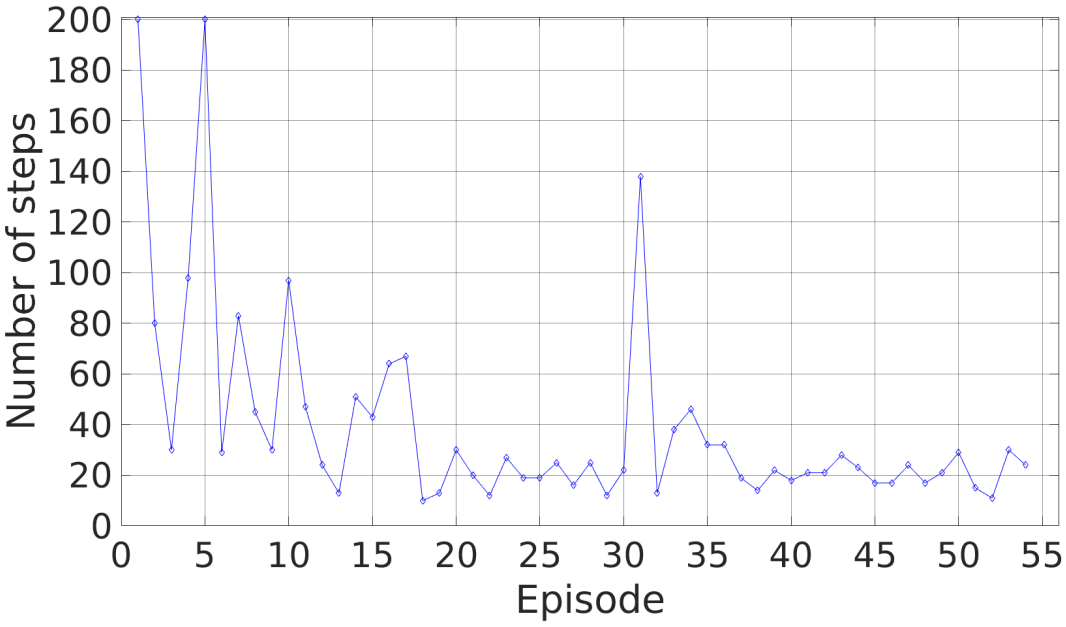


Figura 4.5: Número de pasos por episodio cuando el objeto está delante (prueba 2).

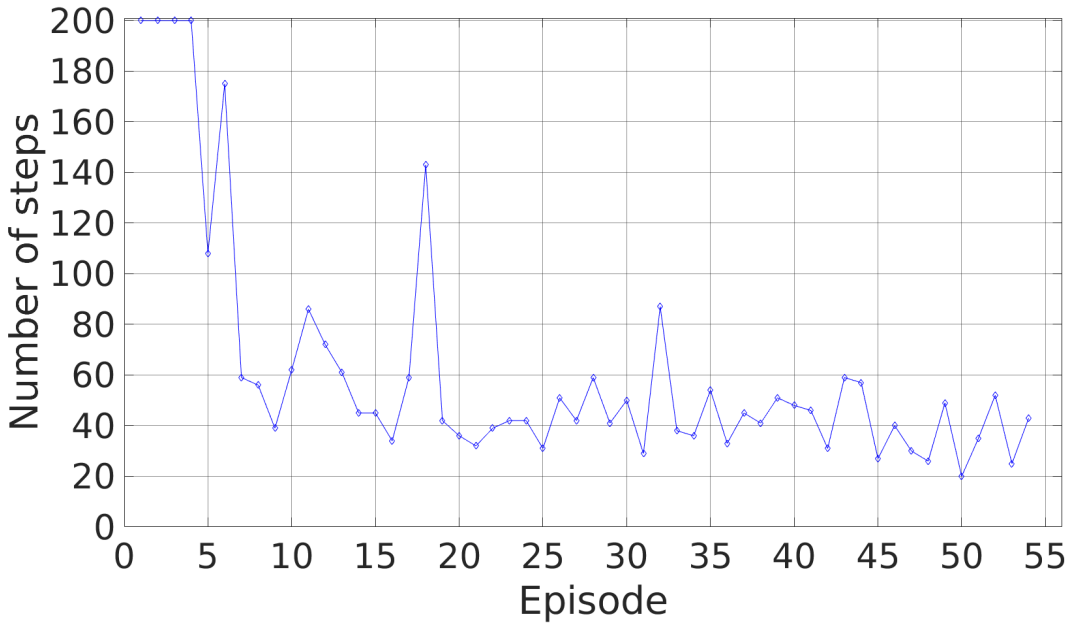


Figura 4.6: Número de pasos por episodio cuando el objeto está a la izquierda (prueba 3).

4.2. Caso insatisfactorio

La diferencia con el caso anterior es la recompensa por centrar o descentrar el objeto, que en este se ha eliminado. Esto provoca que el robot no tenga información sobre la zona por la que es alcanzable el objeto (recordemos que solamente puede alcanzarlo si lo tiene centrado), como se explica en la Sección 3.3.3.

En este caso se han realizado dos pruebas de 100 y 55 episodios respectivamente, repitiendo el entorno del último experimento del caso satisfactorio, donde el objeto aparece a la izquierda del robot (a 80 cm a la izquierda y a 1.05 m de distancia). Se ha elegido este entorno ya que es una situación compleja pero repetitiva, por lo que no debería tardar en aprender a alcanzar el objeto, pero tampoco debería ser trivial. Sin embargo, en la primera prueba, siendo el número de pasos totales por episodio de 200, alcanza el objeto el $4\,\%$ de las veces y en la segunda, con un número de pasos totales de 1000, el $12{,}73\,\%$ de las veces, no llegando a converger el número de pasos como en un aprendizaje correcto. Se ha realizado el mismo experimento con un número distinto de pasos máximos de simulación, para comprobar si el parámetro de aprendizaje incorrecto (por el que no alcanzaba el objeto) era este o las recompensas.

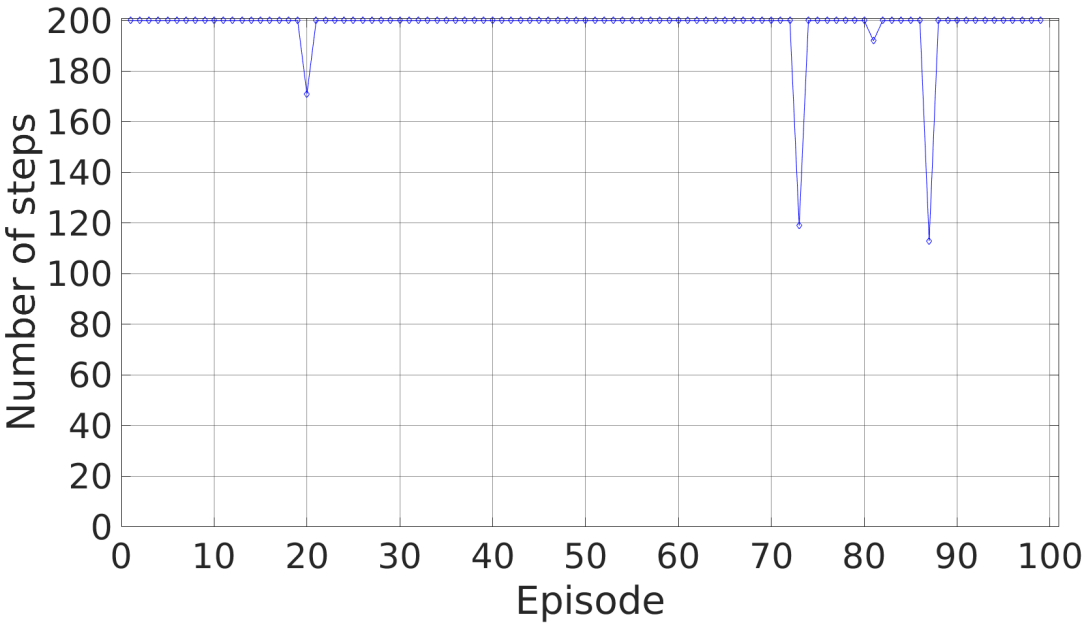


Figura 4.7: Número de pasos por episodio para el caso insatisfactorio con un máximo de 200 pasos.

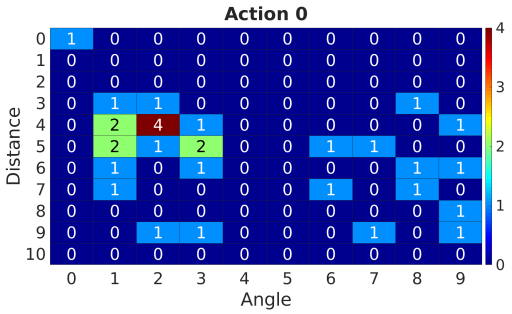
Figura 4.8: Número de pasos por episodio para el caso insatisfactorio con un máximo de 1000.

En las Figuras 4.7 y 4.8 se visualiza el número de pasos utilizados en cada episodio, no llegando a alcanzar el objeto si llega al número de pasos totales permitidos por episodio.

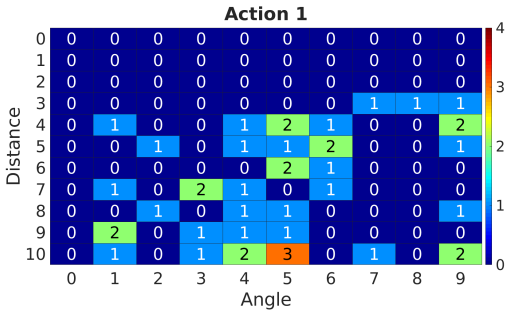
Así demostramos que la definición de las recompensas depende de la tarea a realizar y afecta mucho al resultado y que, al aumentar el número de pasos por episodio, aumentamos el porcentaje de acierto debido al factor aleatorio del aprendizaje (exploración).

4.3. Análisis básico de la política aprendida

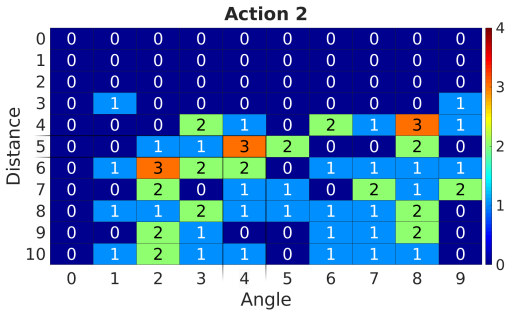
Una vez realizados los experimentos anteriores, vamos a mostrar la política aprendida por el robot en el experimento 1 del caso satisfactorio, ya que es la prueba que se ajusta mejor a un aprendizaje real, teniendo en cuenta diferentes posiciones del objeto y la mesa. En la Figura 4.9 mostramos 5 mapas de calor distintos (uno por cada posible acción, indicada a los pies de cada subfigura) donde el valor de cada celda muestra el número de estados (formados por el valor discreto de la distancia, ángulo y altura al objeto y valores binarios que indican si se ha alcanzado el objeto y si tiene el brazo plegado o no, definidos más detalladamente en la Sección 3.3.1) con el par (distancia, ángulo) que tiene dicha acción como política (es decir, como acción más favorable en ese estado). En los ejes X e Y se muestran los valores discretos del ángulo y de la distancia al objeto, respectivamente. Se han escogido estos valores del estado porque son los más representativos del mismo y, si se escogieran todos, tendríamos 4400 celdas en cada mapa de calor, en lugar de las 110 mostradas en los presentes. Debido a esto tenemos más de una política para la misma tupla (distancia, ángulo, acción). El significado de cada valor discreto de la distancia y el ángulo está explicado en la Sección 3.3.1 en las Tablas 3.2 y 3.3, respectivamente.



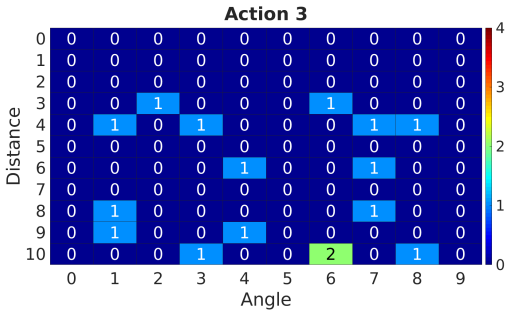
(a) Girar a la izquierda



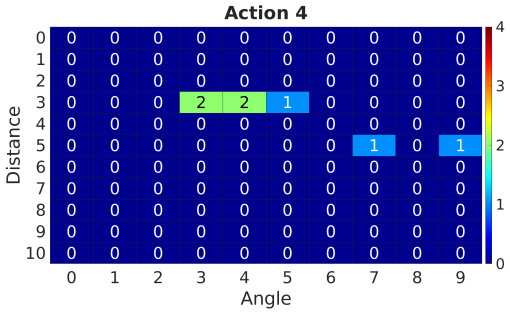
(b) Girar a la derecha



(c) Avanzar



(d) Retroceder



(e) Mover el brazo

Figura 4.9: Política del experimento 1 del caso satisfactorio.

Analizando la Figura 4.9, podemos apreciar que la política aprendida por el robot no es totalmente correcta. En esta política hay movimientos acertados, que son: girar a la izquierda cuando el objeto está a la izquierda de la imagen o no lo ve, a la derecha cuando el mismo está a la derecha del todo de la imagen, avanzar cuando el objeto no está al alcance, retroceder cuando el robot esta pegado a la mesa (distancias 3 y 4) y el objeto está descentrado, para rectificar el ángulo de alcance, y mover el brazo cuando el objeto está al alcance (distancia 3 y ángulos 3 al 5).

Sin embargo, hay muchos movimientos aprendidos erróneamente. Estos son girar a la izquierda cuando el objeto está a la derecha, rotar a la derecha cuando está a la izquierda o centrado, avanzar cuando el objeto está totalmente descentrado (por lo que lo perdería de vista), retroceder cuando el objeto está lejos o mover el brazo cuando el objeto no es alcanzable. Esto demuestra que, a pesar de que el robot ha aprendido a alcanzar el objeto en diversas situaciones, hay otras en las que todavía no ha aprendido adecuadamente, necesitando mayor número de episodios para completar su correcto aprendizaje.

Además, podemos reconocer algunos estados que no han sido explorados. Esto es debido a que no es posible alcanzarlos, ya que no tiene sentido que el robot no vea el objeto y nos proporcione un valor discreto de la distancia o ángulo distinto de 0, o que la distancia sea menor a 3, debido a que a esa distancia el robot estaría chocando con la mesa.

En las Figuras 4.10, 4.11, 4.12, 4.13 y 4.14 mostramos las distancias euclídeas entre los vectores V_i y V_{i-1} (los vectores V en el paso i e i-1) a lo largo de todos los pasos de la simulación lanzada en el experimento 1 del caso satisfactorio (siendo los diamantes rojos la distancia entre el vector V en el inicio de cada episodio, y en el último paso del episodio anterior). Recordemos que el vector V almacena la calificación máxima de la matriz *Q-Table* en cada estado y que dichas puntuaciones se actualizan siguiendo la Ecuación 1.1 explicada en la Sección 1.2:

$$Q_k(s,a) = (1-\alpha_k) * Q_{k-1}(s,a) + \alpha_k[R(s,a,s') + \gamma * V_{k-1}(s')]$$

De esta forma, como α converge a 0 según la ecuación $\frac{1}{visitas(estado,acción)^{0,5}}$, el nuevo valor de la matriz *Q-Table* será mínimo, y por tanto, la distancia entre los vectores V de pasos consecutivos, cuando se hayan explorado todos los pares (estado, acción) alcanzables un número considerable de veces, o no consigamos recompensa y el nuevo estado al que llegamos no haya sido explorado todavía. En las gráficas mostradas en las Figuras 4.10, 4.11, 4.12,

4.13 y 4.14, un valor alto de distancia implica que hay un cambio grande en matrices V de pasos consecutivos, y por tanto el robot ha aprendido en gran medida. Así, si observamos dichas Figuras, apreciamos que, aunque hay muchos pasos de ejecución que presentan una diferencia mínima, incluso en los últimos pasos sigue habiendo pasos consecutivos con una distancia considerable. Con esto confirmamos que el robot ha aprendido correctamente algunas políticas, pero todavía harían falta más episodios para un aprendizaje pleno.

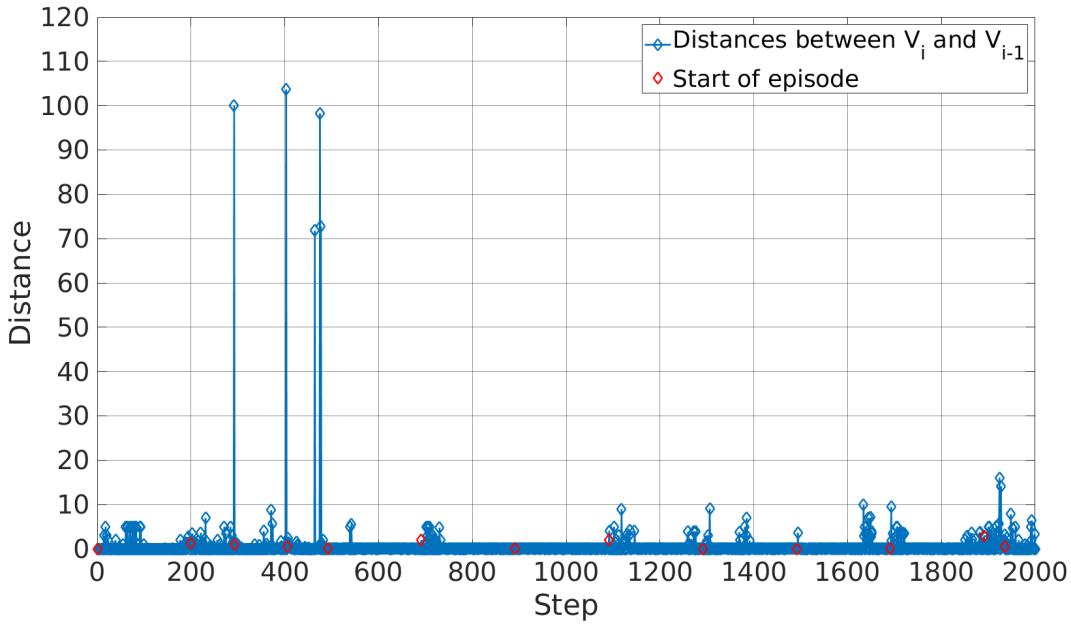


Figura 4.10: Diferencias entre V_i y V_{i-1} (pasos 0 al 2000).

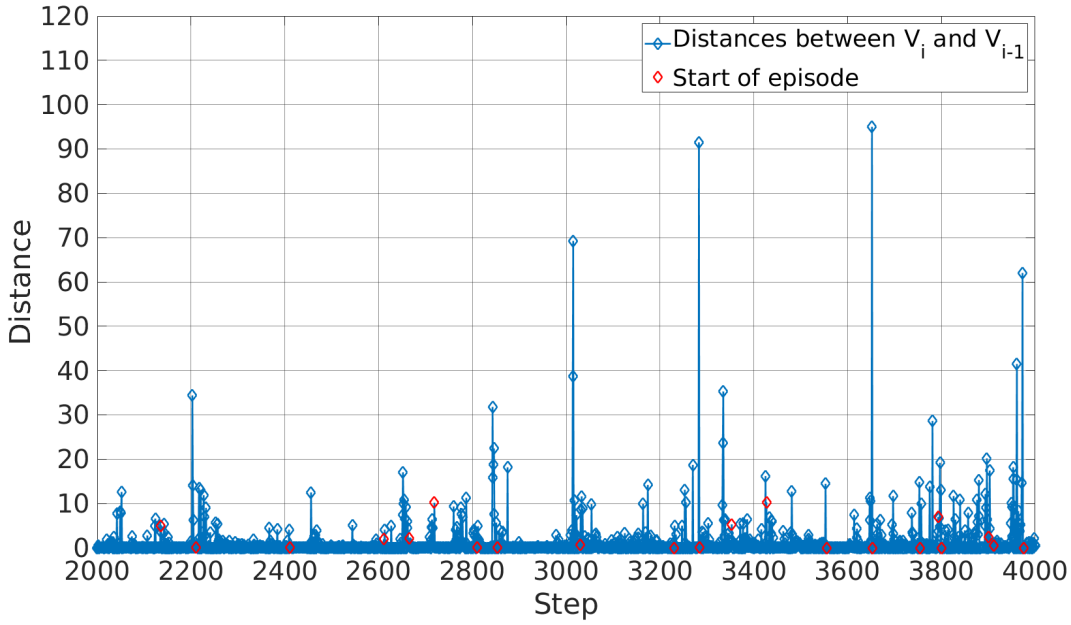


Figura 4.11: Diferencias entre V_i y V_{i-1} (pasos 2000 al 4000).

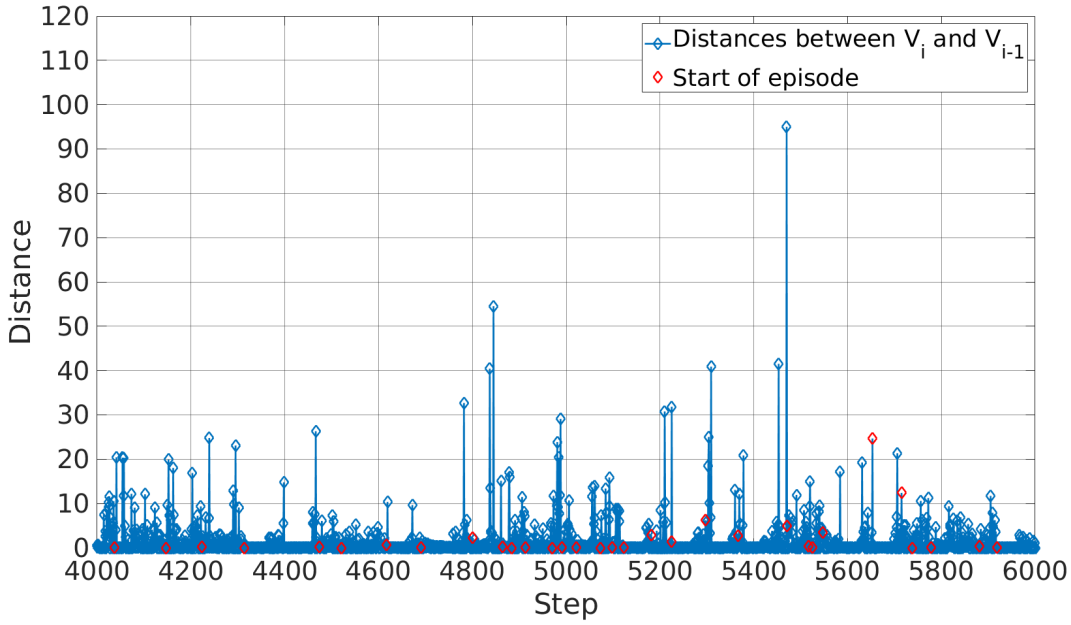


Figura 4.12: Diferencias entre V_i y V_{i-1} (pasos 4000 al 6000).

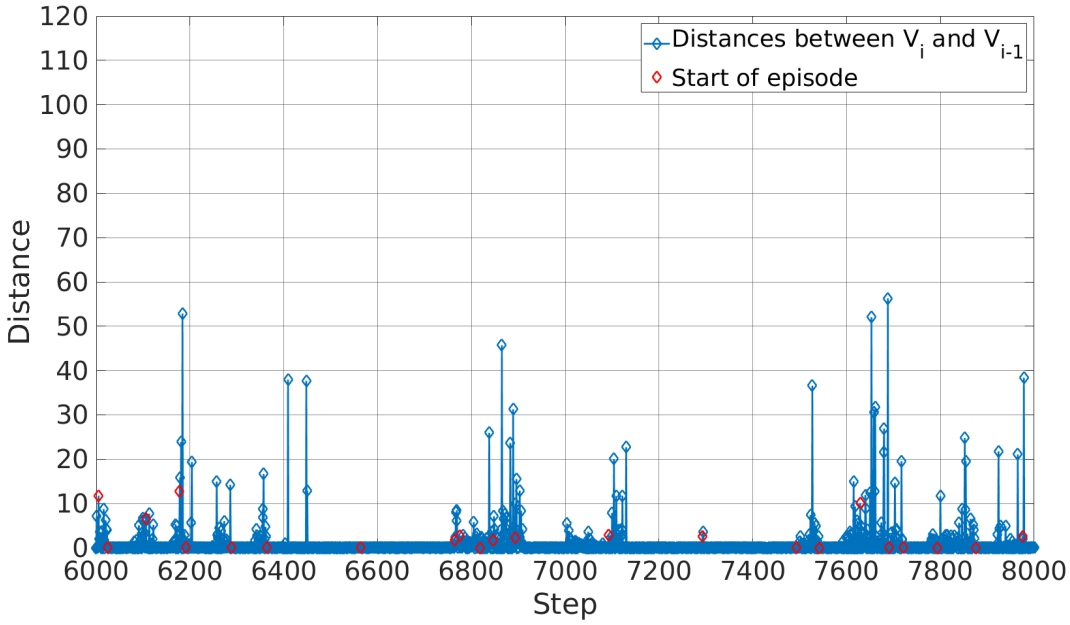


Figura 4.13: Diferencias entre V_i y V_{i-1} (pasos 6000 al 8000).

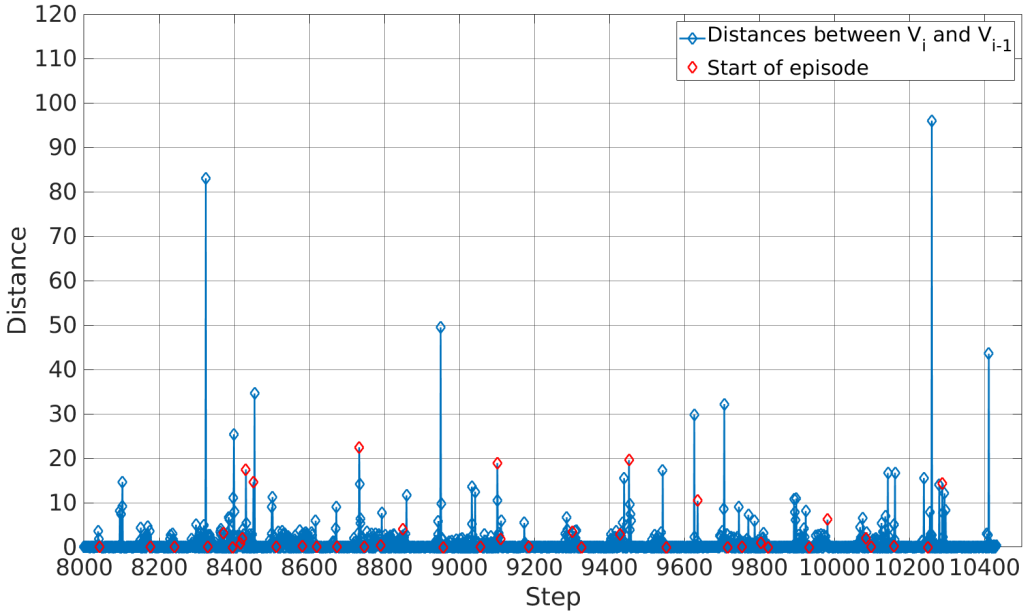


Figura 4.14: Diferencias entre V_i y V_{i-1} (ultimos pasos).

Capítulo 5

Conclusiones y trabajos futuros

En este último capítulo presentamos las conclusiones de este TFG y los trabajos futuros que se podrían derivar del mismo.

5.1. Conclusiones

Se ha conseguido implementar un algoritmo de aprendizaje por refuerzo para el robot CRUMB. Este algoritmo se encarga de que el robot aprenda a localizar un objeto, acercarse y alcanzarlo sin necesidad de una programación explícita del comportamiento. Para ello utilizamos el método de AR libre de modelos (*model-free*) *Q-Learning*.

Para esa finalidad, se ha tenido que poner a punto una plataforma de experimentación simulada y real, utilizando el programa Gazebo como simulador y resolviendo problemas posteriores. Entre estos problemas cabe destacar la detección y localización del objeto, así como el cálculo de la distancia a este, la lectura de las articulaciones del brazo, su movimiento hacia el objeto o la presencia de una pinza hueca en la simulación.

Con la implementación del algoritmo *Q-Learning*, se ha demostrado la mayor de las ventajas del AR frente a la programación explícita: no es necesario considerar todas las posibles situaciones en las que se podría encontrar el sujeto de la acción (en nuestro caso el robot CRUMB), que sí tendría que tenerse en cuenta en una programación explícita, aprendiendo estrategias complejas que resuelvan el problema (por ejemplo, retroceder a pesar de que no es lo mejor a priori, con el fin de mejorar la visión del objeto y obtener posteriormente un mejor alcance). Debido a esto, si se quieren incluir nuevas situaciones utilizando un método de AR,

como podría ser un nuevo entorno de trabajo, en el peor de los casos habría que modificar solamente el sistema de recompensas, mientras que en una programación explícita, habría que volver a implementar gran parte del código.

También se ha comprobado que con uno de los métodos más simples de AR (*Q-Learning*), se pueden realizar tareas complejas que hoy en día se realizan con métodos más laboriosos y costosos computacionalmente. Sin embargo, para que este método funcione correctamente tenemos que realizar un mayor esfuerzo en el procesado previo, como se puede apreciar en el capítulo 3, como puede ser la detección precisa de un objeto, el movimiento de un brazo manipulador o el ajuste minucioso de los parámetros de aprendizaje.

A modo de resumen, en la Tabla 5.1 se muestran los problemas más importantes aparecidos durante el desarrollo del TFG, así como la manera en la que se resolvió y la aportación que hace esa resolución a nuestro trabajo:

Problema	Resolución	Aportación
Imposibilidad de trabajar con imagen RGB para detectar el objeto	Transformación a imagen en modelo de color HSV	Posibilidad de trabajar la detección del objeto con OpenCV
Color rojo con valores de matiz entre 350 y 10 en el modelo de color HSV	Inversión del mapa de color de la imagen	Detección del objeto usando un único filtro para detectar color cian
Reflejos en un entorno real	Uso de filtros Gaussianos para suavizar la imagen y y descarte de la imagen de profundidad	Reducción del error en la localización del objeto en un entorno real
Diferencia entre las informaciones proporcionadas por el mismo *topic* en un entorno real y otro simulado	Diferenciación automática en código entre si estamos trabajando en un entorno real o simulado	Facilidad de ajuste del código implementado a un entorno real
Número de estados imposible de abarcar	Discretización y reducción de estados	Posibilidad de realizar un AR tabular y simplificación de la tarea

Posición del objeto variante en cada episodio	Cálculo del modelo cinemático directo e inverso en cada paso de simulación	Movimiento del brazo variable según posición del objeto
Efector final del brazo hueco (solamente la superficie externa es maciza)	Cálculo de posiciones para el correcto alcance del objeto	No dependencia de la información del esfuerzo realizado por la pinza para el alcance del objeto, la cual es errónea
Imagen, sensor y brazo manipulador en diferentes sistemas de referencia	Creación de matrices de transformación entre los distintos sistemas	Localización del objeto con respecto al sistema de referencia del brazo
Error en el cálculo de la distancia debido al proceso de obtención de la misma	Restricción del espacio de trabajo en simulación	Mayor precisión en el cálculo de la distancia
Baja rapidez y precisión en el aprendizaje debido a la escasez de recompensas	Incremento en el número de recompensas	Mayor rapidez y precisión en el aprendizaje
Bucle de "acción incorrecta - acción correcta"para recibir recompensas indefinidamente	Añadido de recompensas negativas en modo espejo	Aprendizaje más correcto al no entrar en bucle
Permanencia en el mismo estado eventualmente al realizar una acción de movimiento o giro de la base robótica	Modificación del algoritmo de forma que realice la acción de movimiento o giro de la base hasta que cambie el valor discreto de la distancia o ángulo, respectivamente	Aprendizaje más certero y rápido
Recompensación excesiva de acciones según orden de realización durante el aprendizaje	Parámetro α modificado a $\frac{1}{visitas(estado,acción)^{0,5}}$	Aprendizaje más gradual e independencia del orden de realización de las acciones en el mismo

Tabla 5.1: Problemas resueltos en este trabajo y aportación.

5.2. Líneas de trabajo futuras

A continuación, proponemos las siguientes líneas de trabajo que podrían realizarse gracias a este Trabajo de Fin de Grado:

- Mejorar el aprendizaje. Ahora tenemos la estructura básica y un análisis de las dificultades encontradas, pero el aprendizaje es mejorable.
- Realizar la implementación de un método de AR basado en modelos (*model-based*) para la misma tarea, elaborando una comparativa de ambos métodos. Esto podría proporcionar una vista perfecta de cuáles son las ventajas y desventajas de los dos tipos de métodos y si uno de ellos es mejor que el otro para la tarea planteada.
- Ajustar detalles de la implementación realizada para su perfecto funcionamiento en un entorno real con el robot CRUMB, sin hacer uso de simulaciones. Esto permitiría que el robot se aplicase en tareas concretas de la industria o para mejorar la calidad de vida de los ciudadanos.
- Mejorar los sistemas de localización del objeto, incrementando la precisión de los mismos e insertando un sistema de visión por computador más complejo. De esta forma se permitiría la localización de varios objetos a la vez, los cuales no tendrían por qué ser rojos, ampliando su aplicación en la industria o la vida cotidiana, además de un aprendizaje con más grados de discretización y distancia posible.
- Aumentar la precisión del movimiento del brazo, realizando trayectorias más exactas, así como rectificar la información proporcionada por los *topics* para que no haya discrepancias entre la suministrada por la simulación y el robot real.
- Implementar un sistema de control remoto para ejecutar el aprendizaje en el robot real de forma eficiente y simple, mediante un protocolo SSH. Actualmente, el ordenador que ejecuta el algoritmo tiene que estar físicamente conectado al robot, por lo que el aprendizaje no se puede ejecutar en un ordenador de sobremesa para mayor potencia de procesado. Esto sería posible utilizando un control remoto.

- Migración a ROS Melodic para evitar la posible obsolescencia de los programas implementados para el robot. ROS Melodic ha ampliado su gama de sistemas operativos soportados, incluyendo Windows 10 y permitiendo su uso desde versiones más modernas de Linux y iOS, además de una mejora en el lenguaje de implementación. El lenguaje de programación pasa a ser C++14 en lugar del 03 permitido por ROS Indigo. Todo esto en conjunto ampliaría el uso de los programas implementados a ordenadores personales con diferentes sistemas operativos y facilitaría la labor de implementación de nuevas características.
- Actualizar el *hardware* del robot por uno más novedoso y preciso. Esto mejoraría el trabajo de este algoritmo pero, disminuyendo costes computacionales, se aumentarían los de infraestructura.

Apéndice A

Manual de instalación

En este manual de instalación se explicará cómo instalar todo lo necesario para ejecutar el algoritmo de AR implementado, así como la simulación del aprendizaje que realiza el robot, y la posible implementación de más algoritmos para el robot CRUMB.

A.1. Instalación de Ubuntu 14.04.5 LTS

Primero tendremos que instalar el sistema operativo Ubuntu 14.04 LTS en una partición de nuestro ordenador.

Tiene que ser esa versión de Ubuntu y no otra porque nuestro *Turtlebot* utiliza ROS Indigo, el cual se ha comprobado, mediante una serie de pruebas, que solo es compatible con esa versión de Ubuntu, según se explica en la sección 3.1.

Para la instalación tendremos que realizar los siguientes pasos:

1. Descargar *Ubuntu 14.04.5 LTS (Trusty Tahr) desktop image* de la página oficial de Ubuntu - `http://releases.ubuntu.com/14.04/` [37].

2. Descargar Rufus para poder instalar el sistema operativo desde un dispositivo *Universal Serial Bus* (USB), desde `https://rufus.ie/` [38].

3. Convertir un dispositivo USB en uno de arranque con Rufus, utilizando para ello la imagen .iso de Ubuntu anteriormente descargado.

4. Crear una partición vacía (con unos 70GB de almacenamiento será suficiente).

5. Instalar el sistema operativo en la partición, ejecutando el dispositivo USB desde el sistema de arranque.

A.2. Actualización de cmake

A continuación debemos actualizar *cmake* [39] (un compilador de archivos CMakeList), ya que en un futuro podríamos querer ejecutar la simulación en V-REP 3.4 (la versión mínima que admite ROS). Esta actualización es muy importante hacerla en este momento, porque dicha versión de V-REP necesita *cmake* 3.0, mientras que Ubuntu 14.04 LTS nos provee por defecto de *cmake* 2.8.

Además, todo lo que vamos a instalar posteriormente necesita de *cmake*, por lo que si en un futuro quisiéramos actualizar la versión de esta librería, tendríamos que volver a instalar todas las librerías necesarias para el aprendizaje de nuevo.

Para realizar dicha actualización, debemos ejecutar los siguientes comandos en el terminal [1]:

```
$ sudo apt-get remove cmake cmake-data
$ sudo apt-get install cmake3 cmake3-data
```

A.3. Instalación y preparación de ROS Indigo

Una vez tenemos nuestro sistema operativo y la librería *cmake* actualizada, podemos proceder con la instalación de ROS. Instalaremos ROS Indigo, ya que se ha probado en la Sección 3.1 que es el único compatible con el robot utilizado. Para ello procedemos con los siguientes pasos:

1. Seguimos las indicaciones del siguiente tutorial, instalando *ros-indigo-desktop-full*: `http://wiki.ros.org/indigo/Installation/Ubuntu` [41].
   Si da error después de ejecutar el comando *install*, debemos introducir las siguientes líneas en el terminal antes de volver a efectuarlo:

   ```
   $ sudo apt-get update
   $ sudo apt-get -f upgrade
   $ sudo apt-get autoremove
   ```

[1] Según se indica en [40]

2. Creamos el directorio de trabajo siguiendo el tercer apartado del siguiente tutorial (para *catkin*, el cual es el sistema de configuración y compilación oficial de ROS):

   `http://wiki.ros.org/ROS/Tutorials/InstallingandConfiguringROSEnvironment` [42].

3. Copiamos los siguientes paquetes del material adjunto al proyecto a la carpeta *src* del directorio de trabajo:

   - Arbotix_ros
   - Kobuki_testing
   - Widow_arm
   - Widow_arm_testing

4. Nos trasladamos a nuestro directorio de trabajo creado en el apartado previo y desde ahí, a la carpeta *src/arbotix_ros/arbotix_python/bin*.

   A continuación, damos permiso de ejecución a los archivos de la carpeta ejecutando el siguiente comando (esto lo realizamos para después poder utilizar el brazo):

```
$ sudo chmod +x *
```

5. Instalamos varios paquetes necesarios para el proyecto con la siguiente instrucción:

```
$ sudo apt-get install ros-indigo-hokuyo-node ros-indigo-kobuki ros
   -indigo-kobuki-core ros-indigo-kobuki-desktop ros-indigo-kobuki-
   msgs ros-indigo-yocs-msgs ros-indigo-yujin-ocs ros-indigo-
   freenect-camera ros-indigo-freenect-launch ros-indigo-freenect-
   stack ros-indigo-map-store ros-indigo-turtlebot ros-indigo-
   turtlebot-apps ros-indigo-turtlebot-create ros-indigo-turtlebot-
   create-desktop ros-indigo-turtlebot-msgs ros-indigo-turtlebot-
   simulator ros-indigo-turtlebot-dashboard ros-indigo-turtlebot-
   interactive-markers ros-indigo-turtlebot-rviz-launchers ros-
   indigo-widowx-arm ros-indigo-widowx-arm-controller ros-indigo-
   arbotix-msgs
```

6. Instalamos armadillo [43], que es una librería de álgebra lineal para C++, siguiendo el tutorial mostrado en la página web `https://mecatronicauaslp.wordpress.com/2013/12/17/instalar-armadillo-biblioteca-de-algebra-lineal-en-visual-studio-2012-x86/` [44] hasta el apartado 4 inclusive, para poder usar los códigos implementados en este TFG. A continuación se debe ejecutar la siguiente línea de código en el terminal:

```
$ sudo apt-get install libarmadillo* libblas3 libblas-dev
```

7. Volvemos a nuestro directorio de trabajo, en nuestro caso *~/catkin_ws* y ejecutamos el siguiente comando para crear los ficheros donde se encontrarán nuestros nodos ROS:

```
$ catkin_make
```

8. Para mayor comodidad, introducimos el comando *"source devel/setup.bash"* en el archivo oculto *bashrc* de Linux, para no tener que estar ejecutándolo constantemente en cada terminal que abramos.

   Para ello introducimos los siguientes comandos en el terminal, siendo *~/catkin_ws/* la ruta completa a mi directorio de trabajo:

```
$ echo "source ~/catkin_ws/devel/setup.bash" >> ~/.bashrc
$ source ~/.bashrc
```

A.4. Instalación de simuladores

Una vez instalado Ubuntu y la distribución de ROS pertinente, procederemos a instalar los simuladores. Con ellos seremos capaz de realizar las distintas pruebas de aprendizaje sobre el robot sin que este realice esfuerzos mecánicos que pongan en riesgo sus componentes, tales como motores o articulaciones.

A.4.1. Instalación de Gazebo ROS

Para la instalación de este simulador, se ha optado por seguir la guía de instalación proporcionada por los tutores del trabajo, perteneciente al TFM titulado "Modelado y simulación de

un robot *Turtlebot 2* con un brazo manipulador *Widow-X* sobre ROS y Gazebo"de la alumna Marina Aguilar Moreno [18].

Se incluye la parte de dicho manual en nuestro TFG para mayor comodidad del lector, añadiendo la resolución de algunos errores detectados:

A.4.1.1. Instalación y configuración de Gazebo

1. Comprobar que Gazebo funciona correctamente de forma aislada.

```
$ gazebo
```

   Con lo que debería abrirse la pantalla principal de Gazebo. Si no nos aparece esta pantalla pero surge un error *namespace not found*, debemos ejecutar los siguientes comandos[2]:

```
$ sudo sh -c 'echo "deb http://packages.osrfoundation.org/gazebo/
    ubuntu trusty main" > /etc/apt/sources.list.d/gazebo-latest.list
    '
$ wget http://packages.osrfoundation.org/gazebo.key -O - | sudo apt
    -key add -
$ sudo apt update
$ sudo apt upgrade
$ echo "export LC_NUMERIC=C" >> ~/.bashrc
$ source ~/.bashrc
```

2. Comprobar que la versión está instalada correctamente:

```
$ which gzserver
$ which gzclient
```

   Con lo que debería obtenerse:

```
/usr/bin/gzserver
/usr/bin/gzclient
```

3. Instalar *gazebo-ros-packages*:

```
$ sudo apt-get install ros-indigo-gazebo-ros-pkgs ros-indigo-gazebo
    -ros-control
```

[2]Código recogido de [45]

4. Comprobar la integración ROS/Gazebo. Para ello primero se abrirá el núcleo de ROS:

```
$ echo "source /opt/ros/indigo/setup.bash" >> ~/.bashrc
$ source ~/.bashrc
$ roscore
```

Abrir otra terminal e iniciar Gazebo desde ROS:

```
$ rosrun gazebo_ros gazebo
```

Una vez ejecutado debería abrirse la ventana de Gazebo. Finalmente, para comprobar las conexiones entre ROS y Gazebo, abrir otro terminal y escribir:

```
$ rostopic list
```

Con lo que debería obtenerse:

```
/clock
/gazebo/link_states
/gazebo/model_states
/gazebo/parameter_descriptions
/gazebo/parameter_updates
/gazebo/set_link_state
/gazebo/set_model_state
/rosout
/rosout_agg
```

Y en la misma terminal escribir:

```
$ rosservice list
```

Obteniéndose los servicios de ROS y Gazebo:

```
/gazebo/apply_body_wrench
/gazebo/apply_joint_effort
/gazebo/clear_body_wrenches
/gazebo/clear_joint_forces
/gazebo/delete_model
/gazebo/get_joint_properties
/gazebo/get_link_properties
/gazebo/get_link_state
/gazebo/get_loggers
```

```
/gazebo/get_model_properties
/gazebo/get_model_state
/gazebo/get_physics_properties
/gazebo/get_world_properties
/gazebo/pause_physics
/gazebo/reset_simulation
/gazebo/reset_world
/gazebo/set_joint_properties
/gazebo/set_link_properties
/gazebo/set_link_state
/gazebo/set_logger_level
/gazebo/set_model_configuration
/gazebo/set_model_state
/gazebo/set_parameters
/gazebo/set_physiscs_properties
/gazebo/spawn_gazebo_model
/gazebo/spawn_sdf_model
/gazebo/spawn_urdf_model
/gazebo/unpause_physics
/gazebo/get_loggers
/gazebo/set_logger_level
```

Con esto queda instalado Gazebo y conectado con ROS.

A.4.1.2. Instalación Turtlebot

Como se ha indicado a lo largo de la memoria, CRUMB utilizará los paquetes de *Turtlebot* tal y como aparecen en el repositorio de ROS, aunque se han realizado algunas modificaciones que se encuentran en los ficheros adjuntos al proyecto relativos a CRUMB.

Así pues, para instalar los paquetes de *Turtlebot*, únicamete hay que descargarlos desde la terminal:

```
$ sudo apt-get install ros-indigo-turtlebot ros-indigo-turtlebot-apps
   ros-indigo-turtlebot-interactions ros-indigo-turtlebot-simulator ros-
   indigo-kobuki-ftdi ros-indigo-rocon-remocon ros-indigo-rocon-qt-
   library ros-indigo-ar-track-alvar-msgs
```

Una vez instalados ya se puede visualizar *Turtlebot* desde Gazebo:

```
$ roslaunch turtlebot_gazebo turtlebot_world.launch
```

Con lo que debería obtenerse la Figura A.1

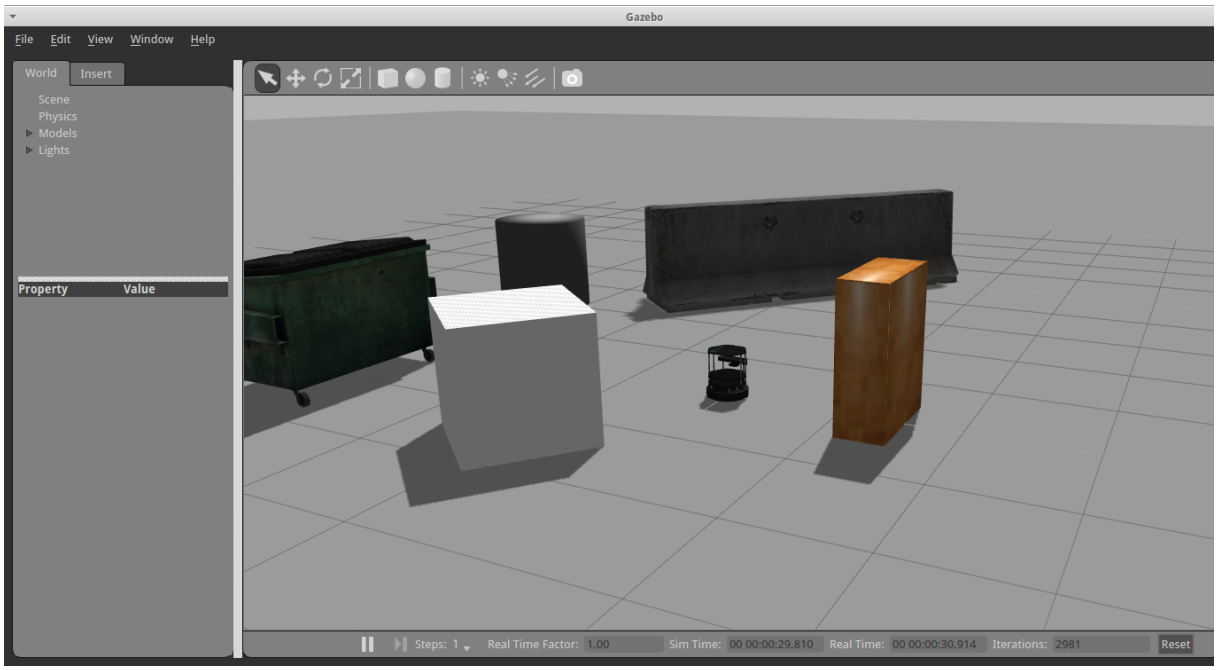


Figura A.1: Simulación Turtlebot en Gazebo.

Si la pantalla de Gazebo se bloquea, mostrando una pantalla en negro, puede ser debido a que la anterior ejecución del programa no se interrumpió correctamente. Para arreglar eso debemos ejecutar:

```
$ killall -9 gzserver
```

Y para manejarla desde teclado, abrir otra terminal:

```
$ roslaunch tutlebot_teleop keyboard_teleop.launch
```

A.4.1.3. Instalación CRUMB

Finalmente, hay que descargar los paquetes de CRUMB y compilarlos desde la terminal. Para ello, guardar los paquetes en la carpeta *catkin_ws/src* creada previamente y copiar en la carpeta CRUMB, el *plugin roboticsgroup_gazebo_plugins* y el paquete *hello_turtlebot* que contiene la librería para navegación. Los paquetes de CRUMB se encuentran en el repositorio *GitHub* [8].

Una vez obtenidos los paquetes, hay que compilarlos desde la ventana terminal:

```
$ cd ~/catkin_ws/
$ catkin build
```

Ya debería poder ejecutarse en el simulador Gazebo el robot CRUMB:

```
$ roslaunch crumb_gazebo crumb_world.launch
```

Ejecutando ese código debería obtenerse la Figura A.2

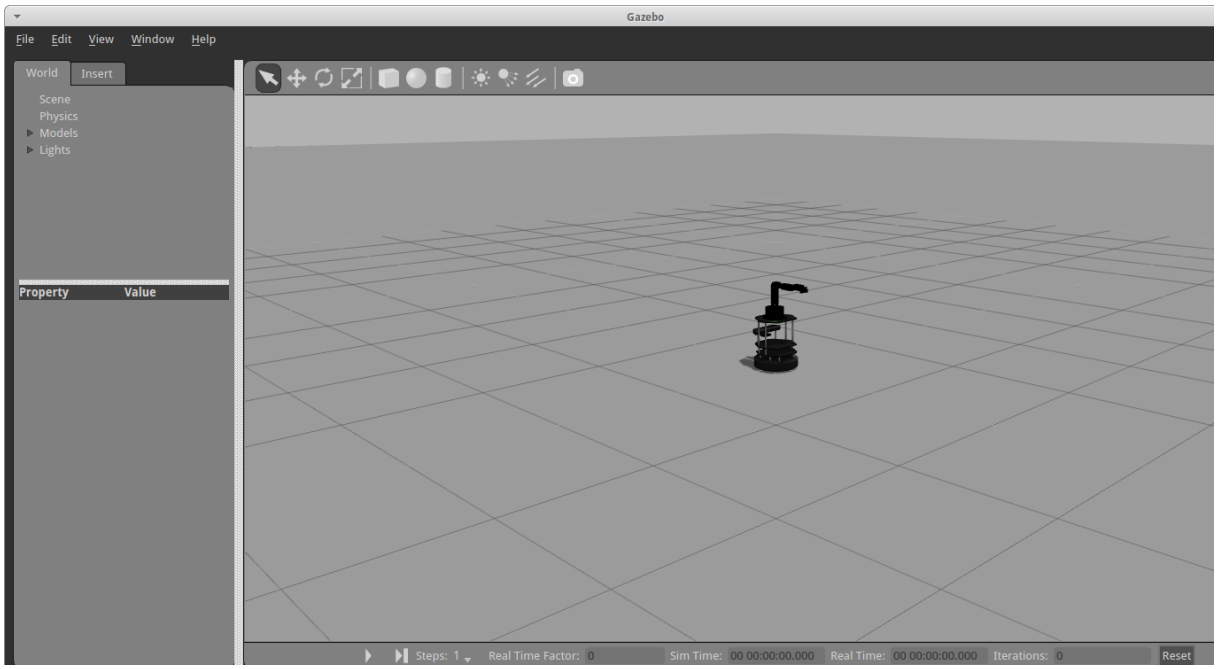


Figura A.2: Simulación CRUMB en Gazebo.

Para trabajar con el simulador sólo hay que darle al botón de arranque y cargar alguna aplicación. Para ver todos los *topics* que se han incluido en CRUMB solamente hay que introducir por la terminal:

```
$ rostopic list
```

Estos *topics* se pueden dividir en 3 grupos, dependiendo de si son relativos al brazo, a la plataforma o a ambos:

- *Topics* para el brazo
    - */arm_1_joint/command*
    - */arm_1_joint/pid/parameter_descriptions*
    - */arm_1_joint/pid/parameter_updates*
    - */arm_1_joint/state*
    - */arm_2_joint/command*
    - */arm_2_joint/pid/parameter_descriptions*
    - */arm_2_joint/pid/parameter_updates*
    - */arm_2_joint/state*
    - */arm_3_joint/command*
    - */arm_3_joint/pid/parameter_descriptions*
    - */arm_3_joint/pid/parameter_updates*
    - */arm_3_joint/state*
    - */arm_4_joint/command*
    - */arm_4_joint/pid/parameter_descriptions*
    - */arm_4_joint/pid/parameter_updates*
    - */arm_4_joint/state*
    - */arm_5_joint/command*
    - */arm_5_joint/pid/parameter_descriptions*
    - */arm_5_joint/pid/parameter_updates*
    - */arm_5_joint/state*
    - */gripper_1_joint/command*
    - */gripper_1_joint/pid/parameter_descriptions*
    - */gripper_1_joint/pid/parameter_updates*
    - */gripper_1_joint/state*

- *Topics* para la plataforma
    - */camera/depth/camera_info*
    - */camera/depth/image_raw*
    - */camera/depth/points*
    - */camera/parameter_descriptions*
    - */camera/parameter_updates*
    - */camera/rgb/camera_info*
    - */camera/rgb/image_color*
    - */camera/rgb/image_color/compressed*
    - */camera/rgb/ image_color/compressed/parameter_descriptions*
    - */camera/rgb/ image_color/compressed/parameter_updates*
    - */camera/rgb/compressedDepth*
    - */camera/rgb/compressedDepth/parameter_descriptions*
    - */camera/rgb/compressedDepth/parameter_updates*
    - */camera/rgb/image_color/theora*
    - */camera/rgb/image_color/theora/parameter_descriptions*
    - */camera/rgb/theora/parameter_updates*
    - */cmd_vel_mux/active*
    - */cmd_vel_mux/input/navi*
    - */cmd_vel_mux/input/safety_controller*
    - */cmd_vel_mux/input/teleop*
    - */cmd_vel_mux/parameter_descriptions*
    - */cmd_vel_mux/parameter_updates*
    - */depthimage_to_laserscan/parameter_descriptions*
    - */depthimage_to_laserscan/parameter_updates*
    - */laserscan_nodelet_manager/bond*
    - */mobile_base/commands/led1*
    - */mobile_base/commands/led2*
    - */mobile_base/commands/motor_power*

- */mobile_base/commands/reset_odometry*
- */mobile_base/commands/velocity*
- */mobile_base/events/bumper*
- */mobile_base/events/cliff*
- */mobile_base/events/wheel_drop*
- */mobile_base/sensors/bumper_pointcloud*
- */mobile_base/sensors/core*
- */mobile_base/sensors/imu_data*
- */mobile_base/sensors/imu_data_raw*
- */mobile_base_nodelet_manager/bond*
- */odom*

- *Topics* compartidos

  - */clock*
  - */gazebo/link_states*
  - */gazebo/model_states*
  - */gazebo/parameter_descriptions*
  - */gazebo/parameter_updates*
  - */gazebo/set_link_state*
  - */gazebo/set_model_state*
  - */joint_states*
  - */rosout*
  - */rosout_agg*
  - */scan*
  - */simulation/noise*
  - */simulation/pose*
  - */simulation/real_time*
  - */simulation/sim_time*
  - */tf*
  - */tf_static*

Puede ocurrir que los *topics* del brazo no aparezcan, mostrando errores correspondientes a *effort-controllers* al ejecutar la simulación de Gazebo. Si eso sucede, ejecutar:

```
$ sudo apt-get install ros-indigo-ros-controllers
```

Si al ejecutar algún fichero *.launch* nos aparece un error de permisos en */home/marina*, debido a que algunas de las librerías utilizadas fueron implementadas por Marina Aguilar Moreno en su TFM [18] desde su propio sistema de archivos, podemos realizar dos operaciones:

1. Crear la carpeta */home/marina*.
2. Irnos al fichero *.world* que ejecuta el *.launch* ejecutado y modificarlo de una de las dos formas:
   - Cambiar la línea donde establece la dirección del vídeo de */home/marina/vídeos* a otra de nuestro sistema de ficheros.
   - Cambiar la variable *"save enabled"* del vídeo a *false* (con esta segunda opción no se guardará ningún vídeo de la simulación).

A.4.2. Instalación de V-REP (Opcional)

Para instalar este simulador en Ubuntu 14.04 LTS con las prestaciones del *plugin RosInterface*, debemos seguir los siguientes pasos:

1. Descargar la versión 3.4.0 PRO EDU de la página web `www.coppeliarobotics.com/previousversions.html` [27].
2. Descomprimir el archivo descargado.
3. Descargar los modelos de la página web de CRUMB [8], trasladándonos a la carpeta deseada y ejecutando:

   ```
   $ git clone https://github.com/CRUMBproject/V-REP.git
   ```

4. Entrar al directorio *V-REP_PRO_EDU_V3_4_0_Linux/programming/ros_packages* y copiar la carpeta *v_repExtRosInterface* en el *src* de nuestro directorio de trabajo de *catkin*.

5. Inicializar la variable de entorno *VREP_ROOT* de la siguiente forma:

```
$ echo 'export VREP_ROOT="path/to/vrep/V-REP_PRO_EDU_V3_4_0_Linux"'
    >> ~/.bashrc
$ source ~/.bashrc
```

6. Ejecutar el siguiente comando en nuestro directorio de trabajo:

```
$ catkin build
```

   Si da fallos debido a que ya existan unas carpetas *build* y *devel*, eliminarlas y volver a ejecutar el comando. A partir de este momento, todas las compilaciones de *catkin* deberán realizarse con este comando.

A.5. Preparación para usar el brazo Widow-X

Para poder utilizar el brazo manipulador del robot CRUMB, primero hay que realizar una serie de preparativos y comprobaciones.

1. Conectamos el robot al puerto *ttyUSB0* de nuestro ordenador. Para saber si está conectado a ese puerto específico, ejecutamos el comando *"ls -l /dev/"* antes y después de conectar el brazo, y comprobamos que el puerto que aparece al conectarlo es el */dev/ttyUSB0*.

2. Concedemos permiso de lectura y escritura a ese puerto con el siguiente comando:

```
$ sudo chmod a+rw /dev/ttyUSB0
```

3. Nos trasladamos a nuestro directorio de trabajo de *catkin* y, desde este, a la carpeta *src/arbotix_ros/arbotix_python/bin* y ejecutamos:

```
$ ./arbotix_terminal
```

4. Ejecutamos el comando ls y comprobamos que reconoce todas las articulaciones del brazo, apareciendo de la 1 a la 6 consecutivamente. Si no aparecen todas, hay que seguir los manuales de las páginas web `https://learn.trossenrobotics.com/arbotix/arbotix-quick-start.htm` [46] y `https://learn.trossenrobotics.com/arbotix/1-using-the-tr-dynamixel-servo-tool#&panel1-1` [19] con estas variaciones:

- En el segundo apartado de la segunda página proporcionada, no hace falta instalar nada ya que Ubuntu contiene los archivos de configuración *Future Technology Devices International* (FTDI) necesarios por defecto.
- Para que el programa DynaManager reconozca las articulaciones, estas han de ser conectadas una a una a la placa *Arduino* del brazo.

A.6. Configuración del láser

El láser del robot también nos puede dar el siguiente fallo al ejecutarlo por primera vez (Figura A.3):

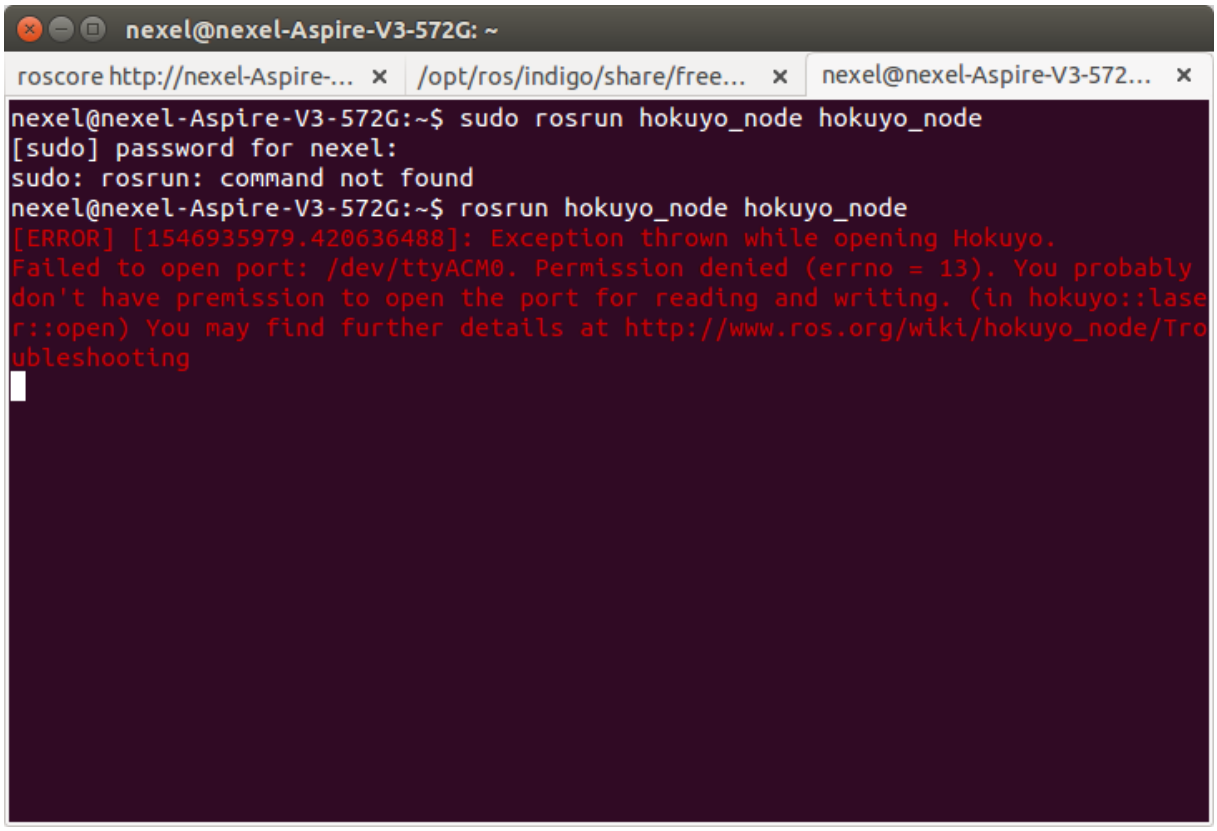


Figura A.3: Fallo láser *hokuyo*.

Esto es posible arreglarlo siguiendo el apartado 0.1 de la página web `http://wiki.ros.org/hokuyo_node/Tutorials/UsingTheHokuyoNode` [47].

Apéndice B

Código C++

En este apéndice incluimos los códigos más relevantes implementados en nuestro TFG. Estos son los de detección y cálculo de la posición del objeto, el MCD, cálculo de la posición del efector final y MCI, basado en los códigos de Matlab del TFM de Marina Aguilar Moreno [18], el cuerpo del aprendizaje y la generación de nuevas simulaciones.

B.1. Detección del objeto

```
/*-------------------------------------
Sergio Gonzalez Muriel
Degree thesis:  Reinforcement learning for object manipulation by a
   robotic arm
Gets the RGB image and processes it in order to detect the object,
   saving a black and white image showing the object in white and the
   rest in black.:
Inputs: image_msg: The image of the RGB camera to process
-------------------------------------*/
void callbackImage(const ImageConstPtr& image_msg){
        try
    {
    cv_ptr = cv_bridge::toCvCopy(image_msg, image_encodings::BGR8);
    }
    catch (cv_bridge::Exception& e)
    {
    ROS_ERROR("cv_bridge exception: %s", e.what());
```

```
    return;
    }

    // Converts image from color to B&W being white the red object

    cv_ptr->image = ~(cv_ptr->image);

    Mat3b imageHSV;
    // If it is not a simulation, applies filter for avoiding
       reflections
    if(!is_simulation){
        GaussianBlur(cv_ptr->image, cv_ptr->image, Size(15,15), 7, 7);
           // Size(9,9), 4, 4
    }
    // Gets the red object
    cvtColor(cv_ptr->image, imageHSV, COLOR_BGR2HSV);
    inRange(imageHSV, Scalar(90 - 10, 100, 100), Scalar(90 + 10, 255,
       255), cv_ptr->image);
    // Applies filter for avoiding false positives
    GaussianBlur(cv_ptr->image, cv_ptr->image, Size(3,3), 3, 3);
}
```

B.2. Cálculo de la posición del objeto

```
/*-----------------------------------
Sergio Gonzalez Muriel
Degree thesis:  Reinforcement learning for object manipulation by a
   robotic arm
Gets the object real position with respect to the sensor frame where:
    max_u: Max X coordinate of the center of the object (Pixels)
    max_v: Max Y coordinate of the center of the object (Pixels)
    min_u: Min X coordinate of the center of the object (Pixels)
    min_v: Min Y coordinate of the center of the object (Pixels)
Outputs: It saves the position of the object in the sensor frame
-----------------------------------*/
void getObjectPosition(){
    if ((max_u >= (cv_ptr->image.cols - 42)) || (min_u <= 42)){
        robot_state.angle_c = -INFINITY;
        robot_state.distance_c = -INFINITY;
        robot_state.height_c = -INFINITY;
        object_center(0) = -INFINITY;
        object_center(1) = -INFINITY;
        ROS_INFO("angle, distance, height: %.10f, %.10f, %.10f",
         robot_state.angle_c, robot_state.distance_c, robot_state.
            height_c);
    }else{
        // Gets the distance of the object
        double f = P(0,0);
        double cx = P(0,2);
        double cy = P(1,2);
        vec3 bitmap_leftmost_pos;
        bitmap_leftmost_pos << min_u << object_center(1) << 1;
        vec4 sensor_leftmost_pos = f * image2sensor(bitmap_leftmost_pos)
           ;

        vec3 bitmap_rightmost_pos;
        bitmap_rightmost_pos << max_u << object_center(1) << 1;
        vec4 sensor_rightmost_pos = f * image2sensor(
           bitmap_rightmost_pos);

        double width = sensor_rightmost_pos(0) - sensor_leftmost_pos(0);
```

```
        dist = ((f * OBJECT_WIDTH) / width) / 1000;
        ROS_INFO("width: %.10f, real distance: %.10f", width, dist);

        // Gets the pixel position in x,y
        vec3 pixel_pos; // 3 x 1
        vec4 result;    // 4 x 1
        pixel_pos(0) = object_center(0);
        pixel_pos(1) = object_center(1);
        pixel_pos(2) = 1;
        result = image2sensor(pixel_pos);
        result /= norm(result);
        robot_state.angle_c = result(0) * dist;
        //It should be -0.12, but as we don't see the entire object we
           have to modify it
        robot_state.height_c = result(1) * dist;
        robot_state.distance_c = result(2) * dist;
        ROS_INFO("\n\nDistance, Angle, height: \n\t(%.10f, %.10f, %.10f)
           \n", robot_state.distance_c, robot_state.angle_c, robot_state
           .height_c);
    }
}
```

B.3. Modelo Cinemático Directo

```
/*-----------------------------------
Sergio Gonzalez Muriel
Degree thesis:  Reinforcement learning for object manipulation by a
   robotic arm
Based on the Matlab code elaborated for the Master Thesis of Marina
   Aguilar Moreno [18]
Gets the direct kinematic model of the Widow-X arm, saving the gripper
   prosition and the transformation matrix T05
-----------------------------------*/
void mcd(){
    vec5 alpha;
    alpha << 0 << -M_PI/2 << M_PI << 0 << M_PI/2;
        vec5 a;
    a << 0 << 0 << d2 << L3 << 0;
        vec5 d = arma::zeros<vec>(5);
        vec5 q;
    q << joint_angles(0)
      << joint_angles(1)-beta
      << joint_angles(2)-beta
      << joint_angles(3)+(M_PI/2)
      << joint_angles(4);
        int i  = 0;
        mat44 T01;
    T01 << cos(q(i)) << -sin(q(i)) << 0 << a(i) << endr
                << sin(q(i))*cos(alpha(i)) << cos(q(i))*cos(alpha(i)) <<
                    -0 << -0*d(i) << endr
                << sin(q(i))*0 << cos(q(i))*0 << cos(alpha(i)) << cos(
                   alpha(i))*d(i) << endr
                << 0 << 0 << 0 << 1 << endr;

        i  = 1;
        mat44 T12;
        T12     << cos(q(i)) << -sin(q(i)) << 0 << a(i) << endr
                << sin(q(i))*0 << cos(q(i))*0 << -sin(alpha(i)) << -sin(
                   alpha(i))*d(i) << endr
                << sin(q(i))*sin(alpha(i)) << cos(q(i))*sin(alpha(i)) <<
                    0 << 0*d(i) << endr
```

```
                << 0 << 0 << 0 << 1 << endr;

        i  = 2;
        mat44 T23;
        T23 << cos(q(i)) << -sin(q(i)) << 0 << a(i) << endr
                << sin(q(i))*cos(alpha(i)) << cos(q(i))*cos(alpha(i)) <<
                    -0 << -0*d(i) << endr
                << sin(q(i))*0 << cos(q(i))*0 << cos(alpha(i)) << cos(
                   alpha(i))*d(i) << endr
                << 0 << 0 << 0 << 1 << endr;

        i  = 3;
        mat44 T34;
        T34     << cos(q(i)) << -sin(q(i)) << 0 << a(i) << endr
                << sin(q(i))*cos(alpha(i)) << cos(q(i))*cos(alpha(i)) <<
                    -0 << -0*d(i) << endr
                << sin(q(i))*0 << cos(q(i))*0 << cos(alpha(i)) << cos(
                   alpha(i))*d(i) << endr
                << 0 << 0 << 0 << 1 << endr;

        i  = 4;
        mat44 T45;
        T45     << cos(q(i)) << -sin(q(i)) << 0 << a(i) << endr
                << sin(q(i))*0 << cos(q(i))*0 << -sin(alpha(i)) << -sin(
                   alpha(i))*d(i) << endr
                << sin(q(i))*sin(alpha(i)) << cos(q(i))*sin(alpha(i)) <<
                    0 << 0*d(i) << endr
                << 0 << 0 << 0 << 1 << endr;

        T05 = T01 * T12 * T23 * T34 * T45;
    getGripperPosition();
}
```

B.4. Cálculo de la posición del efector final

```
/*-----------------------------------
Sergio Gonzalez Muriel
Degree thesis:  Reinforcement learning for object manipulation by a
   robotic arm
Based on the Matlab code elaborated for the Master Thesis of Marina
   Aguilar Moreno [18]
Gets the position of the gripper by means of the direct kinematic model:
 -----------------------------------*/
void getGripperPosition(){
    double ax = T05(0,2);
        double ay = T05(1,2);
        double az = T05(2,2);
        double px = T05(0,3);
        double py = T05(1,3);
        double pz = T05(2,3);

        gripper_position(0) = px + L45*ax;
        gripper_position(1) = py + L45*ay;
        gripper_position(2) = pz + L45*az;

    ang_or(0) = ax;
    ang_or(1) = ay;
    ang_or(2) = az;
}
```

B.5. Modelo Cinemático Inverso

```
/*-----------------------------------
Sergio Gonzalez Muriel
Degree thesis:  Reinforcement learning for object manipulation by a
    robotic arm
Based on the Matlab code elaborated for the Master Thesis of Marina
    Aguilar Moreno [18]
Gets the angle of each joint in order to reach the desired position
by means of the inverse kinematic model, sending a message to the servos
     with the angle to reach and updating the mcd and sensor data:
Inputs:
    next_position: Desired position
 -----------------------------------*/
void mci(vec3 next_position){
        double px = next_position(0) - L45*ang_or(0);
        double py = next_position(1) - L45*ang_or(1);
        double pz = next_position(2) - L45*ang_or(2);

    ROS_INFO("px: %.10f, py: %.10f, pz: %.10f", px, py, pz);
    ROS_INFO("ang_or: [%.10f %.10f %.10f]", ang_or(0), ang_or(1), ang_or
       (2));
    cout << T05;

        double q1 = atan2(py, px);

        double k = pow(pz, 2) + pow(d2, 2) + pow(((px * cos(q1)) + (py *
            sin(q1))), 2) - pow(L3, 2);
        double k1 = 2 * d2 * px * cos(q1) + 2 * py * d2 * sin(q1);
        double k2 = 2 * pz * d2;

        double theta2b = atan2(k1, k2) - atan2(k, -sqrt(pow(k1,2)+pow(k2
           ,2)-pow(k,2)));
        double q2 = theta2b + beta;

        double theta23 = asin((-pz - d2*sin(theta2b))/L3);
        double q3 = q2 - theta23;
```

```
        double L = ang_or(2)*cos(q2-q3) + ang_or(0)*sin(q2-q3)*cos(q1) +
            ang_or(1)*sin(q2-q3)*sin(q1);
        double q4 = acos(-L) - (M_PI/2);

        double q5 = asin(n(0)*sin(q1) - n(1)*cos(q1));

    ROS_INFO("\n\nk: %.2f, k1: %.2f, k2: %.2f, theta2b: %.2f, theta23:
       %.2f, L: %.2f\n", k, k1, k2, theta2b, theta23, L);
    ROS_INFO("\n\nq1: %.2f\nq2: %.2f\nq3: %.2f\nq4: %.2f\nq5: %.2f\n",
       q1,q2,q3,q4,q5);

    if(!(isnan(q1) || isnan(q2) || isnan(q3) || isnan(q4) || isnan(q5)))
       {
        Float64 angle;
        angle.data = q5;
        joints[4].publish(angle);
        angle.data = q4;
        joints[3].publish(angle);
        angle.data = q3;
        joints[2].publish(angle);
        angle.data = q2;
        joints[1].publish(angle);
        angle.data = q1;
        joints[0].publish(angle);
        joint_angles(0) = q1;
        joint_angles(1) = q2;
        joint_angles(2) = q3;
        joint_angles(3) = q4;
        joint_angles(4) = q5;
        robot_state.folded = (next_position(0) == 0
                              and next_position(1) == 0
                              and next_position(2) == 0);
    }else{
        ROS_INFO("Object not reachable.");
    }

    processMessages();
    mcd();
}
```

B.6. Cuerpo del aprendizaje

```
/*
Sergio Gonzalez Muriel
Degree thesis:  Reinforcement learning for object manipulation by a
   robotic arm
Q-Learning implementation
*/

#include "utils.cpp"
/*-------------------------------------
 Methods
 -------------------------------------*/
int main(int argc, char** argv){
    ros::init(argc, argv, "learning_node");

    // We check if it is a gazebo simulation

    Handlers handlers;

    learning(handlers);
    return 0;
}

/*-------------------------------------
 Makes all the processes of learning:
 -------------------------------------*/
void learning(Handlers handlers){
    initializeVecMat();
    readLog();
    cout << "Do you want just to exploit?[y|N] ";
    getline(cin, exploit);
    string gui_response;
    cout << "Do you want to watch the simulation?[y|N] ";
    getline(cin, gui_response);
    if(gui_response == "y"){
        gui = true;
    }
```

```
bool end_simulation = false;

while(ros::ok() && !end_simulation){
    // Sets random seed by the time of the cpu
    srand( (unsigned)time(NULL) );
    startRandomEpisode();
    sleep(3);
    if (gui){
        namedWindow("Red objects image",CV_WINDOW_AUTOSIZE);
    }

    // Creates timers
    double time0 = ros::Time::now().toSec();
    while(time0 == 0){
        time0 = ros::Time::now().toSec();
    }

    robot_state.object_picked = false;
    robot_state.folded = false;

    // Initializes all publishers and subscribers
    ros::master::getTopics(topic_info);
    isSimulation();
    color_image_sub = handlers.getIT().subscribe("/camera/rgb/
       image_color", 1, &callbackImage);
    camera_info_sub = handlers.getNH().subscribe("/camera/rgb/
       camera_info", 1, &callbackCameraInfo);
    joint_states_sub = handlers.getNH().subscribe("/joint_states",
       1, &getGripperEffortCallback);
    sim_pose_sub = handlers.getNH().subscribe("/simulation/pose", 1
       , &getSimulationPoseCallback);

    joints[0] = handlers.getNH().advertise<Float64>("/arm_1_joint/
       command", 1);
    joints[1] = handlers.getNH().advertise<Float64>("/arm_2_joint/
       command", 1);
    joints[2] = handlers.getNH().advertise<Float64>("/arm_3_joint/
       command", 1);
```

```
joints[3] = handlers.getNH().advertise<Float64>("/arm_4_joint/
    command", 1);
joints[4] = handlers.getNH().advertise<Float64>("/arm_5_joint/
    command", 1);

gripper = handlers.getNH().advertise<Float64>("/gripper_1_joint/
    command", 1);
base = handlers.getNH().advertise<Twist>("/mobile_base/commands/
    velocity", 1);

steps = 0;
prev_V = V;
robot_state.angle_d = 0;
robot_state.height_d = 0;
robot_state.distance_d = 0;
robot_state.angle_c = 0;
robot_state.height_c = 0;
robot_state.distance_c = 0;
bool first_step = true;


bool end_episode = false;
while(ros::ok() && !end_episode){
    update_pose = true;
    // Updates state
    processMessages();
    updateState();

    // If it's the first time, sets the arm to the initial
       position
    if (first_step){
        openGripper();
        foldArm();
        // 1. Get state
        updateState();
        sa = getIndexFromState();
        first_step = false;
    }
```

```
// 2. Select action
selectAction();

// 3.1 Move arm if reachable
if(action == 4){
    if(object_reachable){
        ROS_INFO("Moving arm...");
        moveArmToObject();
        if(exploit != "y"){
            end_episode = true;
        }
        robot_state.object_picked = true;
    }else{
        ROS_INFO("Trying to move arm but object is not
           reachable...");
    }
}
// 3.2 Move base if not reachable
else{
    if(robot_state.folded == 0){
        foldArm();
        ros::Duration(3).sleep();
    }
    Twist base_movement;
    // Move front
    if (action == 2){
        ROS_INFO("Moving front...");
        base_movement.linear.x = 0.1;
    }
    // Move back
    else if(action == 3){
        ROS_INFO("Moving back...");
        base_movement.linear.x = -0.1;
    }
    // Turn left
    else if(action == 0){
        ROS_INFO("Turning left...");
        base_movement.angular.z = 0.1;
    }
```

```
// Turn right
else if(action == 1){
    ROS_INFO("Turning right...");
    base_movement.angular.z = -0.1;
}
if(robot_state.distance_d != 0 && robot_state.angle_d !=
    0 && robot_state.height_d != 0){
    int prev_dist = robot_state.distance_d;
    int prev_ang = robot_state.angle_d;
    bool changed_state = false;
    int counter = 0;
    while(!changed_state && counter < 7){
        base.publish(base_movement);
        ros::Duration(3).sleep();
        processMessages();
        updateState();
        if(action == 0 || action == 1){
            changed_state = (prev_ang != robot_state.
                angle_d);
        }else{
            changed_state = (prev_dist != robot_state.
                distance_d);
        }
        counter++;
    }
}else{
    double necessary_time;
    if(action == 0 || action == 1){
        necessary_time = (double)angle_per_level/(double
            )abs(base_movement.angular.z);
    }else{
        necessary_time = (double)distance_per_level/(
            double)abs(base_movement.linear.x);
    }
    double diff_time = 0;
    bool first_loop = true;
    while((diff_time < necessary_time) && ros::ok()){
        if(first_loop){
            time0 = ros::Time::now().toSec();
```

```
                first_loop = false;
            }
            base.publish(base_movement);
            diff_time = ros::Time::now().toSec() - time0;
        }
    }
}
ros::Duration(3).sleep();
// Updates state
processMessages();
updateState();

sp = getIndexFromState();

if (exploit != "y"){

    // 5. Check reward
    calculateReward();

    // Updates visit matrix
    visit_matrix(sa, action)++;

    // Updates Q-matrix
    alpha = 1/(pow(visit_matrix(sa,action),0.5));
    q_matrix(sa,action) = (1 - alpha) * q_matrix(sa,action)
       + alpha * (reward + GAMMA * V(sp));


    // Updates V and policy matrices
    vec prev_V_it = V;      // Prev V function on each
       iteration
    updateVPolicy();
    d = norm(V - prev_V_it);
    e = arma::min(arma::abs(V - prev_V_it));
    actualizeIterationDistanceLog();

    steps++;
    actualizeLog();
    actualizeSimplifiedLog();
```

```
                if(steps == 200){
                    end_episode = true;
                }
            }else{
                actualizeExploitationLog();
            }
            sa = sp;
        }
        if (gui){
            destroyWindow("Red objects image");
        }
        d = norm(V - prev_V);
        e = arma::min(arma::abs(V - prev_V));
        actualizedistanceLog();
        killEpisode();
        if(d < 1){
            counter_continuous_low_distance++;
        }else{
            counter_continuous_low_distance = 0;
        }
        if(d < 0 || e < 0 || counter_continuous_low_distance == 10){
            end_simulation = true;
            ROS_INFO("The robot has already learn. End of simulation..."
               );
        }
    }
}
```

B.7. Generación de nuevos episodios

```
/*-----------------------------------
Sergio Gonzalez Muriel
Degree thesis:  Reinforcement learning for object manipulation by a
   robotic arm
Starts a new random episode:
-----------------------------------*/
void startRandomEpisode(){
    int status;
        // Opens an empty world in gazebo
        if (gui){
            status = system("xterm -hold -e \"roslaunch gazebo_ros
               empty_world.launch paused:=true\" &");
        }else{
            status = system("xterm -hold -e \"roslaunch gazebo_ros
               empty_world.launch paused:=true gui:=false\" &");
        }
        if (status == 0){
            sleep(6);
            double x = MIN_X + ((double)rand()/double(RAND_MAX))* (MAX_X
               -MIN_X);
            double y = MIN_Y + ((double)rand()/double(RAND_MAX))* (MAX_Y
               -MIN_Y);
            int box = MIN_BOX + ((double)rand()/double(RAND_MAX))* (
               MAX_BOX-MIN_BOX);
            double z = 0.05*(double)box;
            stringstream xterm_box; stringstream xterm_object;
            xterm_box << "xterm +hold -e \"rosrun gazebo_ros spawn_model
                -file $(rospack find learning)/urdf/box_" << box << ".
               urdf -urdf -x " << x
                    << " -z " << z << " -y " << y << " -model box\" &";
            xterm_object << "xterm +hold -e \"rosrun gazebo_ros
               spawn_model -file $(rospack find learning)/urdf/cylinder.
               urdf -urdf -x "
                        << (x - 0.45) << " -z " << (z*2+0.05) << " -y "
                           << y << " -model red_object\" &";
            string str(xterm_box.str());
            const char* xterm_box_str = str.c_str();
```

```
system(xterm_box_str);
sleep(3);
str = xterm_object.str();
const char* xterm_object_str = str.c_str();
system(xterm_object_str);
sleep(3);
stringstream xterm_wall;
xterm_wall << "xterm +hold -e \"rosrun gazebo_ros
   spawn_model -file $(rospack find learning)/urdf/wall.urdf
    -urdf -x "
            << (x - 0.5) << " -y " << (y+3) << " -model wall
               \" &";
str = xterm_wall.str();
const char* xterm_wall_str = str.c_str();
system(xterm_wall_str);
sleep(3);
stringstream xterm_wall1;
xterm_wall1 << "xterm +hold -e \"rosrun gazebo_ros
   spawn_model -file $(rospack find learning)/urdf/wall.urdf
    -urdf -x "
            << (x - 0.5) << " -y " << (y-3) << " -model
               wall1\" &";
str = xterm_wall1.str();
xterm_wall_str = str.c_str();
system(xterm_wall_str);
sleep(3);
stringstream xterm_wall2;
xterm_wall2 << "xterm +hold -e \"rosrun gazebo_ros
   spawn_model -file $(rospack find learning)/urdf/wall2.
   urdf -urdf -x "
            << (- 0.5) << " -y " << y << " -model wall2\" &"
               ;
str = xterm_wall2.str();
xterm_wall_str = str.c_str();
system(xterm_wall_str);
sleep(3);
stringstream xterm_wall3;
xterm_wall3 << "xterm +hold -e \"rosrun gazebo_ros
   spawn_model -file $(rospack find learning)/urdf/wall2.
```

```
            urdf -urdf -x "
                    << (x/2- 0.5) << " -y " << y+5.55 << " -Y " <<
                       M_PI/2 << " -model wall3\" &";
        str = xterm_wall3.str();
        xterm_wall_str = str.c_str();
        system(xterm_wall_str);
        sleep(3);
        stringstream xterm_wall4;
        xterm_wall4 << "xterm +hold -e \"rosrun gazebo_ros
            spawn_model -file $(rospack find learning)/urdf/wall2.
            urdf -urdf -x "
                    << (x/2- 0.5) << " -y " << y-5.55 << " -Y " <<
                       M_PI/2 << " -model wall4\" &";
        str = xterm_wall4.str();
        xterm_wall_str = str.c_str();
        system(xterm_wall_str);
        sleep(3);
        // Instantiates a turtlebot in that empty world
        system("xterm -hold -e \"roslaunch crumb_gazebo test.launch
            \" &");
        sleep(10);
        // Unpauses simulation
        system("rosservice call /gazebo/unpause_physics");
        sleep(5);
        episodes++;
        actualizeObjectPositionLog((x - 0.45), y, (z*2+0.05));
    }else{
        system("killall -9 xterm gzserver");
        ros::shutdown();
    }
}
```

Bibliografía

[1] Marco Wiering y Martijn Van Otterlo. «Reinforcement learning». En: *Adaptation, learning, and optimization* 12 (2012), pág. 3.

[2] Richard J. Boucherie y Nico M. Van Dijk. *Markov decision processes in practice*. Vol. 248. Springer, 2017.

[3] Jens Kober, J. Andrew Bagnell y Jan Peters. «Reinforcement learning in robotics: A survey». En: *The International Journal of Robotics Research* 32.11 (2013), págs. 1238-1274.

[4] Angel Martínez-Tenor y col. «Towards a common implementation of reinforcement learning for multiple robotic tasks». En: *Expert Systems with Applications* 100 (2018), págs. 246-259.

[5] Ziad Salloum. *A top view of how Model Based Reinforcement Learning works*. URL: `https://towardsdatascience.com/model-based-reinforcement-learning-cb9e41ff1f0d` (visitado 30-05-2019).

[6] Ángel Martínez-Tenor. «Reinforcement Learning on the Lego Mindstorms NXT Robot. Analysis and Implementation». Tesis de lic. Universidad de Málaga, mayo de 2013.

[7] *CRUMB project*. Departamento de Ingeniería de Sistemas y Automática. UMA. URL: `https://babel.isa.uma.es/crumb/` (visitado 31-05-2019).

[8] *CRUMBproject/V-REP*. CRUMB project. URL: `https://github.com/CRUMBproject/V-REP.git` (visitado 16-01-2019).

[9] *Turtlebot2*. Open Source Robotics Foundation, Inc. URL: `https://www.turtlebot.com/` (visitado 31-05-2019).

[10] *Brazo manipulador Widow-X*. Trossen Robotics. URL: `https://www.trossenrobotics.com/widowxrobotarm` (visitado 31-05-2019).

[11] *es - ROS Wiki*. Open Source Robotics Foundation. URL: http://wiki.ros.org/es (visitado 29-05-2019).

[12] *OpenCV*. OpenCV team. URL: https://opencv.org/ (visitado 29-05-2019).

[13] *Turtlebot*. Open Source Robotics Foundation, Inc. URL: https://www.turtlebot.com/turtlebot2/ (visitado 12-08-2019).

[14] *Kinect*. Wikipedia. URL: https://es.wikipedia.org/wiki/Kinect (visitado 12-08-2019).

[15] *Hokuyo URG-04LX-UG01*. ROS Components. URL: https://www.roscomponents.com/es/lidar-escaner-laser/83-urg-04lx-ug01.html (visitado 23-07-2019).

[16] *Sensores kobuki Turtlebot 2*. Yujin Robot. URL: http://www.robotnik.es/web/wp-content/uploads/2014/04/TB_robot.pdf (visitado 31-05-2019).

[17] *Especificaciones CRUMB*. Departamento de Ingeniería de Sistemas y Automática. UMA. URL: https://babel.isa.uma.es/crumb/index.php/sample-page/ (visitado 31-05-2019).

[18] Marina Aguilar Moreno. «Modelado y simulación de un robot TurtleBot 2 con brazo manipulador WidowX sobre ROS y Gazebo». Tesis de lic. Universidad de Málaga, dic. de 2016.

[19] *Setting DYNAMIXEL IDs with the DynaManager*. Trossen Robotics. URL: https://learn.trossenrobotics.com/arbotix/1-using-the-tr-dynamixel-servo-tool\#\&panel1-1 (visitado 16-01-2019).

[20] *Brazo manipulador Widow-X. Características*. Trossen Robotics. URL: https://www.trossenrobotics.com/shared/productdocs/widowXdraft_07.pdf (visitado 31-05-2019).

[21] *ROS answers*. Open Source Robotics Foundation. URL: https://answers.ros.org/questions/ (visitado 30-05-2019).

[22] *ROS blog*. Open Source Robotics Foundation. URL: http://www.ros.org/news/ (visitado 30-05-2019).

[23] *ROS tutorials*. Open Source Robotics Foundation. URL: http://wiki.ros.org/ROS/Tutorials (visitado 30-05-2019).

[24] *Repositorio GitHub*. GitHub, Inc. URL: https://github.com/ (visitado 30-05-2019).

[25] *Gazebo: Robot simulation made easy.* Open Source Robotics Foundation. URL: http://gazebosim.org/ (visitado 12-08-2019).

[26] Morgan Quigley, Brian Gerkey y William D. Smart. *Programming Robots with ROS: a practical introduction to the Robot Operating System*. O'Reilly Media, Inc., 2015.

[27] *Previous Versions of V-REP*. Coppelia robotics. URL: www.coppeliarobotics.com/previousversions.html (visitado 16-01-2019).

[28] *OpenCV manual. Introduction*. OpenCV team. URL: https://docs.opencv.org/4.1.0/d1/dfb/intro.html (visitado 29-05-2019).

[29] *VREP downloads*. Coppelia Robotics. URL: http://www.coppeliarobotics.com/downloads.html (visitado 31-05-2019).

[30] *VREP RosInterfaces*. Coppelia Robotics. URL: http://www.coppeliarobotics.com/helpFiles/en/rosInterfaces.htm (visitado 31-05-2019).

[31] *ROS Melodic Installation*. Open Source Robotics Foundation. URL: http://wiki.ros.org/melodic/Installation (visitado 31-05-2019).

[32] ully Foote ¡tfoote at openrobotics.org¿ y Ken Conley ¡kwc at willowgarage.com¿. *ROS Target Platforms*. URL: http://www.ros.org/reps/rep-0003.html (visitado 31-05-2019).

[33] *HSL and HSV*. Wikipedia. URL: https://en.wikipedia.org/wiki/HSL_and_HSV (visitado 16-01-2019).

[34] *Kinect Depth Camera info*. MediaWiki. URL: http://www.fishcamp.com/pdf/mt9m001_1300_mono.pdf (visitado 12-08-2019).

[35] *Kinect RGB Camera info*. MediaWiki. URL: http://devel.0cpm.org/reverse/grandstream/datasheet/MT9V112.pdf (visitado 12-08-2019).

[36] John J. Craig. *Introduction to robotics: mechanics and control, 3/E*. Pearson Education India, 2009.

[37] *Ubuntu 14.04.5 LTS (Trusty Tahr)*. Ubuntu. URL: http://releases.ubuntu.com/14.04/ (visitado 16-01-2019).

[38] Pete Batard. *Rufus. Cree unidades USB arrancables fácilmente*. URL: https://rufus.ie/ (visitado 16-01-2019).

[39] *CMake*. Kitware. URL: https://cmake.org/ (visitado 16-01-2019).

[40] ellen. *CMake Error: Could not find CMAKE_ROOT?* URL: https://askubuntu.com/questions/1014670/cmake-error-could-not-find-cmake-root (visitado 16-01-2019).

[41] CesarHoyosM. *Ubuntu install of ROS Indigo*. URL: http://wiki.ros.org/indigo/Installation/Ubuntu (visitado 16-01-2019).

[42] Marguedas. *Installing and Configuring Your ROS Environment*. URL: http://wiki.ros.org/ROS/Tutorials/InstallingandConfiguringROSEnvironment (visitado 16-01-2019).

[43] Conrad Sanderson y Ryan Curtin. *Armadillo*. URL: http://arma.sourceforge.net/ (visitado 16-01-2019).

[44] Enrique Coronado. *Instalar Armadillo (biblioteca de álgebra lineal) en Visual Studio 2012*. URL: https://mecatronicauaslp.wordpress.com/2013/12/17/instalar-armadillo-biblioteca-de-algebra-lineal-en-visual-studio-2012-x86/ (visitado 16-01-2019).

[45] Stefano Primatesta. *Problem with Indigo and Gazebo 2.2*. URL: https://answers.ros.org/question/199401/problem-with-indigo-and-gazebo-22/ (visitado 16-01-2019).

[46] *ArbotiX-M Robocontroller Getting Started Guide*. Trossen Robotics. URL: https://learn.trossenrobotics.com/arbotix/arbotix-quick-start.html (visitado 16-01-2019).

[47] Isaac Saito. *How to use Hokuyo Laser Scanners with the hokuyo_node*. URL: http://wiki.ros.org/hokuyo_node/Tutorials/UsingTheHokuyoNode (visitado 16-01-2019).