UNIVERSIDAD DE MÁLAGA

E.T.S. INGENIERÍA INFORMÁTICA
UNIVERSIDAD DE MÁLAGA

Ingeniería Informática

Acoplamiento autónomo de un robot móvil en su base de carga mediante la detección de códigos QR

Autonomous docking of a mobile robot in its charging station by the detection of QR codes

Realizado por
Enrique Ramírez García

Tutorizado por
Antonio Javier González Jiménez
José Raúl Ruiz Sarmiento

Departamento
Ingeniería de Sistemas y Automática
Universidad de Málaga

MÁLAGA, SEPTIEMBRE, 2021

ESCUELA TÉCNICA SUPERIOR DE INGENIERÍA INFORMÁTICA

GRADO EN INGENIERÍA INFORMÁTICA

Acoplamiento autónomo de un robot móvil en su base de carga mediante la detección de códigos QR

Autonomous docking of a mobile robot in its charging station by the detection of QR codes

Realizado por:

Enrique Ramírez García

Tutorizado por:

Antonio Javier González Jiménez, José Raúl Ruiz Sarmiento

MÁLAGA, DICIEMBRE, 2021

Resumen

El objetivo principal que persigue este TFG es la implementación de un programa que permita a un robot realizar un acoplamiento automático sobre una base de carga mediante el uso de códigos QR como patrones de reconocimiento. La automatización de este proceso de acoplamiento tiene como fin último que el robot sea capaz de realizar una recarga de su batería para prevenir situaciones en la que el nivel de la misma sea tan bajo que impida al robot realizar cualquier otra tarea. Para poder realizar este acoplamiento automático, se ha implementado un algoritmo capaz de localizar la estación de carga a partir de un código QR que la identifica. Una vez detectada, y con la premisa de que el código QR se encuentra en una superficie plana, se procede a realizar el cálculo de la homografía existente entre este plano y una visión sin distorsión de perspectiva del mismo, que nos permite hallar la matriz de transformación de los puntos existentes en un plano a su análogo en otra imagen. Gracias al cálculo de esta matriz, obtenemos la rotación y traslación existentes entre estas imágenes, a partir de la cuales, el robot recibe los comandos correspondientes para poder realizar el acoplamiento en la estación de carga correctamente. La técnica desarrollada ha sido evaluada con éxito empleando la arquitectura de control robótica *Robot Operating System* (ROS) y el popular simulador *Gazebo*.

Palabras claves: Acoplamiento automático, códigos QR, homografía, detección de patrones, robots móviles, ROS.

Abstract

The main goal of this Undergraduate Thesis is to design a tool that allows a robot to perform an autonomous docking on its charging station by using QR codes as recognition patterns. The ultimate aim of the automation of this docking process is to make the robot be able to recharge its battery in situations such as an extremely low battery level, which prevents the robot from performing any other task. In order to perform this autonomous docking, an algorithm has been implemented to locate the charging station based on a QR code that identifies it. Once detected, and with the premise that the QR code is on a flat surface, we proceed to calculate the homography between this plane and the same plane without any kind of perspective distortion. This allows us to obtain the transformation matrix of the existing points on a plane and the correspondent ones belonging to the same plane in another image. Thanks to the calculation of this matrix, we can obtain the existing rotation and translation between these images, from which the robot receives the corresponding commands to correctly perform the docking on the charging station. The developed technique has been successfully tested using the Robot Operating System (ROS) and the well-known Gazebo simulator.

Keywords: Autonomous docking, QR codes, homography, pattern detection, mobile robots, ROS.

Índice general

Capítulo 1

Introducción

El gran auge de los robots en estos tiempos ha propiciado su utilización en un contexto diario. Debido a ello, y puesto que la mayoría de estos robots funcionan con baterías recargables mediante estaciones de carga, una de las preocupaciones que surgen al utilizarlos es la de controlar el nivel de batería de estos. Este TFG pretende dotar a un robot con la capacidad de realizar un acoplamiento en su base de carga con el fin de que sea capaz de realizar la posterior recarga de forma autónoma.

A continuación, se presentará una introducción con las razones que han motivado a la realización de este proyecto, así como los objetivos a completar con la finalización del mismo.

1.1. Contexto y motivación

Como se ha mencionado anteriormente, en los tiempos que corren, la utilización de robots en el día a día es cada vez más frecuente. La robótica se ha adentrado en aspectos de nuestra vida cotidiana tales como en la educación, teniendo como ejemplo la robótica educativa [28], en la sanidad, como los robots asistentes que ayudan al mantenimiento de la salud [8], o en las tareas domésticas o del hogar, como por ejemplo, los famosos robots aspiradores *Roomba* [29].

Siguiendo con estos últimos, la finalidad de estos robots consiste en facilitar a sus dueños la realización de tareas tediosas y repetitivas con el objetivo de que estos puedan dedicar su tiempo a asuntos más sustanciales. Lógicamente, y dado que la mayoría de estos robots basan su funcionamiento en el uso de baterías, para que estos robots ejecuten estas tareas correctamente, deben precisar de un nivel de batería adecuado para ello. Es aquí donde surge la necesidad de poder controlar los niveles de batería de estos dispositivos, donde habitualmente, se hace uso de una estación o base de carga en la que el robot se estaciona para iniciar la recarga de la batería.

Con el objetivo de que el robot realice esta tarea de manera automática, se deberá asignar un criterio para que el robot sea capaz de discernir cuando es el momento adecuado en el que deberá proceder a realizar esta recarga. Algunos criterios que se pueden establecer para realizar esta recarga podrían ser que la batería del robot se encuentre en un nivel extremadamente bajo o que la batería estuviera en un nivel insuficiente que le permita realizar satisfactoriamente sus tareas asignadas.

Teniendo esto en cuenta, este proyecto se centrará en la automatización del proceso de estacionamiento del robot en la base de carga para su posterior recarga. Para dotar al robot de la capacidad para realizar esta tarea, se hace necesaria la implementación/adaptación de un algoritmo que sea capaz de localizar la base de carga. Con el fin de que el robot sea capaz de distinguir la base de carga del resto de elementos del entorno, haremos uso de los códigos QR como patrones de detección. Asimismo, el robot deberá estar dotado de una cámara RGB para poder realizar esta detección.

En caso de que el robot no sea capaz de realizar la detección del patrón, pudiendo ser esto debido a la distancia a la que se encuentra del mismo, el robot navegará hacia una zona aproximada en la que pueda realizar la detección.

Una vez la detección ya esté completada, y con el objetivo de ser capaces de comandar al robot hasta su estación de carga, haremos uso de las **homografías**[1], de las que hablaremos en mayor profundidad en el apartado 3.1.2. De manera resumida, la homografía es una herramienta que nos proporciona la **Visión por Computador** y que nos permitirá obtener la correspondencia entre los puntos de dos imágenes en diferentes planos a partir de una matriz de transformación. En nuestro caso, una imagen vendrá dada por la posición actual del robot al localizar el QR, y la otra vendrá dada por la posición del robot estacionado en la base de carga al localizar el QR. Esta matriz de transformación se compone de una rotación, una traslación y un vector normal al plano. Estas componentes serán las que nos permitirán implementar la navegación en nuestro robot, enviando los comandos de movimiento a partir de estas.

Mediante estos comandos, el robot realizará el acoplamiento en la estación de carga. Un ejemplo de acoplamiento automático lo podemos ver en la figura 1.1.

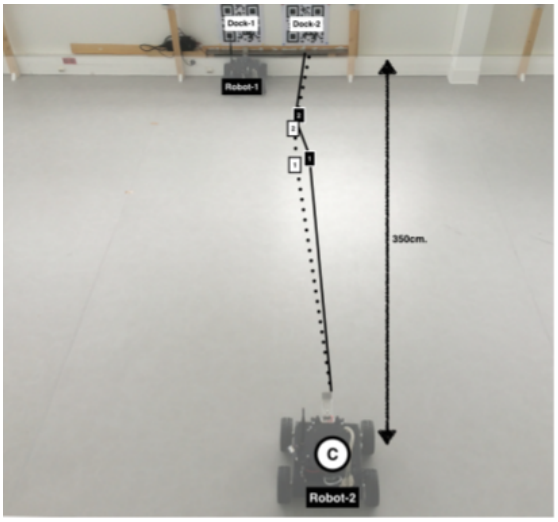


Figura 1.1: Ejemplo [20] de acoplamiento automático realizado por un robot móvil. El robot detecta el código QR y procede moverse hacia la localización del mismo para realizar el acoplamiento.

[1] `https://en.wikipedia.org/w/index.php?title=Homography_(computer_vision)&oldid=1017381135`

1.2. Objetivos

Objetivo principal

El objetivo principal de este proyecto consiste en el desarrollo e implementación de los algoritmos necesarios para que el robot obtenga la capacidad de realizar un acoplamiento automático y, consecuentemente, realizar la recarga de su batería. Para ello, se deberán completar los siguientes **objetivos específicos**.

Objetivos específicos

- **Implementación/Adaptación de algoritmo de detección:** Implementar o adaptar un algoritmo que permita realizar la correcta detección de los patrones usados en este proyectos, es decir, los códigos QR. Para ello, se hará uso de librerías como OpenCV o zbar.

- **Cálculo de la homografía H y sus parámetros:** Desarrollar un programa que sea capaz de realizar el cálculo de la homografía H entre la imagen del robot en su posición actual al detectar el QR y la imagen del robot al realizar el acoplamiento, con el QR detectado. A partir de esta homografía, podremos hallar la matriz de transformación, obteniendo la correspondiente rotación, traslación y vector normal perpendicular al plano. Estos parámetros nos servirán para comandar al robot hacia su base de carga.

- **Navegación del robot hasta la base de carga:** Implementar un algoritmo que, en función de la detección del patrón y la homografía hallada previamente, comande al robot hasta su base de carga para que se realice el acoplamiento.

1.3. Metodología de trabajo

La metodología a seguir para el desarrollo de este proyecto esta basada en varias etapas en las que se incrementará secuencialmente las funcionalidades de los programas implementados. A esta metodología se le conoce como **metodología incremental** [10].

Durante cada etapa se han realizado las siguientes tareas (Figura 1.2):

1. **Estudio del tema**
2. **Reunión con los tutores**
3. **Planificación**
4. **Desarrollo, implementación y pruebas**

En primera instancia, se habló de la idea general del trabajo, sus objetivos y debatir sobre el software a utilizar. En un primer momento, se optó por hacer uso de Unity para realizar las pruebas una vez desarrollado el programa. Pero dado que Unity hace uso de C#, y la librería base de nuestro proyecto, OpenCV, está disponible en C++ y Python, sumado a que Unity se usa fundamentalmente como motor de videojuegos, se planteó el cambio a un entorno que estuviera contemplado para la simulación de robots. Por ello, se escogió ROS [18] para el proceso de simulación del robot, ya que es una herramienta ampliamente usada en este contexto.

También cabe destacar que, si bien existen varios wrappers de OpenCV en C#, muchas de las funciones más importantes que se han usado no llegan a estar plenamente implementadas, a lo que sumándole lo anterior, se optó por dejar de lado Unity y C# y usar ROS con Python como lenguaje.

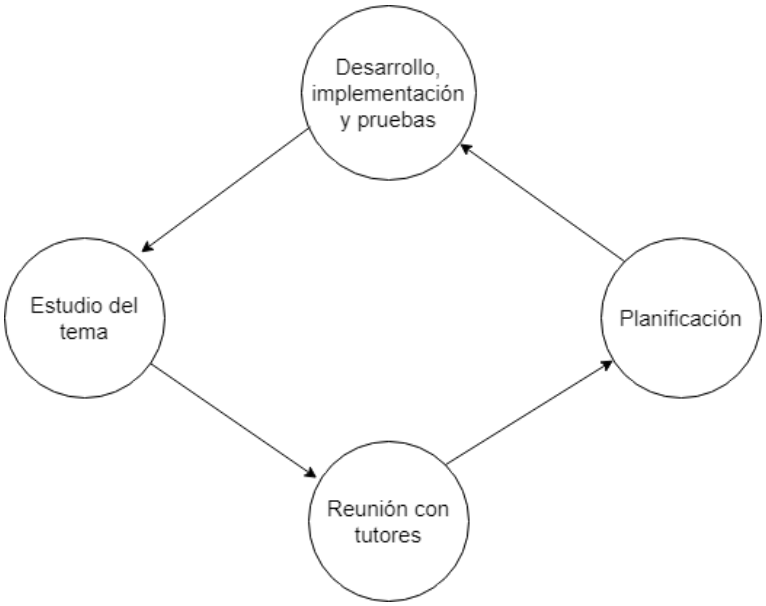


Figura 1.2: Diagrama de la metodología utilizada en el proyecto. En cada iteración se incrementan las funcionalidades de la aplicación siguientes los pasos establecidos.

1.4. Estructura de la memoria

En esta sección, haremos una breve descripción de la estructura que sigue el documento. Este documento se encuentra dividido en capítulos en los que se detallan el proceso de desarrollo del proyecto.

Capítulo 1: En este capítulo se ofrece una introducción al tema tratado en este proyecto, exponiendo el contexto y la motivación para llevarlo a cabo, así como los objetivos que se deben completar.

Capítulo 2: En este capítulo se pone en contexto al lector sobre el estado del arte en la detección de patrones y el acoplamiento autónomo en bases de carga.

Capítulo 3: En este capítulo se introducen una serie de fundamentos teóricos necesarios para comprender como se lleva a cabo la implementación del proyecto.

Capítulo 4: En este capítulo, se desarrolla toda la implementación realizada,

describiendo cada una las funcionalidades desarrolladas en este proyecto, así como una breve descripción de la metodología de trabajo que se ha llevado a cabo y las tecnologías empleadas para realizar el proyecto.

Capítulo 5: En este capítulo se comentan los resultados obtenidos en la simulación, realizando también un análisis de diferentes pruebas a partir de estos resultados.

Capítulo 6: En este último capítulo, se comentan las conclusiones obtenidas tras la realización del proyecto, así como las posibles mejoras a realizar en un futuro.

Capítulo 2

Conceptos previos y estado del arte

En esta sección se pondrán en contexto el estado del arte en el que se encuentran los distintos algoritmos tanto de detección de patrones como de acoplamiento automático en estaciones de carga.

2.1. Detección de patrones

La **detección de patrones** es un problema bastante recurrente y abordado en la **Visión por Computador**, puesto que implica la detección de figuras claves que van a permitir automatizar multitud de procesos a partir de esta detección.

En nuestro caso, colocaremos el patrón encima de la base de carga y a una distancia apropiada, de modo que detectando el patrón, en realidad se está localizando la base de carga. Con esto pretendemos que, al detectar el patrón utilizado (es decir, el código QR) se comiencen a enviar los comandos necesarios para que el robot realice el acoplamiento en su base de carga.

Existen diversas técnicas para realizar esta detección. Si nos alejamos de los códigos QR y nos centramos en patrones más generales (como la que podemos apreciar

en la figura 2.1), podemos usar herramientas de la Visión por Computador como el detector de Canny [9], el ROI [5] (Region of Interest) y establecer una serie de restricciones geométricas que se adecuen al patrón que estamos tratando de localizar. Esto se trata en mayor profundidad en [13], donde se utilizan patrones distintos a los QR para localizar la estación de carga.

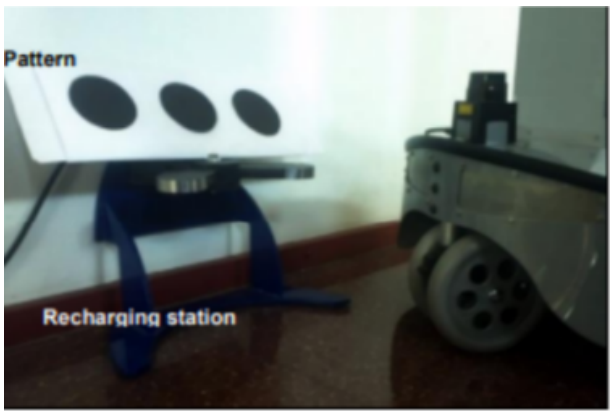


Figura 2.1: Ejemplo de patrón utilizado para detectar una base de carga en [13]. En este caso, el patrón se encuentra centrado en esta base.

Si nos centramos en los códigos QR como patrones, al tratarse de un tipo de patrón relativamente nuevo, no existen muchas técnicas basadas en su detección. Una de las propuestas que se utilizan, es la basada en el método de detección rápida de objetos [24], usada principalmente en la detección de rostros, y que funciona mediante características Haar-like [30] computadas a partir de imágenes integrales [24] y clasificadores *boosting and cascade*. Todo esto se puede ver con mayor detalle en [24] y [2]. Este método, como la gran mayoría, se basan en la estructura general de un código QR, según el estándar ISO/IEC 18004. Como vemos en la figura 4.6, podemos diferenciar las siguientes partes:

- 3 *finder pattern* (FIP) en las esquinas.
- 2 *timing pattern* (TP) entre los *finder patterns*.
- N *alignment patterns* (AP) en el área del QR.

Es mediante esta estructura en la que basan sus métodos de detección el método de detección rápida de objetos. Otro método también mencionado en [2] se basa

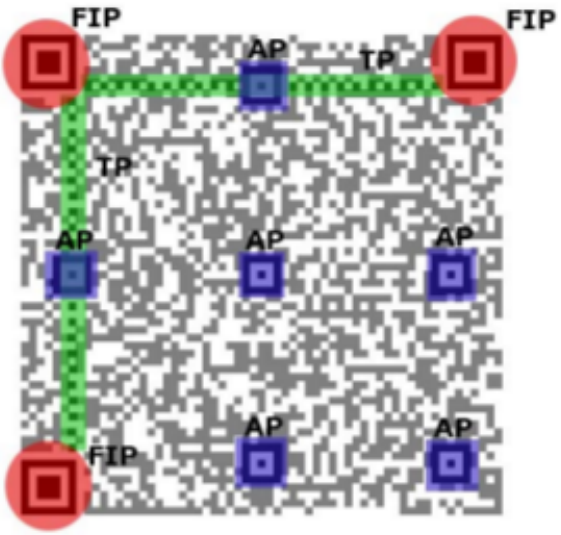


Figura 2.2: Estructura general de un código QR.

Fuente: `https://www.iso.org/standard/62021.html`

en tratar el problema como un problema de clasificación binario que clasifica los píxeles de una imagen como código QR o no.

2.2. Acoplamiento autónomo

El acoplamiento autónomo o automático, es un proceso de gran importancia para nuestro proyecto. Tomando como ejemplo a los robots domésticos, el acoplamiento automático nos permitirá que este robot realice la recarga de su batería al acabar cualquiera de sus tareas.

En este contexto, podemos encontrar varias técnicas para implementar el acoplamiento autónomo. Una de estas técnicas, esta basada en el algoritmo desarrollado en [13]. Brevemente, el algoritmo consta de los siguientes pasos:

1. Localizar la estación de carga.

2. Comparar la posición actual del robot con la esperada al estar en la acoplado en la estación de carga.

3. Enviar los comandos de movimiento al robot que minimicen el error entre

ambas posiciones.

4. Repetir si no ha habido acoplamiento.

Los pasos enumerados anteriormente los podemos ver mejor detallados en la figura 2.3.

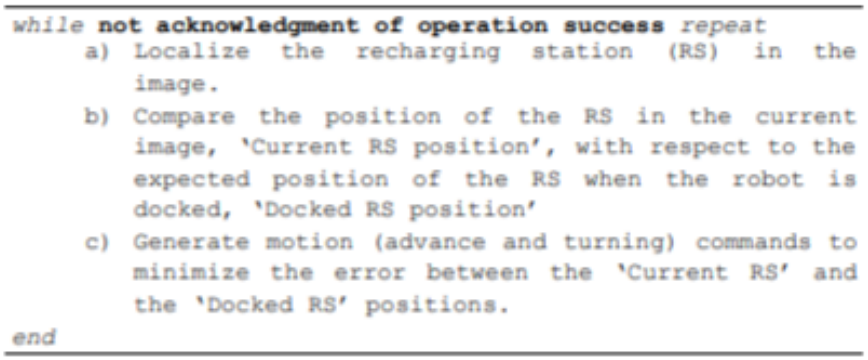


Figura 2.3: Algoritmo de acoplamiento autónomo según [13]

Otra técnica de acoplamiento autónomo, implementada en [25], se basa en un algoritmo compuesto de 2 fases: una fase de **aproximación** y otra, que es el propio **acoplamiento**. De manera resumida, en la fase de aproximación, el robot avanza hasta llegar a una zona de cambios en la región en la que se encuentra el objetivo (la base de carga en nuestro caso), comenzando la fase de acoplamiento. En la figura 2.4 podemos ver unos ejemplos en los que se dan ambas fases.

Otra propuesta para realizar el acoplamiento automático, consiste en el uso del **Aprendizaje por Refuerzo** [7, 22]. El aprendizaje por refuerzo es una de las principales áreas dentro del aprendizaje automático. Su objetivo consiste en decidir como debe actuar un agente en un determinado entorno tratando de maximizar algún tipo de recompensa o "premio", el cuál se deberá definir.

El aprendizaje por refuerzo provoca que un agente tenga que adquirir un comportamiento óptimo, interactuando con el entorno mediante el ensayo y error sin que

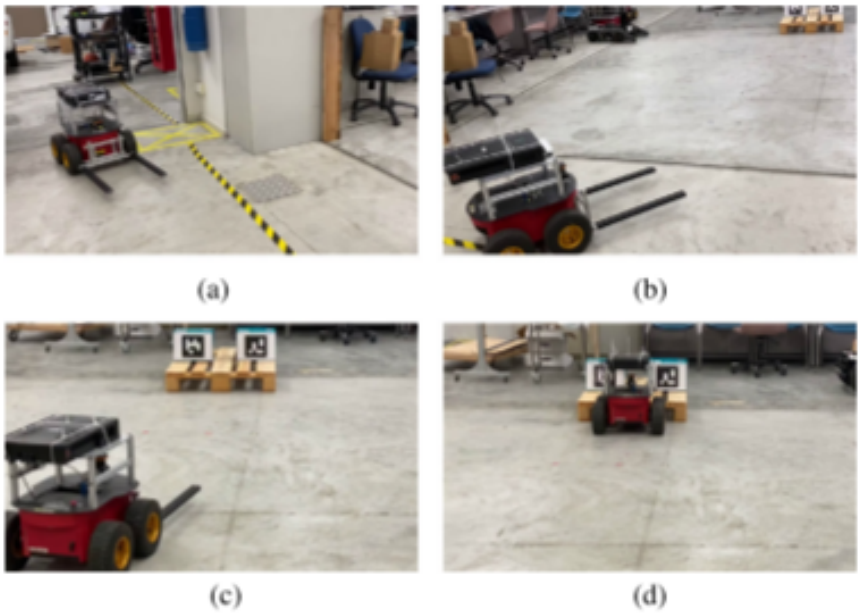


Figura 2.4: Ejemplos de las distintas fases del algoritmo según [25]. En este caso, el acoplamiento se realice en un palet en vez de en una base de carga.

se involucre el ser humano [17, 21].

Al entrenar un red de aprendizaje por refuerzo, se trata de construir una función que establezca una correspondencia entre los estados y las decisiones que tome el agente a partir de un objetivo prefijado por una función de recompensa. Las decisiones tomadas por el agente se aplican en el entorno, es decir, el espacio observado. En la figura 2.5 podemos ver un diagrama que resume este proceso. Algunas soluciones que implementan esta técnica las podemos encontrar, por ejemplo, en [6], donde se utiliza el aprendizaje automático cuando el robot activa la acción el proceso de acoplamiento en su base de carga. La solución propuesta consiste en que, con una determinada frecuencia, la red de aprendizaje por refuerzo recibe los siguientes parámetros:

- Una imagen proveniente de una cámara RGB para detectar el entorno.
- Un array con distancias obtenidas a partir de un escáner láser. Estas distancias servirán para evadir obstáculos.
- El centro y el tamaño del patrón que se va a usar y que identifica la estación

de recargar del robot. Esto se obtiene a partir de la imagen anterior.

Con estos parámetros, la red de aprendiza por refuerzo infiere acciones hasta que se tiene éxito o, tras un tiempo determinado, se contabiliza como un error.

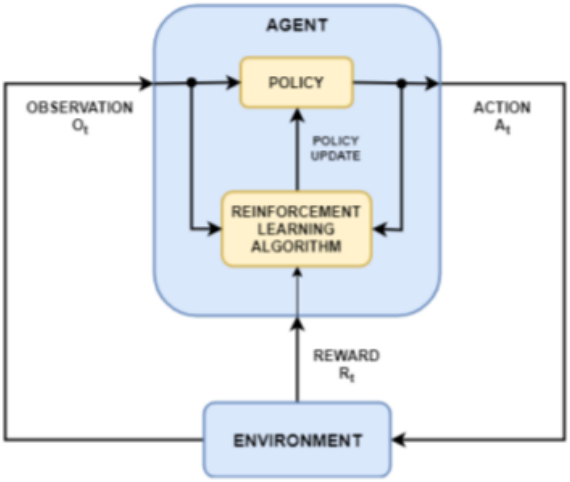


Figura 2.5: Ejemplo de diagrama en el que se resumen el funcionamiento del aprendizaje por refuerzo.

Fuente:`https://es.mathworks.com/help/reinforcement-learning/ug/what-is-reinforcement-learning.html`

Capítulo 3

Fundamentos Teóricos

3.1. Visión por Computador

La **Visión por Computador** [**12**], también conocida como visión artificial, se puede definir como el proceso de extracción de información del mundo real a partir de imágenes, usando para ello un ordenador. El objetivo es dotar al ordenador de un sistema de visión similar al del ser humano. Existen multitud de campos de aplicación para la Visión por Computador: industria (guiado de robots), medicina (diagnostico por computador), sistemas de seguridad (detección de movimientos), etc. En nuestro caso, la Visión por Computador nos será de gran ayuda para computar de manera sencilla y eficiente tanto la **Detección de Patrones** como el **Cálculo de la Homografía**.

3.1.1. Detección de Patrones

Comenzando por la detección de patrones, la Visión por Computador nos ofrece distintas técnicas para realizarla. Un posible método para detectar patrones, consiste en detectar puntos clave (o *keypoints*) que sean distintivos de la imagen. Como hemos visto en el apartado 2.1, existen diferentes técnicas para abordar la problemática de la detección de figuras claves que permiten identificar de manera unívoca el objeto en cuestión. Algunos algoritmos que utilizan este método son

Harris [14] o SIFT [15]. En nuestro caso, para detectar un código QR, usaremos una implementación basada en la detección de las 4 esquinas (ver figura 3.1) características de este tipo de códigos, siendo estas nuestros puntos claves.

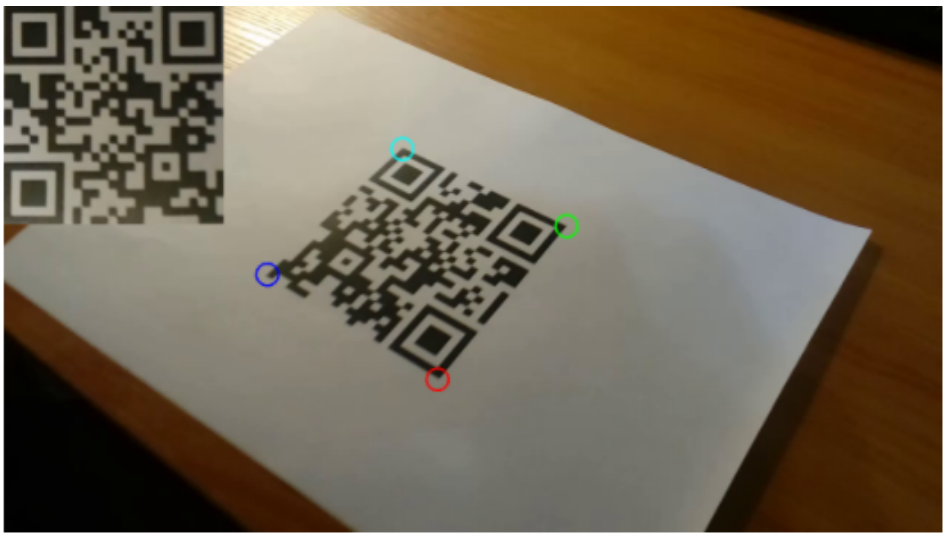


Figura 3.1: Esquinas características en un código QR. Los círculos marcan las esquinas que se usarán como puntos claves para la detección del código QR

3.1.2. Homografía

Otro aspecto clave en este proyecto, y que hemos nombrado varias veces ya a lo largo del documento, es el concepto de Homografía. Una homografía, también conocida como **transformación perspectiva/proyectiva**, se puede describir, de manera resumida, como la transformación existente entre dos imágenes distintas del mismo plano. Esta homografía nos permitirá establecer una correspondencia entre los puntos de estas dos imágenes. Esta correspondencia se calcula mediante la *matriz de homografía*, la cual, consiste en una matriz 3x3 (ver figura 3.2).

Como podemos deducir, al tratarse de una matriz 3x3, las homografías trabajan con vectores de 3 dimensiones, mientras que nosotros, al estar trabajando con imágenes, almacenamos los puntos de estas como vectores de 2 dimensiones, por lo que no podemos operar usando esta matriz de homografía en las condiciones

actuales. Para solventar este problema, introduciremos las **Coordenadas Homogéneas [3]**.

$$\begin{bmatrix} a & b & c \\ d & e & f \\ g & h & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = \begin{bmatrix} u \\ v \\ w \end{bmatrix} \qquad \begin{aligned} x' &= u/w \\ y' &= v/w \end{aligned}$$

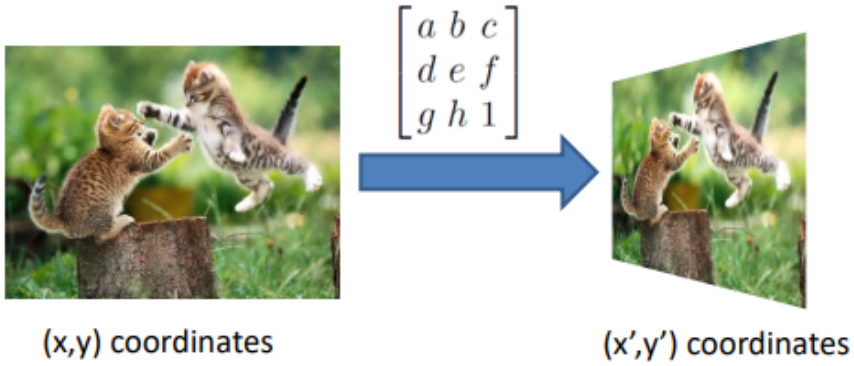


Figura 3.2: Ejemplo de aplicación de homografía. En la parte de arriba, observamos la transformación mediante la matriz de homografía del punto (x, y) al punto (x', y') usando coordenadas homogéneas. Justo debajo, observamos la aplicación de esta matriz de transformación a los puntos de la imagen de la izquierda, obteniéndose los puntos correspondientes en la imagen de la derecha.

Fuente: Szeliski, R. (2010). Computer vision: algorithms and applications.

Las coordenadas homogéneas nos servirán para expresar un punto con 2 dimensiones (2D) como un punto con 3 dimensiones (3D) y viceversa. Para pasar de 2D a 3D, basta con añadir al vector 2D una tercera componente, siendo esta un número cualquiera no negativo (por conveniencia, se utiliza el 1), obteniéndose así el vector $(x, y, 1)$. Para realizar el proceso inverso, es decir, pasar un punto 3D (X, Y, Z) a 2D (x, y), solo tenemos que dividir las componentes X e Y por la tercera componente Z, teniendo ($x = \frac{X}{Z}$ e $y = \frac{Y}{Z}$). Esto nos será de gran utilidad la hora de realizar la transformación de la homografía, pues nos permitirá pasar de coordenadas 2D (puntos de la cámara) a coordenadas 3D (puntos del mundo real) con relativa facilidad.

Cabe destacar que, si se diera el caso en que el valor de la tercera componente fuera 0, obtendríamos un punto en el infinito que no podríamos representar mediante

coordenadas no homogéneas. Por tanto, gracias a las coordenadas homogéneas, podremos operar usando la matriz de transformación de homografía.

Volviendo a las propias homografías, estas no son más que un tipo de transformación lineal entre dos planos. De entre todas las transformaciones disponibles, las homografías son las más generales, habiendo otras que se usan para un propósito más específico. Nuestra matriz de transformación en la homografía posee 8 grados de libertad, pero al usar otras transformaciones más simples, podemos reducir significativamente los grados de libertad necesarios.

Otra método para entender las homografías, es ponerlas en el contexto de otras transformaciones geométricas. A continuación, se presentará la jerarquía de transformaciones existentes (ver figura 3.3) y como se puede descomponer la homografía a partir de otras transformaciones más simples.

Definiremos, por tanto, las siguientes transformaciones:

- **Transformación de traslación:** Esta transformación es una de la más simples, realizando únicamente una traslación 2D. Se pueden escribir de 3 formas:
  - $x' = x + t$
  - $x' = [I\ t]x$
  - $x'_h = \begin{bmatrix} I & t \\ 0^T & 1 \end{bmatrix} x_h$

  Donde tenemos que t es el vector de traslación, I la matriz de identidad 2x2 y 0^T es un vector de ceros. La segunda forma, consistente en una matriz 2x3, es más compacta mientras que la tercera, al ser una matriz 3x3, nos permite aplicar la multiplicación de matrices y su inversa (notese que, en este caso, el subíndice h indica que trabajamos en coordenadas homogéneas).

Esta transformación posee 2 grados de libertad (1 para cada componente del vector t).

- **Transformación Rígida:** También conocida como Isométrica o Euclídea, ya que se caracteriza por preservar la distancia Euclídea (la distancia entre dos puntos de una imagen será igual que en la imagen transformada). Se usa para aplicar rotaciones y traslaciones, por lo que tiene 3 grados de libertad (los 2 de la traslación más uno para los ángulos de rotación). Puede escribirse como:

$$x' = \begin{bmatrix} R & t \\ 0^T & 1 \end{bmatrix} x$$

Donde $R = \begin{bmatrix} cos\theta & -sen\theta \\ sen\theta & cos\theta \end{bmatrix}$ es una matriz de rotación 2x2, t es un vector de traslación de dimensión 2 y 0^T es un vector de 2 ceros.

- **Transformación de Similitud:** Una transformación de similitud es similar, valga la redundancia, a una transformación rígida, con la excepción de que en la de similitud se añade un factor de escala. Este escalado es invariante respecto a la dirección y esta transformación tiene 4 grados de libertad. Esta transformación se puede escribir como:

$$x' = \begin{bmatrix} sR & t \\ 0^T & 1 \end{bmatrix} x$$

Con s siendo un escalar que representa la escala descrita anteriormente.

- **Transformación Afín:** Esta transformación es como la de similitud, pero compuesta por dos rotaciones y dos escalados. Esta transformación posee 6 grados de libertad y se puede escribir como:
Con A siendo una matriz de 2x2 que contiene las rotaciones y escalados anteriores.

$$x' = \begin{bmatrix} A & t \\ 0^T & 1 \end{bmatrix} x$$

- **Transformación Proyectiva:** Por último, llegamos a la homografía o transformación proyectiva. Se trata de una transformación lineal en coordenadas homogéneas. Esta transformación sería no lineal si no fuera por las coordenadas homogéneas. Estas transformación posee 8 grados de libertad y se escribe como:

$$x' = \begin{bmatrix} A & t \\ v^T & v_3 \end{bmatrix} x$$

Con $v = (v_1, v_2)^T$, siendo este v lo que la diferencia de la transformación afín. A su vez, el vector v es el responsable de los efectos no lineales en esta transformación. Para el factor de escala, podemos fijar el valor de v_3 a 1, pudiendo escribirse entonces como:

$$x' = \begin{bmatrix} a & b & c \\ d & e & f \\ g & h & 1 \end{bmatrix} x$$

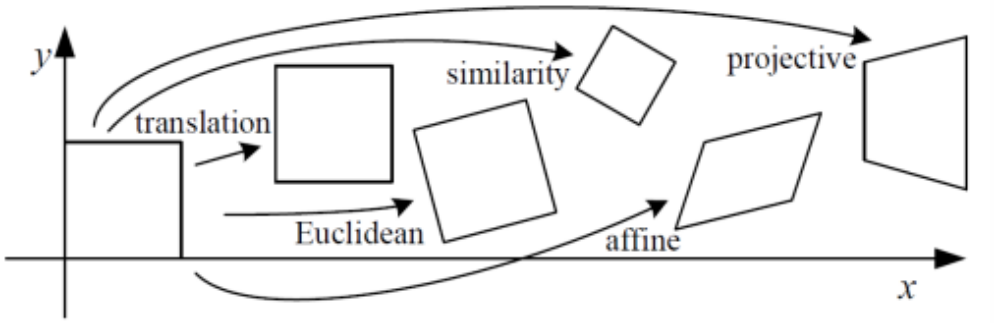
Donde $A = \begin{bmatrix} a & b \\ \\ d & e \end{bmatrix}$.

Otra forma de representar a la homografía es mediante la descomposición de las transformaciones previas que hemos mostrado anteriormente, siendo de la forma

$$H = H_S H_A H_P = \begin{pmatrix} sR & t \\ 0^T & 1 \end{pmatrix} \begin{pmatrix} U & 0 \\ 0^T & 1 \end{pmatrix} \begin{pmatrix} I & 0 \\ v^T & v_3 \end{pmatrix} = \begin{pmatrix} A & t \\ v^T & v_3 \end{pmatrix}$$

En esta ocasión H_S representa una transformación de similitud, H_A una trasformación afín y H_P una proyección. Por tanto, para este caso tenemos que

$A = sRU + tv^T$ y U una matriz triangular superior, como las usadas en la factorización LU [1], normalizada con determinante 1. Cabe destacar que, para que esta descomposición sea válida, v_3 no puede valer 0. Adicionalmente, si el factor de escala s es positivo, la descomposición es única.



Transformation	Matrix	# DoF	Preserves	Icon
translation	$\left[\, \boldsymbol{I} \mid \boldsymbol{t} \,\right]_{2\times3}$	2	orientation	
rigid (Euclidean)	$\left[\, \boldsymbol{R} \mid \boldsymbol{t} \,\right]_{2\times3}$	3	lengths	
similarity	$\left[\, s\boldsymbol{R} \mid \boldsymbol{t} \,\right]_{2\times3}$	4	angles	
affine	$\left[\, \boldsymbol{A} \,\right]_{2\times3}$	6	parallelism	
projective	$\left[\, \tilde{\boldsymbol{H}} \,\right]_{3\times3}$	8	straight lines	

Figura 3.3: Esquema jerárquico de las transformaciones geométricas más utilizadas. En la tabla, podemos ver tanto los grados de libertad (DoF) como otras características de cada transformación.

Fuente: Szeliski, R. (2010). Computer vision: algorithms and applications.

Con las homografías ya introducidas, ahora debemos resolver el sistema queda planteado en las mismas, a partir de su ecuación propia que vimos anteriormente. Este sistema lo podemos ver en la figura 3.4.

[1] https://es.wikipedia.org/wiki/Factorizaci%C3%B3n_LU

$$\lambda \begin{bmatrix} x_i' \\ y_i' \\ 1 \end{bmatrix} = \begin{bmatrix} h_{00} & h_{01} & h_{02} \\ h_{10} & h_{11} & h_{12} \\ h_{20} & h_{21} & h_{22} \end{bmatrix} \begin{bmatrix} x_i \\ y_i \\ 1 \end{bmatrix}$$

Figura 3.4: Sistema de ecuaciones para resolver una homografía.

Uno de los métodos mas simples para resolver este sistema es el método DLT (Direct Linear Transformation) en el que, a partir de las coordenadas de un punto en ambas imágenes (original y resultante de la homografía), podemos construir un sistema de ecuaciones para aislar los valores de la matriz, quedando de la siguiente forma:

$$x_i' = \frac{h_{00}x_i + h_{01}y_i + h_{02}}{h_{20}x_i + h_{21}y_i + 1} \longrightarrow x_i'(h_{20}x_i + h_{21}y_i + 1) = h_{00}x_i + h_{01}y_i + h_{02}$$

$$y_i' = \frac{h_{10}x_i + h_{11}y_i + h_{12}}{h_{20}x_i + h_{21}y_i + 1} \longrightarrow y_i'(h_{20}x_i + h_{21}y_i + 1) = h_{10}x_i + h_{11}y_i + h_{12}$$

Este sistema puede reescribirse de la siguiente forma:

$$\mathbf{Ah} = 0 \longrightarrow \begin{bmatrix} -x_i & -y_i & -1 & 0 & 0 & 0 & x_i'x_i & x_i'y_i & x_i' \\ 0 & 0 & 0 & -x_i & -y_i & -1 & y_i'x_i & y_i'y_i & y_i' \end{bmatrix} \begin{bmatrix} h_{00} \\ h_{01} \\ h_{02} \\ h_{10} \\ h_{11} \\ h_{12} \\ h_{20} \\ h_{21} \\ 1 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \end{bmatrix}$$

Donde h es la solución y puede estar escalado por un valor k. Como vemos, con menos de 4 pares puntos independientes, existen infinitas soluciones, pero si usamos 4 o más pares de puntos independientes, tendremos el siguiente sistema:

$$Ah = 0 \longrightarrow \begin{bmatrix} -x_1 & -y_1 & -1 & 0 & 0 & 0 & x'_1x_1 & x'_1y_1 & x'_1 \\ 0 & 0 & 0 & -x_1 & -y_1 & -1 & y'_1x_1 & y'_1y_1 & y'_1 \\ & & & & \vdots & & & & \\ -x_n & -y_n & -1 & 0 & 0 & 0 & x'_nx_n & x'_ny_n & x'_n \\ 0 & 0 & 0 & -x_n & -y_n & -1 & y'_nx_n & y'_ny_n & y'_n \end{bmatrix} \begin{bmatrix} h_{00} \\ h_{01} \\ h_{02} \\ h_{10} \\ h_{11} \\ h_{12} \\ h_{20} \\ h_{21} \\ 1 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ \vdots \\ 0 \\ 0 \end{bmatrix}$$

Para este sistema tenemos 2 casos:

1. **Exactamente 4 pares de puntos:** En este caso, existe una única solución con k y h distintos de 0, pero esta puede verse afectada por el ruido.

2. **Más de 4 pares de puntos:** En este caso, tenemos un sistema sobre-determinado y no existiría una solución distinta que $h = 0$. Sin embargo, podemos obtener la solución que minimice el error en las coordenadas de los puntos (es decir, usar mínimos cuadrados). Esta solución presenta mayor robustez frente al ruido.

Cabe destacar que, aunque en nuestro caso estamos hablando de las homografías, dependiendo de la trasformación que usemos, menos pares de puntos serán necesarios (aunque como hemos mencionado anteriormente, cuantos más puntos, mayor robustez al ruido).

La homografía es una herramienta ampliamente utilizada en Visión por Computador, usándose en la rectificación de imágenes, registro de imágenes o para computar el movimiento de la cámara entre dos imágenes (obteniendo la rotación y traslación asociadas). Aplicándolo a nuestro caso, la función que realizará la homografía será, una vez realizada la detección del código QR, calcular la matriz de transformación entre la imagen del código QR detectado desde la posición actual del robot y el QR desde la posición en la que se encontraría el robot al realizar la recarga. Para el cálculo de esta homografía, usaremos los 4 puntos del código QR que hemos calculado en el proceso de detección.

Capítulo 4

Implementación

En este capítulo, mostraremos la manera en que se ha implementado la aplicación y el software utilizado para ello.

La aplicación implementada consta de diversos módulos[1] que realizan las funcionalidades necesarias para alcanzar los objetivos descritos. Esta división en módulos hace que sea más fácil identificar en que parte se realiza cada función y, de ser necesario, poder modificarla.

Los módulos de los que está compuesta la aplicación son:

- Detección de códigos QR (*qr_detector.py*)
- Calibración de cámara (*calibration.py*)
- Cálculo de Homografía y vectores (*calculate_homography.py*)

Finalmente, se ha añadido un módulo adicional con la navegación y el acoplamiento, en el que se detalla el proceso seguido al realizarse la simulación de los módulos anteriores en nuestro robot.

La utilización de estos módulos requiere utilizar imágenes o vídeos en los que puedan ser aplicados. El resto del proceso se realiza de forma automática.

[1]Los modulos se pueden ver en detalle en `https://github.com/quiquesk8/TFG.git`

En primer lugar, se introducirán las tecnologías que se han empleado para el desarrollo y, finalmente, se comentará cada módulo en detalle.

4.1. Tecnologías Empleadas

A continuación, se procederá a describir las herramientas que han facilitado el desarrollo de este proyecto. Se obviarán herramientas comunes tales como el sistema operativo, editores de código, etc.

4.1.1. OpenCV

Como ya se ha mencionado en apartados anteriores, para la implementación de varios de los algoritmos de este proyecto, se hará uso de la librería **OpenCV** [**4**], la cual, se trata de una librería de código abierto, disponible para distintos lenguajes de programación (C++,Python,etc), que proporciona multitud de algoritmos de Visión por Computador. Dentro de esta librería podemos encontrar varios módulos (ver figura 4.1) que implementan los algoritmos que nos permitirán realizar la detección de objetos (los códigos QR en nuestro caso) y el cálculo de la homografía necesaria, así como funciones más básicas que nos permitirán procesar, leer o guardar imágenes.

Para nuestro proyecto emplearemos los siguientes módulos, que comentaremos en mayor detalle a continuación. Estos son **calib3d** y **objdetect**. El primero nos permite realizar reconstrucciones 3D y realizar operaciones relacionadas con la calibración de la cámara, que nos servirá para el cálculo de nuestra homografía o realizar la calibración de la cámara entre otros. El segundo modulo, nos permitirá implementar la detección de objetos para nuestro proyecto, que en nuestro caso consiste en la detección de los códigos QR como patrones.

Si nos adentramos en estos módulos, podemos observar distintas características. Cada módulo suele componerse principalmente por una serie de clases y funciones,

(a) Módulos principales de OpenCV

(b) Algunos de los módulos extra de OpenCV

Figura 4.1: Documentación de OpenCV. A la izquierda se pueden observar los módulos principales y a la derecha, algunos de los módulos especiales adicionales.

además de una descripción teórica al inicio de cada módulo que pone en contexto el funcionamiento del mismo[2].

En el módulo calib3d, algunas de las funciones más destacables son **calibrateCamera()**, que permite realizar la calibración de una cámara, **findHomography()**, que permite encontrar la homografía entre varios puntos de dos imágenes, o **decomposeHomographyMat()** que permite descomponer la matriz de transformación de una homografía en posibles rotaciones, traslaciones y vectores normales al plano.

En el módulo objdetect, podemos destacar algunas clases como **QRCodeDetector**, que permite realizar la detección y decodificación de los códigos QR entre

[2]Ejemplo de la estructura de un modulo en OpenCV: `https://docs.opencv.org/4.5.3/d5/d54/group__objdetect.html`

otras cosas, o los tipos de datos struct **DetectionROI** y **HOGDescriptor**, que implementan respectivamente la técnica de regiones de interés (ROI) [5] y el descriptor basado en los histogramas de gradientes orientados (HOG) [11].

4.1.2. ROS

Para la simulación de nuestro algoritmo utilizaremos, entre otras herramientas que mencionaremos posteriormente, ROS [18, 19], cuya instalación se detalla en en el apéndice B.

Figura 4.2: Logo de ROS.

ROS, acrónimo de Robot Operating System, se trata de un framework que permite el desarrollo de software para robots. Posee una gran cantidad de librerías y herramientas que simplifican la labor de crear diferentes comportamientos en un robot en distintas plataformas robóticas. A su vez, proporciona un entorno de ejecución que permite la interacción de las distintas piezas de software entre ellas.

ROS es un software de código abierto que posee una gran comunidad de usuarios por el mundo, quienes contribuyen aportando documentación, tutoriales, resolución de dudas en foros asociados, etc. Esta filosofía desemboca en un entorno colaborativo en el que se puede obtener y compartir conocimientos en el campo de la robótica.

El ecosistema de ROS aporta diferentes herramientas que facilitan el desarrollo de software para robots. Para comprender como ROS interactúa con otros programas y software, explicaremos brevemente los elementos principales que conforman ROS:

- **Paquetes**: ROS está organizado en paquetes, en los que se almacenan conjuntamente los diferentes procesos, librerías, archivos de configuración y código, etc.

- **Nodos:** Los nodos son archivos ejecutables localizados en los paquetes. Permiten la comunicación con otros nodos y que pueden publicar o suscribirse a tópicos. Pueden estar escritos tanto en C++ como en Python.

- **Mensajes:** Los nodos se comunican entre ellos enviando mensajes a los tópicos. Los mensajes son estructuras de datos simples que pueden contener desde tipos primitivos estándar (integer, floating point, boolean, etc.) hasta arrays.

- **Tópicos:** Los tópicos o *topics* son los encargados de almacenar el contenido de los mensajes. El funcionamiento del proceso de comunicación funciona de la siguiente manera (ver figura 4.3). Un nodo envía un mensaje mediante la publicación de un tópico dado. De esa forma, cualquier nodo suscrito a ese tópico recibirá dicho mensaje. Todos los mensajes pertenecientes a un mismo tópico deben compartir la misma estructura de datos.

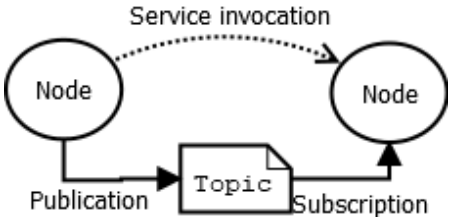


Figura 4.3: Grafo simplificado que ilustra el funcionamiento de la comunicación en ROS.

Fuente: `http://wiki.ros.org/ROS/Concepts`.

4.1.3. Gazebo

De manera subsecuente a ROS, para la simulación también utilizaremos la herramienta Gazebo.

Gazebo es un simulador de robots que permite el desarrollo y creación de un entorno simulado en el que el robot desarrollará las tareas expuestas anteriormente. Mientras que ROS sería el encargado de imprimir la lógica al robot, Gazebo se encarga de simular el entorno en el que el robot va a desarrollar dicha lógica.

Por tanto, Gazebo funciona de forma conjunta con ROS para hacer que el robot desempeñe sus funciones correspondientes de manera correcta, existiendo algunos comandos en ROS que integran los entornos simulados de burgeno2021dockingazebo, como *Roslaunch*[3], que permite integrar los nodos de ROS en el entorno generado en en dicho simulador.

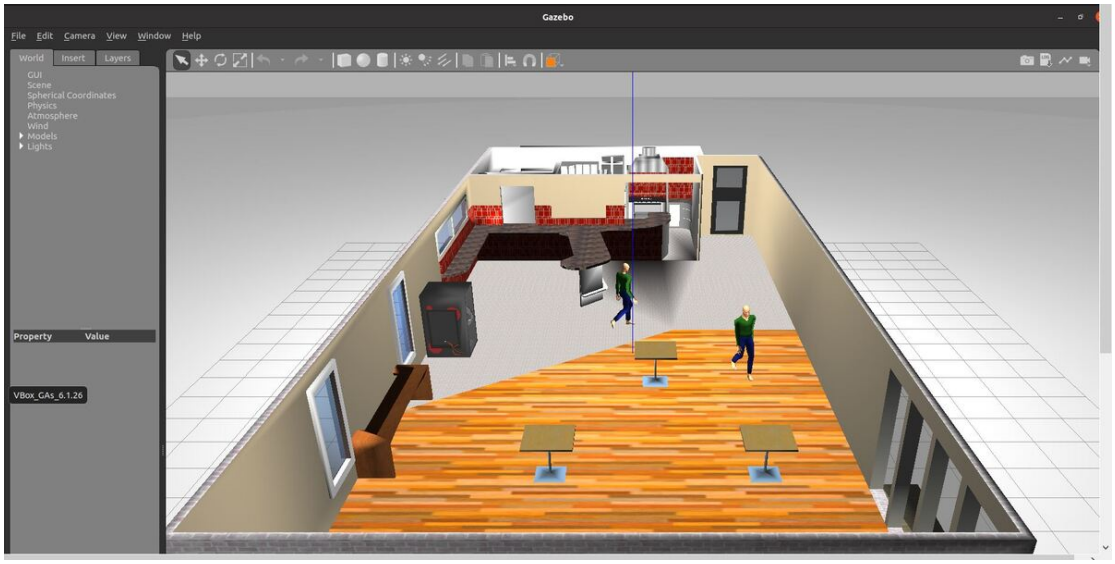


Figura 4.4: Ejemplo de entorno simulado con Gazebo.

4.1.4. Máquina virtual

Tanto ROS como Gazebo están pensados para ser usados en una distribución basada en Linux, sin embargo, la mayor parte de la aplicación se ha desarrollado sobre un entorno Windows, por lo que se hace necesario la utilización de una máquina virtual, en nuestro caso Virtual Box[4], para poder usar ambas distribuciones

[3] `http://wiki.ros.org/roslaunch`
[4] `https://www.virtualbox.org/`

de manera simultanea. Cabe destacar que el uso de una máquina virtual puede afectar al rendimiento de la simulación de manera sustancial.

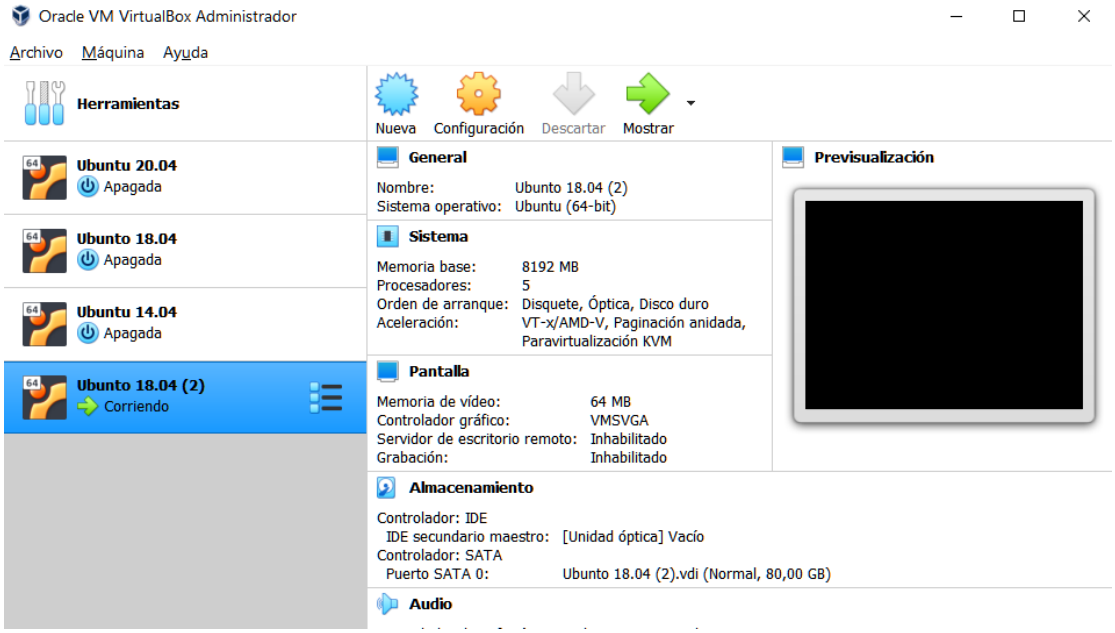


Figura 4.5: Ejemplo diferentes distribuciones virtualizadas con Virtual Box.

4.2. Implementación de los módulos

4.2.1. Detección de códigos QR

Para la detección de códigos QR, existen diferentes librerías operativas, como la que proporciona la ya mencionada en otros capítulos OpenCV o la proporcionada por la librería zbar, **pyzbar**.

Si centramos la vista en OpenCV, esta librería nos proporcionará multitud de herramientas en los demás módulos, por lo que usarla como método de detección nos proporcionará una mayor compatibilidad y homogeneidad entre los módulos, pues estos están implementados para que funcionen entre sí, a diferencia de si utilizásemos librerías externas que, en un principio, no estarían preparadas para trabajar con los datos que necesita y devuelve OpenCV.

Para empezar a usarla, deberemos instalar la librería OpenCV en primer lugar. Este proceso se puede ver de forma detallada en el apéndice A.

Una vez instalada en nuestro proyecto, usaremos la versión para Python **cv2** para importar OpenCV de manera completa. Tras esto, nos encontramos en disposición para comenzar a utilizar esta librería.

Para realizar la detección de los códigos QR con OpenCV, utilizaremos la clase **QRCodeDetector()**, perteneciente al módulo de detección de objetos de OpenCV. Esta clase implementa multitud de métodos muy útiles a la hora de trabajar con códigos QR. Los métodos más útiles son **detect()** y **detectAndDecode()**. Con el primero, obtenemos las 4 esquinas del QR, si es que existe en la imagen que pasamos como parámetro. Con el segundo, además de la detección, se decodifica el contenido del QR una vez detectado con el método anterior. A partir de estos 4 puntos obtenidos, seremos capaces de dibujar el contorno de nuestro código QR para poder localizarlo visualmente, como podemos ver en la figura 4.6. Este contorno también se genera con OpenCV.

Poniendo el foco ahora en la librería **pyzbar**, esta funciona de manera similar al método de OpenCV ya expuesto con algunas sutiles diferencias. En este caso, una vez importada la librería, la función que nos permitirá realizar la detección y decodificación del QR es **decode()**, a la que deberemos pasar la imagen en la que se encuentre el QR. Al igual que con OpenCV, el proceso de instalación se encuentra detallado en el apéndice A.

Esta función devuelve varios componentes, que enumeraremos a continuación.

- **Data:** Este elemento contiene los datos decodificados del QR.
- **Rect:** Este elemento esta formado por 4 componentes. Las dos primeras (x,y) son las coordenadas de la primera esquina del QR, mientras que las otras dos son el ancho y el alto del QR.

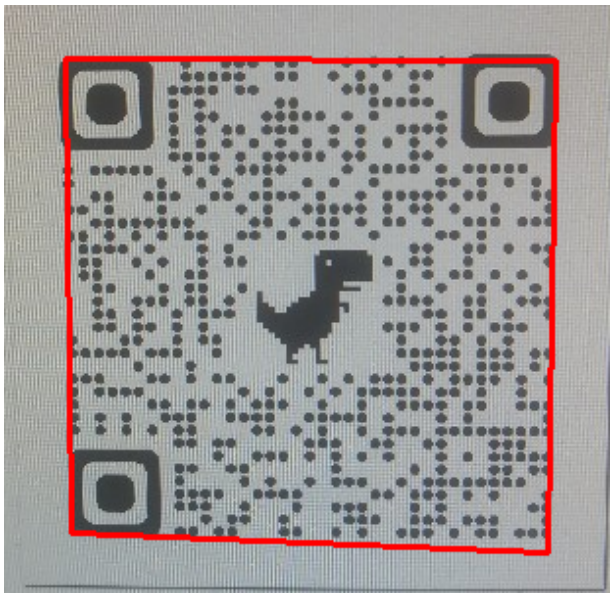


Figura 4.6: Ejemplo de detección de código QR con OpenCV. A partir de las esquinas obtenidas en la detección, generamos el contorno que se muestra en la imagen.

- **Polygon:** Este elemento devuelve los 4 puntos que conforman las 4 esquinas del QR.

Con estos componentes, se puede realizar la detección y decodificación de un código QR, como podemos ver en la figura 4.7.

Figura 4.7: Ejemplo de detección de código QR con pyzbar.

4.2.2. Calibración de cámara

El proceso de **calibración de una cámara** [**23**] se antoja esencial en este proyecto, ya que para poder hallar la homografía que nos permitirá comandar a nuestro robot, debemos conocer la matriz K de parámetros intrínsecos de la cámara. Ahondaremos en estos detalles a continuación.

Este proceso consiste en la obtención de dos tipos de parámetros: los parámetros **extrínsecos** y los **intrínsecos**.

Los parámetros extrínsecos son aquellos que depende de la posición y orientación de la cámara en el mundo real mientras que los parámetros intrínsecos son los inherentes para una misma cámara y no varían mientras se mantenga esta cámara. También se incluye en este proceso la obtención de los parámetros de distorsión radial, que son los que nos permitirán eliminar la distorsión en las imágenes que genere la cámara, aunque sobre todo nos centraremos en los dos primeros.

Los parámetros extrínsecos están formados por un vector de traslación (t) y un vector de rotación (R) en el sistema de referencia del mundo real. Estos dos vectores forman un total de 6 parámetros extrínsecos, que son las 3 componentes del vector t y los 3 ángulos de rotación de las matrices de rotación (ver figura 4.8), conocidos como *yaw, pitch y roll* [26].

$$R_z(\theta) = \begin{bmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix} \quad R_y(\theta) = \begin{bmatrix} \cos\theta & 0 & \sin\theta \\ 0 & 1 & 0 \\ -\sin\theta & 0 & \cos\theta \end{bmatrix} \quad R_x(\theta) = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos\theta & -\sin\theta \\ 0 & \sin\theta & \cos\theta \end{bmatrix}$$

Figura 4.8: Matrices de rotación para roll, pitch y yaw respectivamente.

Por otro lado, los parámetros intrínsecos están compuestos por la distancia focal (f), los parámetros de escala (k_x, k_y) y el centro de la imagen, es decir, las coordenadas del punto principal (u_0, v_0), formando un total de 5 parámetros intrínsecos, que son los que nos darán la matriz K para hallar nuestra homografía.

Una vez sabemos esto, existen distinto métodos para implementar la calibración de una cámara. El que nosotros implementaremos, y a la vez uno de los más usados, será el método de Zhang [31], en el que se siguen los siguientes pasos:

1. Imprimir la plantilla de calibración (por ejemplo, un tablero de ajedrez) y pegarla en una superficie plana.

2. Tomar varias fotos de la plantilla desde distintas posiciones y ángulos. A mayor numero de tomas, la calibración será más precisa.

3. Detectar los esquinas de los cuadrados en las imágenes.

4. Estimar los parámetros intrínsecos y extrínsecos mediante una solución de forma no cerrada [31].

5. Refinar la solución anterior mediante un sistema no lineal para encontrar unos valores más precisos.

Una vez expuesto el método con el que calibraremos nuestra cámara, pasaremos a hablar del modulo en el que se ha implementado todo lo anterior. Como en el caso de la detección, OpenCV proporciona una implementación para realizar esta calibración.

En primer lugar, debemos definir el número esquinas por fila y por columna que hay en nuestra plantilla de calibración, de tal forma que quede definido nuestro patrón de calibración. En nuestro ejemplo, usaremos un tablero de 8x6.

A continuación, dibujaremos todas las esquinas en el tablero. Para ello, debemos cargar todas las fotos de nuestro tablero desde distintas perspectivas y usar la función de OpenCV **findChessboardCorners()** para encontrar estas esquinas. Esta función recibe cada imagen del tablero (previamente convertida a una imagen en gris) junto con el par del número esquinas por fila y por columna. Posteriormente, los pintamos con **drawChessboardCorners()** usando las esquinas ya detectadas.

Por último, para realizar la calibración con el método de Zhang, haremos uso de la función **calibrateCamera()**, a la que le pasamos todos los puntos de las imágenes de nuestro tablero y la proyección de estos puntos. Con esto, obtenemos la matriz K de parámetros intrínsecos, los vectores de rotación y traslación entre todas las imágenes usadas en la calibración (parámetros extrínsecos) y los coeficientes de distorsión. El resultado final, con todas las imágenes con sus respectivas esquinas pintadas, lo podemos ver en la figura 4.9.

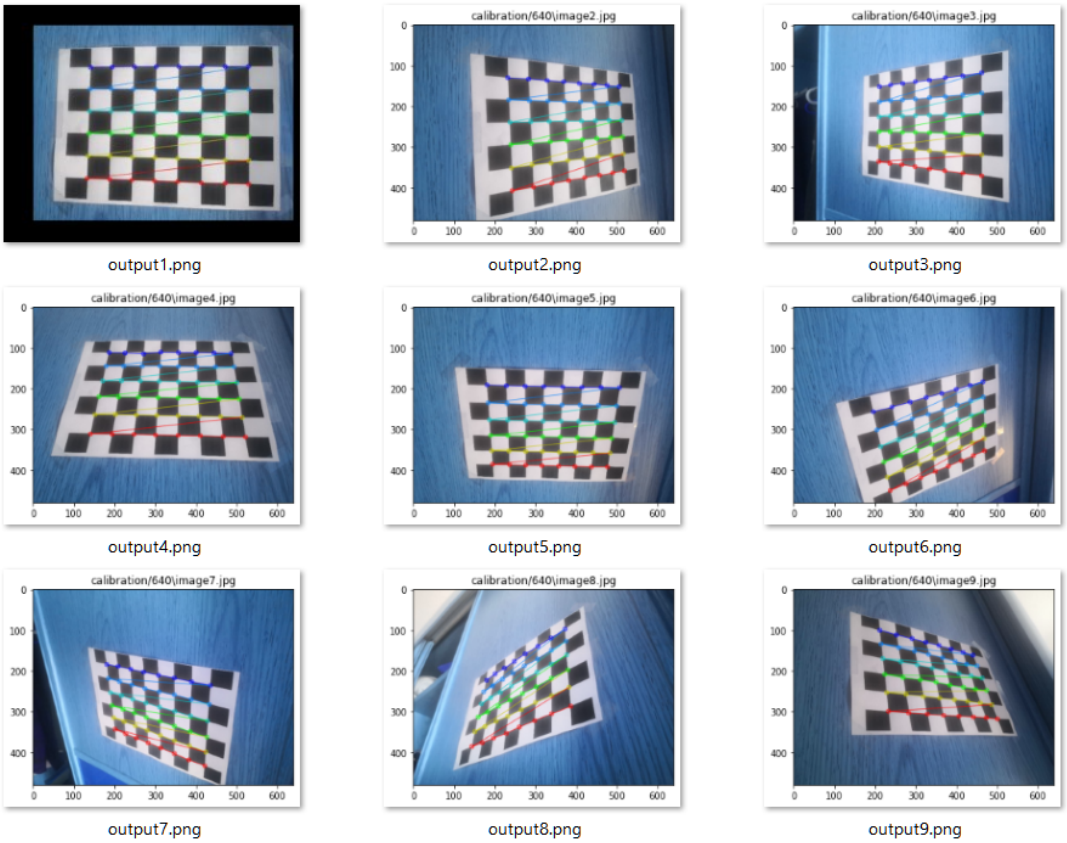


Figura 4.9: Proceso de calibración para tablero 8x6 con 9 imágenes.

Finalizada la calibración de la cámara y obtenida la matriz de calibración K con los parámetros intrínsecos de la misma, podemos proceder al cálculo de la homografía.

4.2.3. Cálculo de Homografía y vectores

Como ya se comentó en el apartado 3.1.2, las homografías serán de vital importancia en nuestro proyecto, ya que nos permitirán calcular las correspondencias

entre los puntos de dos imágenes en el mismo plano.

En nuestro caso, usaremos las homografías para hallar las correspondencias entre el código QR desde la posición actual en la que se encuentre nuestro robot y la posición en la que se encontraría el robot al estar realizando la recarga en la estación de carga.

Para la implementación, volveremos a hacer uso de la librería OpenCV, usada en los módulos anteriores, y que también nos proporciona las herramientas necesarias para realizar el cálculo de esta homografía. OpenCV proporciona dos formas de realizar el cálculo de la homografía entre dos imágenes.

La primera, es usar la función **findHomography()**, que como su nombre indica, encuentra la homografía existente entre dos planos. A esta función le deberemos pasar los puntos del código QR en el plano actual (en su posición actual) y en el plano en el que realiza la recarga de la batería.

La segunda forma consiste en usar otra función similar, **getPerspectiveTransform()**. Al igual que la anterior función, también recibirá los puntos del código QR en ambos planos para calcular la homografía.

La principal diferencia entre ambas está en el número de puntos que se pasan como coordenadas. Recordando lo expuesto sobre las homografías en la sección 3.1, la solución de una homografía viene dada dependiendo si proporcionamos exactamente 4 pares de puntos o más. *getPerspectiveTransform()* utiliza únicamente 4 pares de puntos, aplicando el caso teórico en el que pasamos 4 pares de puntos, mientras que *findHomography()* puede utilizar 4 o más puntos, aplicando el otro caso existente. En este último, en el caso de pasar más de 4 pares, es posible escoger el algoritmo[5] con el que se computa la homografía. A efectos prácticos, *findHomography()* se suele usar cuando el conjunto de puntos se detecta de mane-

[5] `https://docs.opencv.org/3.4/d9/d0c/group__calib3d.html`

ra automática, mientras que *getPerspectiveTransform()* se utiliza habitualmente cuando se escogen manualmente los puntos.

Antes de continuar, es recomendable comprobar que la homografía se ha calculado correctamente. Para ello, podemos hacer uso de la función **warpPerspective()**, que nos servirá para aplicar una transformación de perspectiva aplicando una matriz de transformación. Si calculamos la homografía entre dos imágenes en planos distintos, una forma de comprobar que se ha aplicado esta homografía correctamente, es usar esta función con la imagen en el plano original y ver si da como resultado la imagen en el plano transformado. Esto lo podemos ver en la figura 4.10, donde podemos comprobar que la segunda y la tercera imagen son prácticamente idénticas.

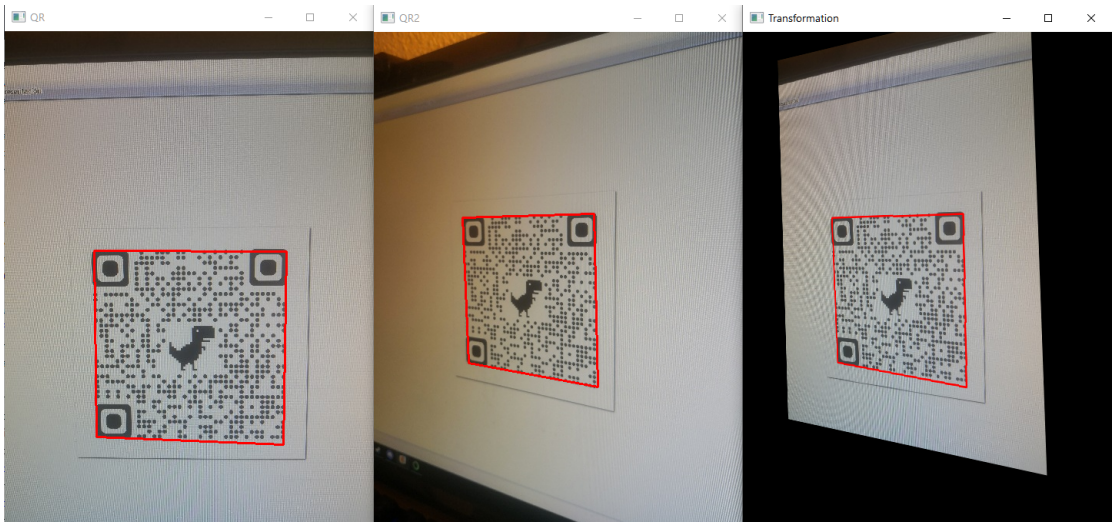


Figura 4.10: Comprobación de aplicación de homografía. La primera imagen desde la izquierda es la imagen en el plano original, la segunda corresponde a la transformación realizada moviendo la cámara. La tercera, corresponde a la transformación realizada con la función *warpPerspective()* y la matriz de homografía entre las dos primeras imágenes.

Una vez tenemos nuestra matriz de transformación de homografía calculada, nuestro siguiente objetivo es obtener los vectores de rotación, traslación y normales que conforman esta homografía o, en otras palabras, descomponer la homografía.

Para ello, OpenCV ofrece el método **decomposeHomographyMat()**, en el que, dada la matriz de transformación de la homografía y la matriz de calibración de la cámara de los parámetros intrínsecos (que calculamos en el modulo anterior), obtenemos la rotación, traslación y vector normal al plano que necesitamos para comandar al robot.

Sin embargo, este método presenta un inconveniente, y es que no devuelve una única rotación, traslación y vector normal al plano, sino que devuelve 4 posibles soluciones. Los detalles de la descomposición y de como se realizan los podemos encontrar [16].

Como trabajar con 4 posibles soluciones no resulta nada óptimo, un método para paliar este problema consiste en usar la función **filterHomographyDecompByVisibleRefpoints()**. Esta función se basa en que, como se dice en [16], de las 4 soluciones devueltas, como mínimo 2 son soluciones única y las otras 2 son sus opuestas. Si tenemos acceso al conjunto de puntos en el plano original y a los del plano tras aplicar la homografía, podemos verificar cuales son realmente potenciales soluciones y cuales son las soluciones opuestas comprobando que homografías son consistentes con los puntos proporcionados.

Con este método, tenemos garantizado que, como máximo, vamos a tener dos posibles conjuntos de rotaciones, traslaciones y vectores normales, por lo que es mucho más factible para la simulación, pudiendo elegir cual de los dos usar basándonos, por ejemplo, en los vectores calculados en una etapa anterior a este.

4.2.4. Navegación y acoplamiento automático

Una vez ya hemos completado el cálculo de la homografía, obteniendo sus respectivos vectores de rotación y traslación, los utilizaremos para realizar la navegación del robot y su posterior acoplamiento a la estación de carga.

En primer lugar, cabe destacar que el funcionamiento de la navegación y del algoritmo de acoplamiento automático está supeditado al robot que se utilice. ROS proporciona una amplia variedad de robots que van desde robots simples hasta robots humanoides. En nuestro caso, la implementación se ha basado en el robot conocido como **Turtlebot**[6], en concreto el Turtlebot3, que posee dos versiones, *Burger* y *Waffle pi* (ver figura 4.11). Existen varias diferencias entre estos modelos, pero la principal es que el modelo *Burger* no incluye cámara, mientras que el *Waffle pi* si, por lo que usaremos este último.

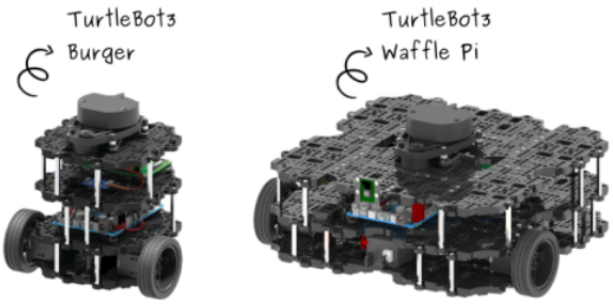


Figura 4.11: A la izquierda, la versión *Burger* del Turtlebot. A la derecha, su versión *Waffle pi*.

Otra característica del Turtlebot que conviene tener en cuenta a la hora de realizar la navegación y el acoplamiento, es que este robot es no holonómico [27], lo que en nuestro caso, al tener el robot dos ruedas, implica que únicamente puede desplazarse hacia delante o hacia atrás y rotar sobre si mismo en el eje *yaw*, que ya mencionamos en el apartado 4.2.2. Por tanto, si queremos que el robot se mueve hacia derecha o izquierda, primero se debe realizar una rotación y, posteriormente, una traslación.

Una vez conocido el robot con el que vamos a desarrollar la navegación, conviene examinar también el entorno creado en el que realizaremos la simulación. Como comentamos en apartados anteriores este entorno ha sido generado mediante gazebo y es el que podemos ver en la figura 4.12.

[6] `https://emanual.robotis.com/docs/en/platform/turtlebot3/overview/`

Figura 4.12: Entorno en el que se realiza la simulación en gazebo

Fuente:`https://github.com/aws-robotics/aws-robomaker-small-house-world.git`

Aclarados estos detalles, si nos centramos en el entorno, podemos ver que, además del mobiliario generado, nos encontramos con los 3 elementos claves en la simulación: el robot, el código QR y la estación de carga, que podemos ver remarcados en la figura 4.13.

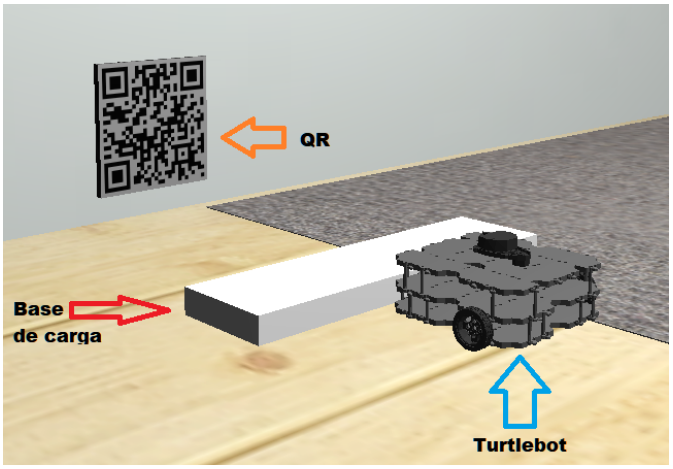


Figura 4.13: Elementos clave de la simulación. La flecha azul muestra el robot, la roja la estación de carga y la naranja el código QR.

Teniendo todo esto en cuenta, se realizará la simulación de la siguiente forma:

1. En función de la posición ya conocida del robot en el entorno simulado, se procederá a realizar la navegación, si el robot no se encuentra en una zona apropiada para poder realizar el acoplamiento correctamente, o el acoplamiento en su base de carga si se encuentra lo suficientemente cerca de esta.

2. Si se realiza la navegación, el robot navegará hasta un punto que denominaremos " zona de acoplamiento " y que el robot deberá conocer previamente.

3. Tras realizar la navegación, o si el robot se encontraba desde un inicio en esa zona, se procede a realizar el acoplamiento.

4. Si el robot no detecta el código QR, inicia una rotación sobre su eje hasta ser capaz de detectarlo.

5. Una vez detectado el QR, mediante los vectores de traslación (T) y de rotación (R) obtenidos al descomponer la homografía calculada en la etapa anterior, haremos girar y/o avanzar en función a estos mediante el envío de comandos de velocidad al robot.

6. Si el acoplamiento se ha completado correctamente, se termina el proceso. En caso contrario, volvemos a repetir el paso de la homografía en la posición actual del robot.

Este proceso lo podemos ver de una forma más esquematizada mediante el diagrama de flujos de la figura 4.14.

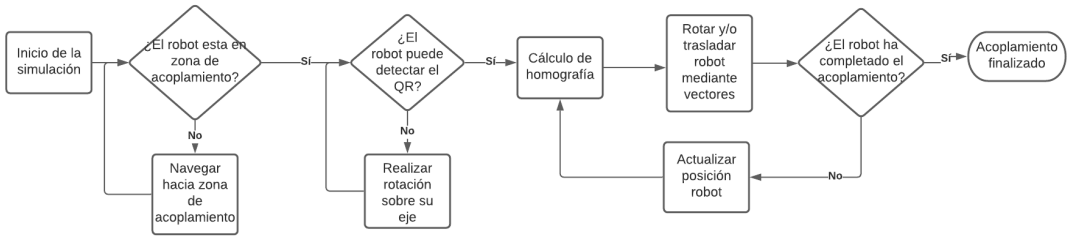


Figura 4.14: Diagrama de flujo que clarifica el proceso seguido en la simulación.

Capítulo 5

Pruebas y resultados

En esta sección se comentarán los resultados obtenidos en la simulación propuesta, así como las diferentes pruebas realizadas con el fin de ver el rendimiento del método aplicado.

5.1. Detección de códigos QR

Como se mencionó en el apartado 4.2.1, en este proyecto se han probado dos librerías diferentes para realizar la detección de nuestro patrón, es decir, del código QR. Sin embargo, existen más librerías además de las mencionadas que también pueden realizar esta labor. Por ello, en este apartado analizaremos y discutiremos la comparativa que se realiza en [1], para dotar al trabajo de una mayor completitud.

Además de **zbar** y **OpenCV**, en esta comparativa se analizan, adicionalmente, las librerías **BoofCV, Quirc** y **ZSxing**. Para realizar esta comparativa, se aplican estas librerías en imágenes modificadas de distintas formas, viendo como afecta estas modificaciones al rendimiento. Las categorías en las que se divide cada modificación son:

- **Blurred:** La imagen puede estar borrosa debido al enfoque o al movimiento de la cámara.

- **Brightness:** La misma imagen se compara en situaciones con brillo y sin brillo.
- **Bright Spots:** Cambios abruptos en el brillo de determinadas zonas de la imagen.
- **Close:** Imagen tomada a una distancia demasiado cercana.
- **Curved:** El QR se encuentra en superficies no planas.
- **Damaged:** El QR posee errores de impresión, arañazos o esta dañado de forma general.
- **Glare:** Se observan destellos que afectan al QR.
- **High Version:** Códigos QR que contienen una gran cantidad de datos en su interior.
- **Monitor:** Imagen tomada desde un monitor o pantalla. Cuanto más cerca se realice, más píxeles serán visibles.
- **Lots:** Imagen en la que aparecen más de un código QR a la vez.
- **Nominal:** Caso en el que se intenta representar al código QR sin ninguna alteración.
- **Non-Compilant:** El código QR es alterado con un fin artístico o corporativo.
- **Pathological:** Códigos QR diseñados intencionalmente con el fin de romper el detector.
- **Perspective:** Código QR visto desde distintos ángulos.
- **Rotated:** El código QR se encuentra rotado un cierto ángulo con respecto a su posición original.
- **Shadows:** La iluminación no es uniforme en la imagen debido a la aparición de sombras.

Habiendo puesto en contexto las pruebas que se han realizado sobre el conjunto de datos del que se ha hecho uso en [1], a continuación, analizaremos y discutiremos los resultados que se han obtenido en las diferentes categorías para cada librería y, posteriormente, el tiempo de ejecución de cada una.

Como podemos observar en la figura 5.1, en lineas generales no existe una librería que domine en todas las categorías. BoofCV es de las mejores en la mayoría de las categorías, excepto en *Non-Compilant* y *Pathological*. A su vez, zbar también obtiene buenos resultados en lineas generales, mientras Zxing muestra grandes resultados solo en algunas categorías concretas como *lots*. Quirc se desempeña bien en situaciones comunes, aunque no funciona tan bien en otras situaciones donde el resto de librerías si lo hace. Por ultimo, OpenCV necesita de un gran trabajo interno para realizar la detección, de ahí sus resultado. Además, el solo poder detectar de manera simultanea un QR no le beneficia. En la figura 5.2 podemos ver resumidas las puntuaciones de cada librería.

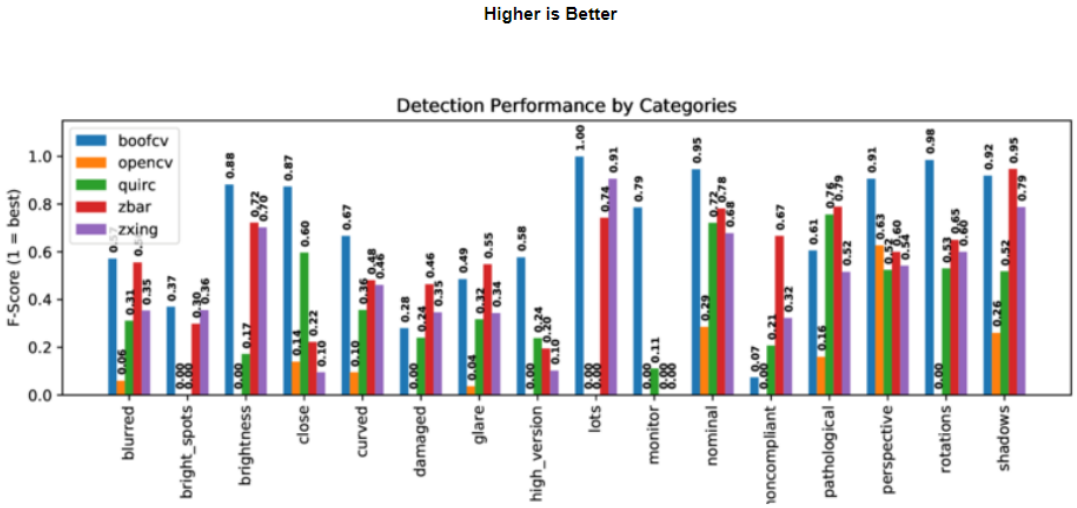


Figura 5.1: Gráfico que compara la detección del QR en cada librería en cada una de las distintas categorías.

Fuente: `https://boofcv.org/index.php?title=Performance:QrCode`

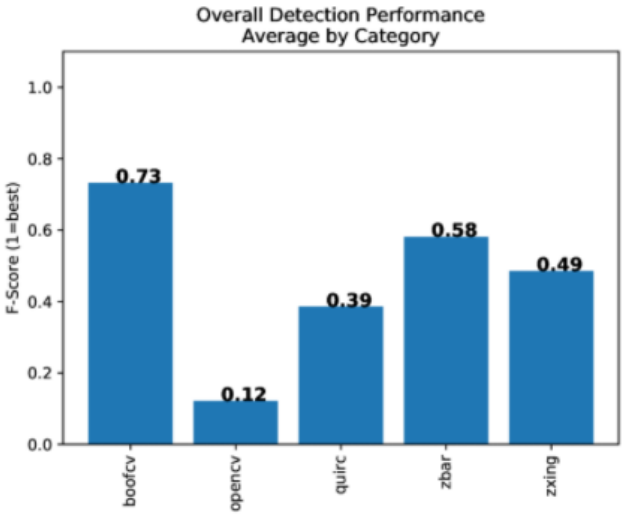


Figura 5.2: Gráfico que resume el desempeño medio en la detección de cada librería.

Fuente: `https://boofcv.org/index.php?title=Performance:QrCode`

Pasando ahora a los resultados en los tiempos de ejecución, podemos ver que, según las figuras 5.3 y 5.4, los más rápidos son Zxing y BoofCV respectivamente, teniendo un amplio margen sobre Quirc. OpenCV tiene un desempeño medio y zbar es la más lenta de todas con diferencia. Algo interesante que podemos observar, es que las más rápidas son librerías implementadas en Java mientras que las más lentas están implementadas en C/C++. Esto puede deberse a detalles algorítmicos de las librerías, como se menciona en [1].

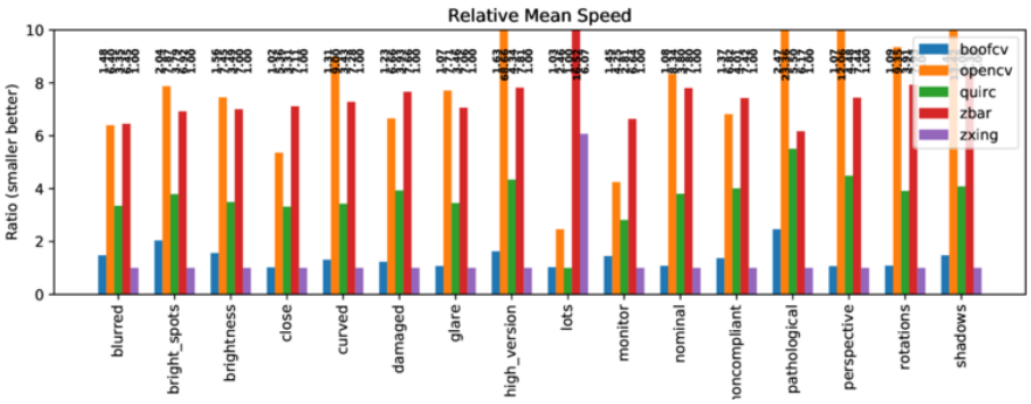


Figura 5.3: Gráfico que compara el tiempo de ejecución en cada librería en cada una de las distintas categorías.

Fuente: `https://boofcv.org/index.php?title=Performance:QrCode`

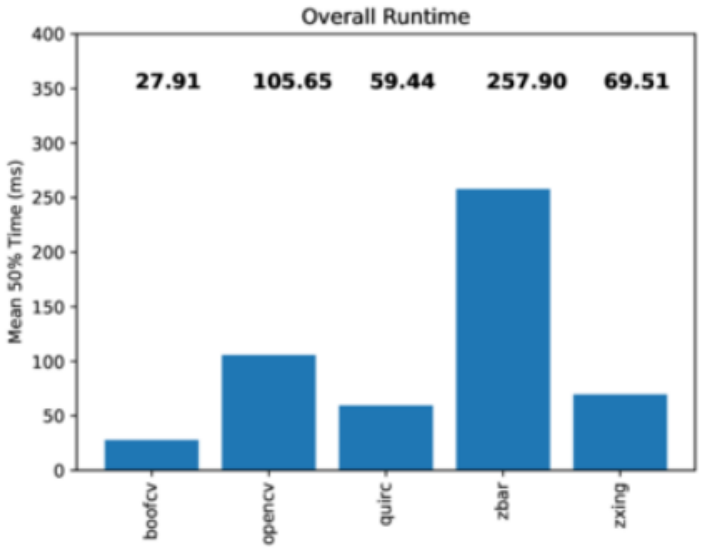


Figura 5.4: Gráfico que resume el rendimiento general de cada librería con respecto a sus tiempos de ejecución (cuanto menos puntuación, mejor rendimiento).

Fuente: `https://boofcv.org/index.php?title=Performance:QrCode`

5.2. Navegación y acoplamiento

Como ya mencionamos anteriormente, para la navegación y el acoplamiento utilizaremos de forma conjunta ROS y Gazebo. Con el entorno ya configurado, se han creado dos nodos en ROS con el fin de realizar la navegación y el acoplamiento, a los que hemos nombrado **Navigation** y **Docking** respectivamente.

Como ya se explicó en el apartado 4.2.4, dependiendo de la ubicación en la que se encuentre nuestro robot, se inicializara un nodo u otro. Si iniciamos el nodo de navegación, el robot se desplazará hasta un punto en el que sea capaz de detectar el QR y, por tanto, iniciar el acoplamiento automático. Este punto se le debe suministrar previamente al robot a partir de la localización del QR. Al iniciar el nodo de acoplamiento, el robot girará sobre si mismo hasta que detecte el QR, momento en el que empezará a recibir los comandos de movimiento pertinentes a partir de la homografía hasta que se complete el acoplamiento.

Comenzando por el nodo de navegación, este funciona de la siguiente manera. En primer lugar, se obtiene la localización actual del robot y se calcula la diferencia entre esta y la posición a la que queremos llegar para empezar a realizar el acoplamiento. Tras esto, obtenemos el ángulo en el que debe rotar el robot para realizar la navegación hasta este punto a partir de la diferencia anteriormente calculada. Posteriormente, el robot comienza a rotar sobre si mismo hasta que la diferencia entre el ángulo en el que nos debemos mover y el ángulo en el que se encuentra el robot sea menor a un umbral que estableceremos. En ese momento, el robot se desplazará hacia delante, repitiendo este proceso hasta que la diferencia entre el punto al que se debe desplazar y en el que se encuentra el robot también sea menor a un umbral dado. Estos umbrales se utilizan para intentar minimizar errores que se puedan dar en el desplazamiento y que pueden provocar que el punto exacto sea difícilmente alcanzable. En la figura 5.5 podemos ver un ejemplo del trayecto recorrido por el robot con RViz[1] y el módulo *see_path.py*.

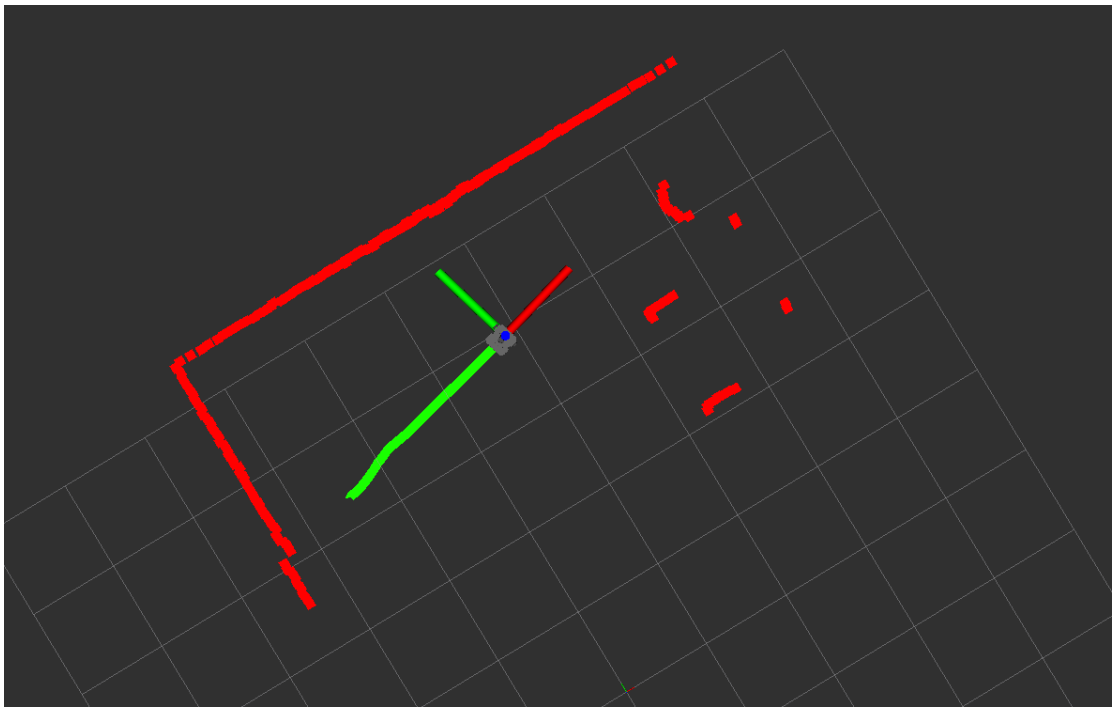


Figura 5.5: Escenificación mediante la herramienta RViz. Se puede observar la posición del robot mediante los ejes, en verde el trayecto realizado y ,en rojo, los obstáculos detectados por el sensor del robot.

[1] `http://wiki.ros.org/rviz`

Este nodo de navegación ha sido probado en un entorno sin obstáculos, obteniendo unos tiempos de ejecución de entre 10-16 segundos aproximadamente partiendo el robot con un ángulo favorable[2] (ver figura 5.6) y unos tiempos entre 19.5-25.5 segundos partiendo el robot con un ángulo desfavorable (ver figura 5.7).

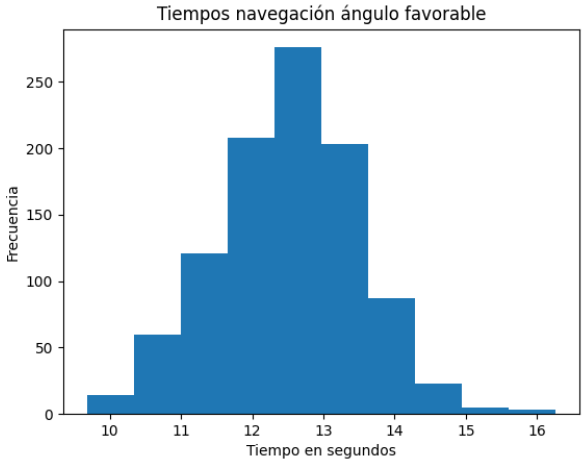


Figura 5.6: Histograma con los tiempos computados para la navegación con un ángulo favorable. Como se puede observar, el tiempo medio ronda los 12,5 segundos aprox.

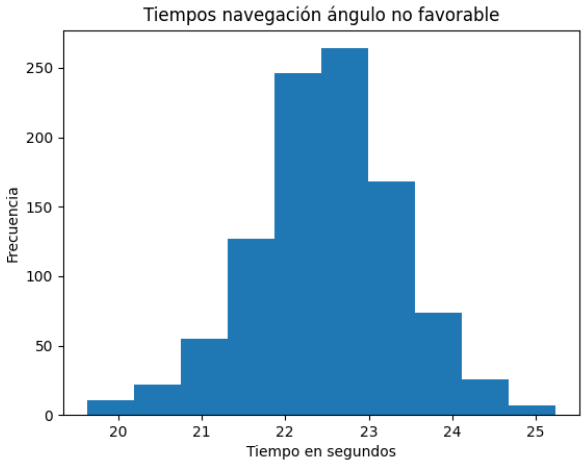


Figura 5.7: Histograma con los tiempos computados para la navegación con un ángulo no favorable. Como se puede observar, el tiempo medio ronda los 22,5 segundos aprox.

[2]Nos referimos por ángulo favorable a un ángulo que permita dejar al robot frente al objetivo, y desfavorable estando de espaldas al objetivo.

Todos estos tiempos se han tomado considerando que el robot se encontraba a una distancia aproximada de 1 a 2 metros del punto especificado.

A continuación, pasaremos a analizar el nodo de acoplamiento. Como ya se ha mencionado anteriormente, cuando se ejecuta este nodo, el robot debe estar a una distancia que le permita detectar correctamente el código QR, para lo cual hemos implementado el ya mencionado nodo de navegación. Una vez nos encontramos a una distancia adecuada, si el robot no tiene el código QR en su rango de visión, comienza a rotar hasta encontrarlo, momento en el que se calcula la homografía entre la vista actual del robot y la vista que tendría el robot cuando realiza el acoplamiento (ver figura 5.8).

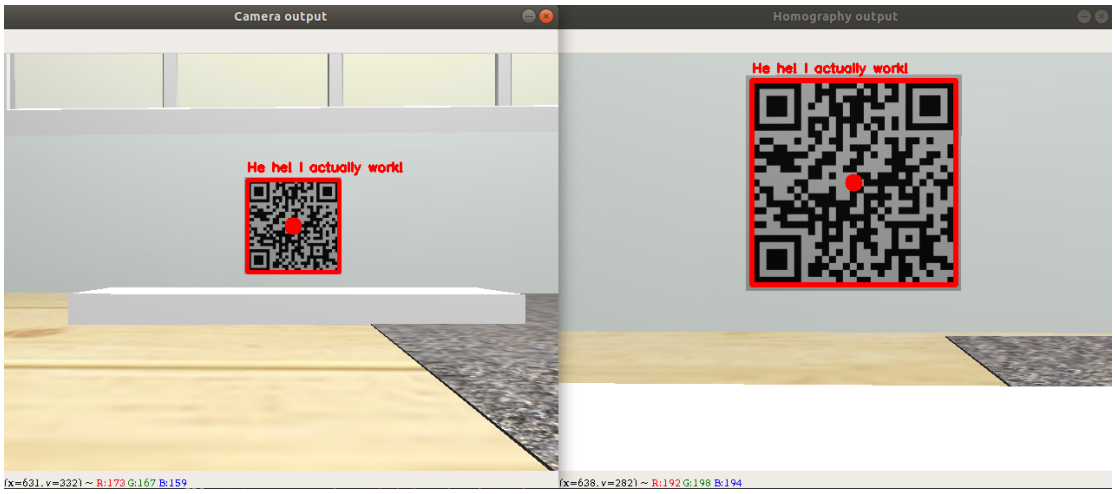


Figura 5.8: Vistas desde la cámara del robot durante el acoplamiento. A la izquierda,podemos ver la vista en el momento actual y, a la derecha, la vista que debería tener cuando el acoplamiento esté completado.

Con la homografía ya calculada, obtenemos los vectores de rotación y traslación para ver la diferencia existente entre ambas imágenes. Si el vector de rotación es mayor que un umbral establecido, el robot seguirá rotando, aunque a una velocidad menor a la establecida cuando el robot no detecta el QR, con el fin de no realizar cambios bruscos y que impidan al robot alcanzar este umbral. Del mismo modo, si el vector de traslación es superior a otro umbral dado, el robot avanzará hacia adelante a una velocidad reducida por el mismo motivo explicado anteriormente. Este proceso se realizará de manera cíclica hasta que ambas vistas coincidan (ver

figura 5.9), momento en el cual el robot se detendrá y se habrá dado por terminado el proceso de acoplamiento. En la figura 4.13 se muestra un ejemplo de la posición final del robot (acoplado a la base de carga) tras completarse dicho proceso.

Figura 5.9: Vistas desde la cámara del robot con el acoplamiento finalizado. Como podemos observar, ambas imágenes son casi idénticas.

Como con el nodo de navegación, este nodo ha sido probado con robot partiendo tanto desde un ángulo favorable como desfavorable. En el caso en el que parte desde un ángulo desfavorable, se han obtenido unos tiempos de ejecución de entre 32 y 38 segundos aprox (ver figura 5.10) mientras que, partiendo desde un ángulo favorable, se reduce a unos tiempos entre 13.5 y 16.5 segundos aprox (ver figura 5.11).

Cabe destacar que el nodo de navegación suele dejar al robot en una posición con un ángulo cercano a uno favorable, pues en este nodo se calcula el ángulo con el que el robot se debe desplazar, como ya se comentó anteriormente, por lo que al usar ambos nodo de manera conjunta, el tiempo en el acoplamiento sería más cercano al del ángulo favorable.

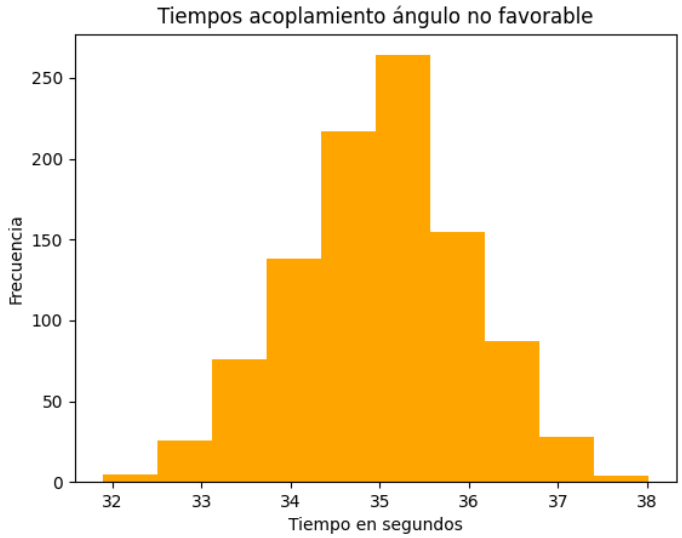


Figura 5.10: Histograma que muestra la distribución de los tiempos del nodo de acoplamiento con un ángulo desfavorable. Podemos observar que el tiempo medio es de 35 segundos aprox.

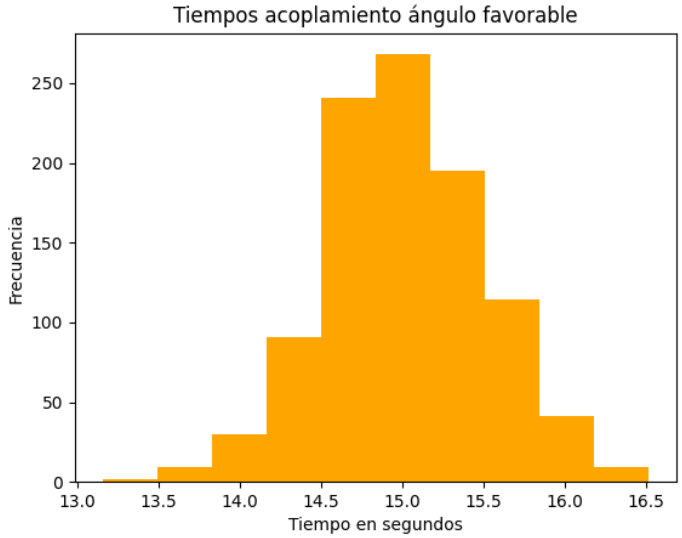


Figura 5.11: Histograma que muestra la distribución de los tiempos del nodo de acoplamiento con un ángulo favorable. Podemos observar que el tiempo medio está en 15 segundos aprox.

Tanto estas pruebas como las realizadas en el nodo de navegación han sido realizadas a partir de 1000 iteraciones cada una.

Capítulo 6

Conclusiones y líneas futuras

En este trabajo de fin de grado, se ha presentado un nuevo método para llevar a cabo el acoplamiento de un robot en su base de carga. Este método utiliza las homografías para llevar a cabo este procedimiento, herramienta ampliamente utilizada en la Visión por Computador y con una gran variedad de aplicaciones.

Para realizar esta implementación, la aplicación se ha divido en diferentes módulos que son necesarios para desarrollar las diferentes fases por las que debe pasar el robot para poder realizar el acoplamiento de forma correcta. En primer lugar, se ha lidiado con la tarea de reconocimiento de patrones, siendo en nuestro caso el patrón un código QR. Se han estudiado los distintos métodos de detección de códigos QR, así como sus respectivas librerías, viendo las ventajas y desventajas de cada una de ellas.

Seguidamente, se ha introducido de manera resumida el problema de la calibración de las cámaras, que es necesario resolver para poder utilizar de manera adecuada la cámara en nuestro robot.

A continuación, se ha seguido con la tarea referente a las homografías, que como se ha visto en anteriores apartados, son transformaciones entre dos imágenes diferentes referidas a un mismo plano. Con esta homografía, se obtiene la matriz de

homografía, que nos ha permitido establecer una correspondencia entre los puntos del código QR que detecta la cámara del robot en tiempo real y los puntos del QR cuando el robot se encuentra acoplado en su estación de carga. El mayor inconveniente a la hora de trabajar con esta herramienta llega a la hora de realizar la descomposición de la matriz de homografía para obtener los vectores de rotación y traslación, pues para cada matriz existen hasta 4 vectores posibles. En un principio, esto puede hacer inservible este método, pero mediante las técnicas explicadas en el apartado 4.2.3, es posible reducir este número de posibles vectores a un máximo de 2, lo cual hace mucho asequible este método.

Finalmente, se han aplicado todos estos conceptos en el robot simulado mediante ROS y Gazebo, comprobando el correcto funcionamiento de los métodos expuestos y analizando su rendimiento.

6.1. Posibles extensiones y vías futuras

Aunque tanto la detección como el acoplamiento funcionan de manera correcta, todavía existen ciertos aspectos que pueden mejorarse en vistas al futuro:

- La posibilidad de dotar al nodo de navegación con la funcionalidad de detectar y evadir obstáculos, ya que esta implementación esta pensada en un entorno ideal sin estos, pero en la realidad, existen multitud de elementos que pueden dificultar la tarea de este nodo. Esto se puede subsanar fácilmente usando el nodo de navegación que implementa el propio ROS[1].

- Dotar al robot de un sistema holonómico o, en su defecto, utilizar un robot con mayor movilidad, puesto que, como se ha mencionado en el apartado 4.2.4, el robot posee limitaciones de movimiento, pudiendo únicamente desplazarse hacia delante, hacia atrás o rotar sobre si mismo, lo que hace imposible, por ejemplo, desplazarse hacia la izquierda o la derecha sin realizar ninguna rotación previa.

[1] `http://wiki.ros.org/navigation`

Apéndice A

Instalación de librerías en Python

En este apéndice, se procederá a explicar el proceso de instalación de las diferentes librerías que se han usado en el proyecto y como utilizarlas una vez instaladas. El proceso, aunque simple, merece un aparte en la memoria, puesto que sin ser capaces de instalar estas librerías, no podríamos realizar este proyecto.

La librería principal que instalaremos en primer lugar será **OpenCV**, que hemos nombrado a lo largo de todo el documento. Cabe destacar que OpenCV proporciona dos versionas diferentes, una versión simple con los módulos básicos y otras versión más completa con varios módulos adicionales, conocida como *contrib*.

En nuestro caso, instalaremos la versión *contrib*, lo cual, haremos ejecutando las siguientes instrucciones en una consola de comandos o terminal:

```
pip install opencv-contrib-python
```

También es recomendable instalar las siguientes librerías de Python, ya que facilitan la utilización de OpenCV y se complementan con esta. Estas librerías son **Numpy**[1], **Matplotlib**[2] y, de manera similar a OpenCV, se instalan con las siguientes instrucciones:

[1] https://numpy.org/
[2] https://matplotlib.org/

```
pip install numpy
pip install matplotlib
```

Tras OpenCV, instalaremos la otra librería que se ha usado en la detección de códigos QR, es decir, **zbar**[3]. Aunque la librería original es zbar, al trabajar nosotros con Python y estar escrita originalmente en C++, usaremos la versión **pyzbar**, que es la operativa para versiones de Python 3 o superiores (no obstante, se puede trabajar con zbar en Python 2 si se desea).

El proceso de instalación es idéntico a los anteriores, teniendo que usar las siguientes instrucciones:

```
pip install pyzbar
```

Con esto, ya tenemos todas las librerías necesarias para poder realizar el proyecto de acuerdo a como está descrito en la memoria.

[3] `https://pypi.org/project/pyzbar/`

Apéndice B

Instalación y configuración de ROS y Gazebo

En este apéndice, se explicará el proceso de instalación a seguir para utilizar tanto ROS como Gazebo. Para ello, nos basaremos en el tutorial que se detalla en la página de ROS[1]. Cabe destacar que ROS está pensando para ser usado en un entorno basado en Linux, siendo en nuestro caso Ubuntu la distribución elegida.

En función de la versión de Ubuntu que se use, tendremos que elegir una versión de ROS que le corresponda. En nuestro caso, al estar usando Ubuntu 18.04, usaremos la versión Melodic de ROS, si bien es recomendable utilizar Noetic[2] (Ubuntu 20.04) o superiores por temas de soporte de actualizaciones.

Para instalar ROS Melodic, deberemos seguir los siguientes pasos:

1. Configurar el archivo *sources.list* para aceptar los programas que provengan de la páginas de ROS mediante:

```
sudo sh -c 'echo "deb http://packages.ros.org/ros/
   ubuntu $(lsb_release -sc) main" > /etc/apt/sources.
   list.d/ros-latest.list'
```

[1] http://wiki.ros.org/melodic/Installation/Ubuntu
[2] http://wiki.ros.org/noetic

2. Configurar las claves mediante:

```
sudo apt install curl
curl -s https://raw.githubusercontent.com/ros/
   rosdistro/master/ros.asc | sudo apt-key add -
```

3. A continuación, comenzamos el proceso de instalación. Para ello, existen diferentes versiones que incluyen más o menos herramientas adicionales en función del uso que vayamos a darle a ROS. En nuestro caso, instalaremos la versión completa, que a su vez también incluye la instalación de Gazebo. Esto lo hacemos mediante:

```
sudo apt install ros-melodic-desktop-full
```

4. Una vez instalado, solo queda actualizar las variables de entorno del sistema para poder detectar tanto ROS como Gazebo. Para ello usamos las siguientes instrucciones:

```
echo "source /opt/ros/melodic/setup.bash" >> ~/.bashrc
source ~/.bashrc
```

Con la instalación ya finalizada, solo nos resta configurar el entorno de trabajo en el que realizaremos nuestro proyecto. Para crear un entorno de trabajo en ROS, lo más sencillo es crear un entorno de trabajo **catkin**[3]. Esto lo haremos con las siguientes instrucciones:

```
mkdir -p ~/catkin_ws/src
cd ~/catkin_ws/
catkin_make
```

Con esto, y tras actualizar nuevamente las variables de entorno como en el paso 4 del proceso de instalación de ROS, ya estamos listos para trabajar tanto con ROS como con Gazebo.

[3] http://wiki.ros.org/catkin/workspaces

Bibliografía

[1] Peter Abeles. Study of qr code scanning performance in different environments. v3. `https://boofcv.org/index.php?title=Performance:QrCode`.

[2] Luiz Belussi and Nina Hirata. Fast qr code detection in arbitrarily acquired images. In *2011 24th SIBGRAPI Conference on Graphics, Patterns and Images*, pages 281–288. IEEE, 2011.

[3] Jules Bloomenthal and Jon Rokne. Homogeneous coordinates. *The Visual Computer*, 11(1):15–26, 1994.

[4] G. Bradski. The OpenCV Library. *Dr. Dobb's Journal of Software Tools*, 2000.

[5] Ron Brinkmann. *The art and science of digital compositing: Techniques for visual effects, animation and motion graphics*. Morgan Kaufmann, 2008.

[6] AM Burgueño-Romero, JR Ruiz-Sarmiento, and J Gonzalez-Jimenez. Autonomous docking of mobile robots by reinforcement learning tackling the sparse reward problem. In *International Work-Conference on Artificial Neural Networks*, pages 392–403. Springer, 2021.

[7] A. M. Burgueño, J. R. Ruiz-Sarmiento, and Javier Gonzalez-Jimenez. *Autonomous Docking of Mobile Robots by Reinforcement Learning Tackling the Sparse Reward Problem*, volume 12862 of *Lecture Notes in Computer Science*, chapter Advances in Computational Intelligence, pages 392–403. Springer International Publishing, Cham, 2021.

[8] LV Calderita, P Bustos, C Suárez Mejías, F Fernández, R Viciana, and Antonio Bandera. Asistente robótico socialmente interactivo para terapias de rehabilitación motriz con pacientes de pediatría. *Revista Iberoamericana de Automática e Informática industrial*, 12(1):99–110, 2015.

[9] John Canny. A computational approach to edge detection. *IEEE Transactions on pattern analysis and machine intelligence*, (6):679–698, 1986.

[10] Alistair Cockburn. Using both incremental and iterative development. *STSC CrossTalk (USAF Software Technology Support Center)*, 21(5):27–30, 2008.

[11] Navneet Dalal and Bill Triggs. Histograms of oriented gradients for human detection. In *2005 IEEE computer society conference on computer vision and pattern recognition (CVPR'05)*, volume 1, pages 886–893. Ieee, 2005.

[12] Javier González. Visión por computador. *ITP Paraninfo*, 1999.

[13] Javier González-Jiménez, Cipriano Galindo, and JR Ruiz-Sarmiento. Technical improvements of the giraff telepresence robot based on users' evaluation. In *2012 IEEE RO-MAN: The 21st IEEE International Symposium on Robot and Human Interactive Communication*, pages 827–832. IEEE, 2012.

[14] Chris Harris, Mike Stephens, et al. A combined corner and edge detector. In *Alvey vision conference*, volume 15, pages 10–5244. Citeseer, 1988.

[15] David G Lowe. Object recognition from local scale-invariant features. In *Proceedings of the seventh IEEE international conference on computer vision*, volume 2, pages 1150–1157. Ieee, 1999.

[16] Ezio Malis and Manuel Vargas. *Deeper understanding of the homography decomposition for vision-based control*. PhD thesis, INRIA, 2007.

[17] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A Rusu, Joel Veness, Marc G Bellemare, Alex Graves, Martin Riedmiller, Andreas K Fidjeland, Georg Ostrovski, et al. Human-level control through deep reinforcement learning. *nature*, 518(7540):529–533, 2015.

[18] Morgan Quigley, Ken Conley, Brian Gerkey, Josh Faust, Tully Foote, Jeremy Leibs, Rob Wheeler, Andrew Y Ng, et al. Ros: an open-source robot operating system. In *ICRA workshop on open source software*, volume 3, page 5. Kobe, Japan, 2009.

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

[20] Roberto Quilez, Adriaan Zeeman, Nathalie Mitton, and Julien Vandaele. Docking autonomous robots in passive docks with infrared sensors and qr codes. In *International Conference on Testbeds and Research Infrastructures for the Development of Networks & Communities (TridentCOM)*, 2015.

[21] Kun Shao, Zhentao Tang, Yuanheng Zhu, Nannan Li, and Dongbin Zhao. A survey of deep reinforcement learning in video games. *arXiv preprint arXiv:1912.10944*, 2019.

[22] Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction*. MIT press, 2018.

[23] Carlos Ricolfe Viala, Antonio José Sánchez Salmerón, and Raúl Simarro Fernández. Técnicas de calibrado de cámaras. *Journal de Mathematiques Pures et Appliques, Leon*, 2003.

[24] Paul Viola and Michael Jones. Rapid object detection using a boosted cascade of simple features. In *Proceedings of the 2001 IEEE computer society conference on computer vision and pattern recognition. CVPR 2001*, volume 1, pages I–I. Ieee, 2001.

[25] Yuanzhe Wang, Mao Shan, Yufeng Yue, and Danwei Wang. Autonomous target docking of nonholonomic mobile robots using relative pose measurements. *IEEE Transactions on Industrial Electronics*, 68(8):7233–7243, 2020.

[26] Eric W Weisstein. Euler angles. *https://mathworld. wolfram. com/*, 2009.

[27] Wikipedia. Holónomo (robótica) — wikipedia, la enciclopedia libre, 2019. [Internet; descargado 11-noviembre-2021].

[28] Wikipedia. Robótica educativa — wikipedia, la enciclopedia libre, 2021. [Internet; descargado 21-noviembre-2021].

[29] Wikipedia. Roomba — wikipedia, la enciclopedia libre, 2021. [Internet; descargado 21-noviembre-2021].

[30] Wikipedia contributors. Haar-like feature — Wikipedia, the free encyclopedia, 2021. [Online; accessed 15-November-2021].

[31] Zhengyou Zhang. A flexible new technique for camera calibration. *IEEE Transactions on pattern analysis and machine intelligence*, 22(11):1330–1334, 2000.

UNIVERSIDAD DE MÁLAGA | uma.es

E.T.S de Ingeniería Informática
Bulevar Louis Pasteur, 35
Campus de Teatinos
29071 Málaga

E.T.S. DE INGENIERÍA INFORMÁTICA