Universidad Zaragoza
1542

Trabajo Fin de Grado

Navegación autónoma de un robot móvil basada en marcas ArUco

Autonomous navigation of a mobile robot based on ArUco markers

Autor

Alba Erdociaín Herrero

Directores

Gonzalo López Nicolás

Rafael Herguedas Gastón

ESCUELA DE INGENIERÍA Y ARQUITECTURA
2021

AGRADECIMIENTOS

A mis directores, Gonzalo y Rafael por haberme dejado realizar este proyecto. Gracias por la dedicación y la paciencia que habéis tenido durante todos estos meses conmigo.

Navegación autónoma de un robot móvil basada en marcas ArUco

RESUMEN

En la actualidad, los robots móviles son de gran ayuda en las empresas ya que son capaces de realizar de manera autónoma ciertas tareas repetitivas, permitiendo así que los operarios ejerzan trabajos más productivos.

Por ello, el objetivo principal es obtener un algoritmo que permita que un robot móvil siga de manera autónoma una marcar artificial. La marca artificial elegida, montada sobre un sistema móvil o directamente manejada por una persona, se basa en el sistema ArUco. Este tipo de sistema puede ayudar en una empresa a que los robots transporten cargas de un lugar a otro siguiendo dichas marcas. En caso de que se quisiese seguir a un empleado se podría utilizar este algoritmo y añadirle mediante visión por computación la detección de personas.

Para poder cumplir este objetivo principal se han ido realizando tareas más sencillas. Primero, se ha instalado Ubuntu y el paquete de ROS (Robot Operating System). Segundo, se ha realizado un estudio de este nuevo entorno que es ROS y las herramientas que contiene, como son RViz y Gazebo. Tercero, se han leído artículos de control con la finalidad de obtener los conocimientos necesarios para desarrollar el algoritmo. Cuarto, se ha desarrollado el algoritmo en el entorno de Matlab y se han realizado distintas simulaciones con distintos valores de las ganancias hasta obtener los valores deseados. Finalmente, se ha programado dicho algoritmo en lenguaje Python, que será el utilizado en el entorno ROS, y se han llevado a cabo las simulaciones para distintas trayectorias con la finalidad de comprobar el correcto funcionamiento del algoritmo.

Índice

Capítulo 1

Introducción

1.1. Motivación

En la actualidad la robótica está teniendo un gran impacto en la sociedad. Esto se puede ver en que, en los inicios la robótica tenía un uso principalmente industrial, sin embargo, actualmente podemos contemplar dos grandes usos: industrial y de servicios.

Los robots industriales son los más conocidos y los que mayor tiempo llevan en el mercado. Estos se pueden encontrar principalmente en las fábricas de automoción y en aquellas fábricas en las que los trabajos son repetitivos o pesados.

La robótica de servicio es aquella que ha sido creada para ser utilizada fuera de las instalaciones industriales. Esta a su vez la podemos clasificar en profesionales y de uso personal. El primer grupo serían aquellos robots que ayudan en las labores médicas, docentes... y el segundo grupo serían los que se usan en nuestro domicilio, como pueden ser las aspiradoras o los limpiadores de piscina.

Otra clasificación que encontramos dentro de los robots autónomos son robots manipuladores fijos y robots manipuladores móviles, como el de la figura 1.1. Los primeros de ellos fueron los que aparecieron inicialmente y están orientados a ayudar en empresas en las que hay que mover cargas pesadas, pero sin llevarlas de un lugar de la nave a otro. Sin embargo, los segundos permiten desplazamientos de un lugar a otro, ya que disponen de ruedas. También nos podemos encontrar con robots móviles pero que no son manipuladores, como sería el caso de un aspirador.

En nuestro caso, el trabajo se centra en el desarrollo de un algoritmo que permita al robot realizar un seguimiento de una marca ArUco, para ello se utiliza un robot manipulador móvil. Esta aplicación estaría pensada para los robots industriales, ya que podría ayudar en empresas a transportar materiales pesados a lo largo de una nave siguiendo una serie de marcas ArUco, o bien se podría hacer que siguiese a personas mediante un reconocimiento visual con técnicas de visión por computación.

El robot que se utilizará recibe el nombre de robot Campero, es un prototipo del modelo comercial "Manipulador móvil RB-EKEN"[1]. Este modelo dispone de un gran número de sensores y una cámara que nos ayudará a poder llevar a cabo el seguimiento de la marca.

Figura 1.1: Robot manipulador móvil en industria.
Fuente: `https://img.interempresas.net/fotos/2608004.jpeg`

1.2. Contexto

Este trabajo es la continuación de otro trabajo de fin de grado, 'Navegación autónoma de robot manipulador móvil con cámara y láser en el entorno ROS'[2]. Ambos trabajos forman parte del proyecto COMMANDIA[3].

COMMANDIA es un proyecto financiado por el Fondo Europeo de Desarrollo Regional (FEDER) y por el Programa Interreg Sudoe. El nombre del proyecto significa Robótica móvil colaborativa de objetos deformables en aplicaciones industriales.

Con respecto al robot que vamos a utilizar, es un prototipo del modelo comercial "Manipulador móvil RB-EKEN"de la empresa Robotnik. Algunas de las características principales son las que se muestran a continuación:

- Dimensiones: 1.204 x 730 x 991 mm. Estas dimensiones son con el brazo plegado
- Peso : 270 kg
- Ruedas : Dispone de dos tipos de ruedas, para exterior ruedas de goma y para interior ruedas mecanum
- Velocidad máxima : 2 m/s
- Carga máxima que puede transportar : 300 Kg

- Autonomía : 4 horas

Para más información sobre este modelo consultar la página de Robotnik[1].

Figura 1.2: Modelo RB-EKEN de Robotnik.
Fuente: `https://robotnik.eu/es/productos/manipuladores-moviles/rb-eken/`

El robot campero (ver figura 1.2) también dispone de numerosos sensores, de láseres y de una cámara PTZ. Esta cámara será de gran utilidad para llevar a cabo este trabajo, ya que con ella se pueden visualizar en todo momento las marcas ArUco.

Como se muestra en la figura 1.3 las marcas ArUco tienen una forma cuadrada, en la que el fondo es blanco y el borde negro. El fondo blanco es un patrón que se identifica de forma exclusiva, mientras que el borde de color negro ayuda a que estas marcas sean localizadas con mayor facilidad. Con respecto al tamaño se puede elegir cualquiera, por lo tanto es muy importante elegir el adecuado para una visualización mucho más precisa.

Además, existen distintos tipos de marcas cada una de ellas pertenece a un diccionario distinto. En este caso se usará el diccionario original ArUco. La librería que detecta dichas marcas ha sido creada por un grupo de investigación de la Universidad de Córdoba [4] [5] [6].

Figura 1.3: Marca ArUco
Fuente: `https://docplayer.es/docs-images/114/210651247/images/33-1.jpg`

1.3. Objetivos

El principal objetivo de este trabajo de fin de grado es implementar un algoritmo de control de seguimiento para un robot móvil, basado en visión y en una marca artificial. Para conseguir dicho propósito se llevan a cabo una serie de tareas que ayuden a conseguir la meta. Estas tareas serán las siguientes:

1. La primera de las tareas a realizar será instalar Ubuntu 16.04, ya que trabajaremos con dicha distribución, de Linux. Además, deberemos descargar los paquetes de ROS.

2. La segunda tarea será familiarizarse con los conceptos de este nuevo entorno de trabajo que es ROS, así como las herramientas que se utilizarán para llevar a cabo la simulación del campero. Estas herramientas como se ha mencionado anteriormente son RViz y Gazebo.

3. Una vez nos hayamos familiarizado con ROS, se tendrá que obtener el algoritmo que nos permita llevar a cabo el seguimiento de una marca ArUco móvil. Para ello se tendrán que leer artículos sobre control.

4. Cuando se comprendan los conceptos importantes, como es la ley de control, deberemos obtener dicho controlador y para ello se usa el entorno de Matlab, el cual ya es conocido.

5. Una vez encontradas las variables necesarias para la ley de control, se pasará todo el código del lenguaje que se usa en Matlab al lenguaje Python, que será el que se utilizará para la implementación del control en la simulación del robot en ROS.

6. El último paso será desarrollar un fichero en lenguaje Python que recoja el control desarrollado anteriormente y se comprobará en simulación que el robot sigue perfectamente a la marca ArUco, la cual presenta un movimiento continuo.

7. Finalmente se sacarán las conclusiones y se redactará la memoria.

1.4. Estructura del documento

A continuación, se describe cómo se va a organizar la memoria del trabajo, el cual se divide en los siguientes capítulos.

- **Capítulo 1**: Se pone en contexto el trabajo que se va a desarrollar y se definen los objetivos.

- **Capítulo 2**: Se presentarán los dos entornos de trabajo que han sido utilizados para llevar a cabo este proyecto. Además, se mencionarán y explicarán brevemente dos de las herramientas que se han utilizado en el entorno de ROS.
- **Capítulo 3**: Se explicarán los fundamentos teóricos en los que se ha basado el desarrollo del trabajo.
- **Capítulo 4**: Se desarrollarán y explicarán detalladamente los algoritmos empleados en ambos entornos de trabajo.
- **Capítulo 5**: Se analizarán los resultados obtenidos en las distintas simulaciones, tanto en el entorno de Matlab como en ROS.
- **Capítulo 6**: Se comentarán las conclusiones obtenidas a lo largo del desarrollo del trabajo.
- **Bibliografía**
- **Anexos**

Capítulo 2

Entorno de trabajo

En este capítulo se mencionarán los dos entornos de trabajo que han sido utilizados para llevar a cabo el proyecto. Se comenzará explicando ROS y algunas de las herramientas de trabajo de las que dispone, y posteriormente será explicado Matlab.

2.1. Robot Operating System (ROS)

ROS (Robot Operating System) se conoce como, un meta sistema operativo. Esto quiere decir que no se puede considerar un sistema operativo como es el caso de Windows, ya que ROS se ejecuta dentro del sistema operativo Linux. Sin embargo, sí que presenta características muy similares a las de cualquier sistema operativo, como puede ser el control de dispositivos a bajo nivel y el manejo de paquetes.

Además, ROS es un software abierto y libre, bajo términos de licencia BSD [7]. Esto quiere decir que se puede encontrar en la red código de otras personas o empresas que hayan decidido dejarlo para que el resto de los usuarios lo utilicen, lo cual es muy ventajoso. De esta manera, las empresas se podrán especializar en realizar ciertas tareas despreocupándose de situaciones que ya han sido resueltas por otras personas. Todo esto hace que se fomente y facilite el acceso al conocimiento. Sin embargo, también existe la opción de vender el código. En este caso, si se han utilizado paquetes de otras personas estos permanecerán abiertos.

En la actualidad ROS va adquiriendo una mayor relevancia ya que cada vez son más el número de personas y empresas que lo utilizan para programar sus robots. Este entorno de trabajo se caracteriza por ser muy flexible y disponer de una gran variedad de herramientas, librerías y paquetes. Todo esto sirve para crear y diseñar robots robustos y con comportamientos variados. Para obtener más información sobre el meta sistema operativo consultar la página de ROS [8].

Para llevar a cabo este trabajo se ha utilizado la distribución ROS Kinetic Kame, ya que está dirigida especialmente a la versión de Ubuntu 16.04, que es la que se utiliza en este trabajo. Su fecha de lanzamiento fue el 23 de Mayo de 2016.

Figura 2.1: Logo de ROS
Fuente: `https://www.canonicalrobots.com/images/cursos/ros-logo.png`

A continuación, se explicarán dos de las herramientas que presenta ROS para poder llevar a cabo la simulación del seguimiento. Primero, se describirá Gazebo y segundo se explicará RViz.

2.1.1. Gazebo

Gazebo es un simulador destinado fundamentalmente al mundo de la robótica. Como se puede ver en la figura 2.3, este simulador es en tres dimensiones (3D) y en él se pueden representar numerosos robots así como los sensores que forman parte de ellos. En este trabajo, esta herramienta servirá para representar el entorno por el que el robot se moverá, siendo de gran utilidad a la hora de conocer si el robot sigue la trayectoria deseada.

En la página de Gazebo [9] se puede descargar la última versión, además se encuentran distintos tutoriales y proyectos que se han realizado con dicho simulador.

Figura 2.2: Logo Gazebo
Fuente del logo: `https://gazebosim.org/`

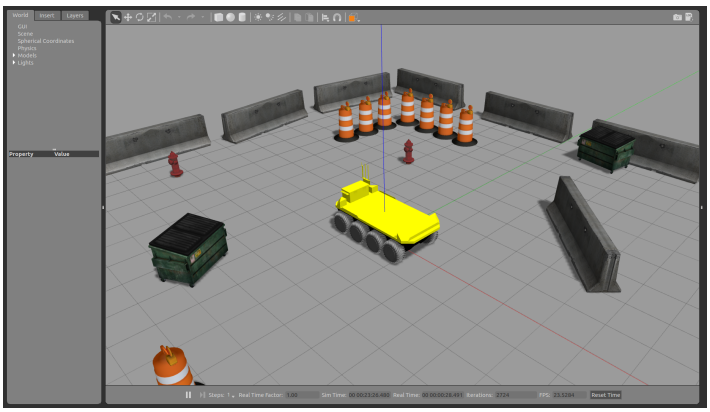

Figura 2.3: Ejemplo de un escenario en Gazebo
Fuente: `https://www.clearpathrobotics.com/assets/guides/kinetic/moose/MooseSimulation.html`

2.1.2. RViz

RViz es una interfaz en tres dimensiones, en nuestro caso aparecerá el robot con los sensores y la marca. Para este trabajo, dicha interfaz nos permitirá realizar diferentes operaciones, entre ellas tenemos la manipulación manual del robot y por lo tanto se podrá obtener un mapa del entorno por el que se mueve el robot. Otra de las operaciones será capturar imágenes con la cámara. Además, permitirá visualizar lo que miden sus sensores siendo de gran utilidad para el desarrollo de nuestro algoritmo.

En la siguiente página [10] se pueden encontrar los pasos a seguir para instalar RViz y algunos tutoriales para aprender conceptos básicos de dicha interfaz.

Figura 2.4: Logo de RViz
Fuente: `https://raw.githubusercontent.com/ros-visualization/rviz/melodic-devel/images/splash.png`

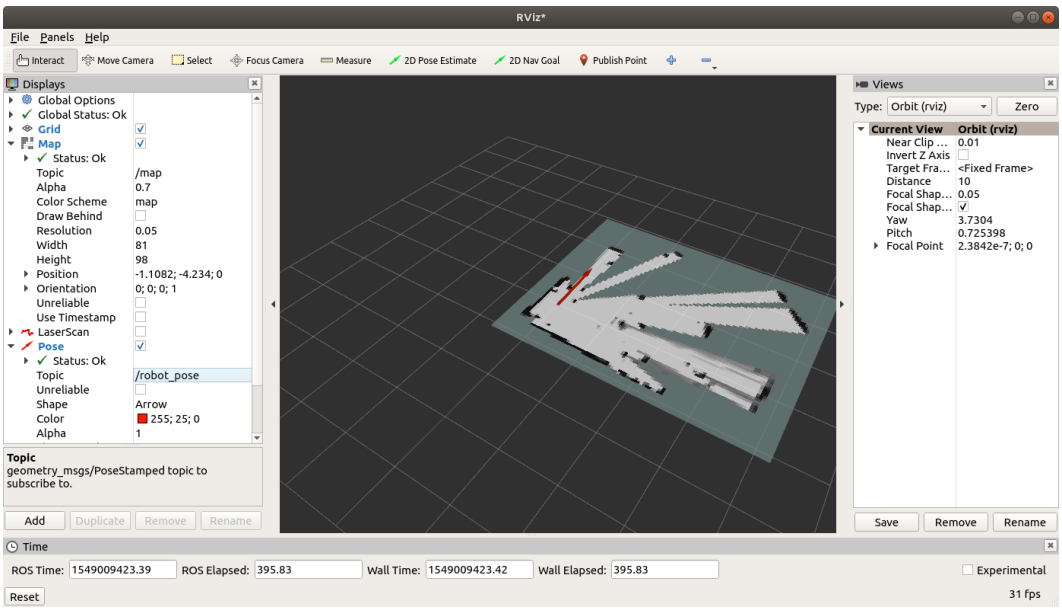

Figura 2.5: Escenario de RViz
Fuente: `https://community.husarion.com/t/rviz-gazebo-mapping-and-navigation-problem/992`

2.2. Matlab

Matlab (Matrix Laboratory) se puede definir como un software de cálculo numérico. Presenta un lenguaje de programación propio y entre las prestaciones que nos ofrece están la manipulación de matrices y vectores, la representación de datos y la implementación de algoritmos.

En este trabajo se ha utilizado la versión 9.11 de Matlab para realizar una primera implementación en la que se evaluará el método de control. En este entorno no se dispone de una visión de la marca ArUco, por lo que una vez analizado el funcionamiento se pasará a realizar el control de seguimiento, un poco más similar a la realidad, en ROS.

Para obtener unos mayores conocimientos de Matlab se recomienda visitar la página MathWorks [11]

Figura 2.6: Logo Matlab
Fuente: `https://www.unex.es/organizacion/servicios-universitarios/servicios/siue/archivos/imagenes/MATLABLogo.png/image`

Capítulo 3

Control de seguimiento

En este capítulo se explicarán las bases teóricas en las que está fundamentado el trabajo. El objetivo del mismo es obtener un algoritmo que permita que el robot siga adecuadamente una marca ArUco. Esta marca se irá desplazando con movimiento continuo y se pretende que el robot le siga manteniendo la distancia con la que se ha iniciado.

Para desarrollar nuestro algoritmo vamos a basarnos en los conocimientos que nos aporta la ingeniería de control, siendo esta una rama de la ingeniería que aplica la teoría de control para diseñar y desarrollar dispositivos con el comportamiento que se desea.

La ingeniería de control comenzó ocupándose de la automatización de dispositivos utilizados en la industria. En la actualidad, todos los conocimientos de dicha rama se han usado para progresar en el desarrollo de todo tipo de dispositivos, los cuáles requieren de un control para poder realizar una tarea de manera autónoma.

Por otro lado, un sistema de control se puede definir como un conjunto de dispositivos, los cuales se encargan de administrar, ordenar o regular cómo se comporta otro dispositivo. Por lo tanto, su objetivo principal será llevar a cabo de manera eficaz las tareas para las cuales ha sido programado.

3.1. Ley de control

En este apartado se explicará la ley de control que ha sido utilizada para que el robot móvil realice el seguimiento de la marca, la cual se desplaza siguiendo una trayectoria desconocida por el robot, de forma suave y continua. Para ello se han leído y adaptado los conocimientos expuestos en el capítulo 34 (Motion Control of Wheeled Mobile Robots), sección 4.2 (Tracking of a Reference Vehicle with the same kinematics) del libro Springer Handbook of Robotics [12].

La ley de control que se explica en el mencionado capítulo hace que un robot tipo monociclo pueda seguir tanto en posición como en orientación a otro robot del mismo tipo, a este segundo lo llamaremos robot de referencia. Para que se pueda realizar dicho seguimiento se necesita una actualización constante tanto de la posición como de la orientación de ambos robots. Con estos valores se calculan los errores de posición en el eje x y en el eje y, además del error en el ángulo θ. A continuación, se pasará a explicar el desarrollo completo de dicha ley de control, aunque en este caso ha sido adaptada a nuestro problema.

Como ya se ha comentado anteriormente, nuestro principal objetivo es obtener un control tanto en posición como en orientación de una marca ArUco. Donde esta marca se moverá con un movimiento arbitrario y el robot deberá seguirla manteniendo la distancia con la que comenzó la ejecución del algoritmo. Además, el movimiento de la marca presentará unas limitaciones en sus desplazamientos y giros porque el robot presenta unas limitaciones de velocidad de 2 m/s.

En esta situación la elección de un punto de referencia, al cual llamaremos P_r, tiene menor importancia que cuando se realiza el seguimiento de control solo de la posición. En este caso P_r se define como el origen de la estructura del chasis del robot.

A continuación se muestra una representación gráfica donde se pueden ver los parámetros anteriormente mencionados:

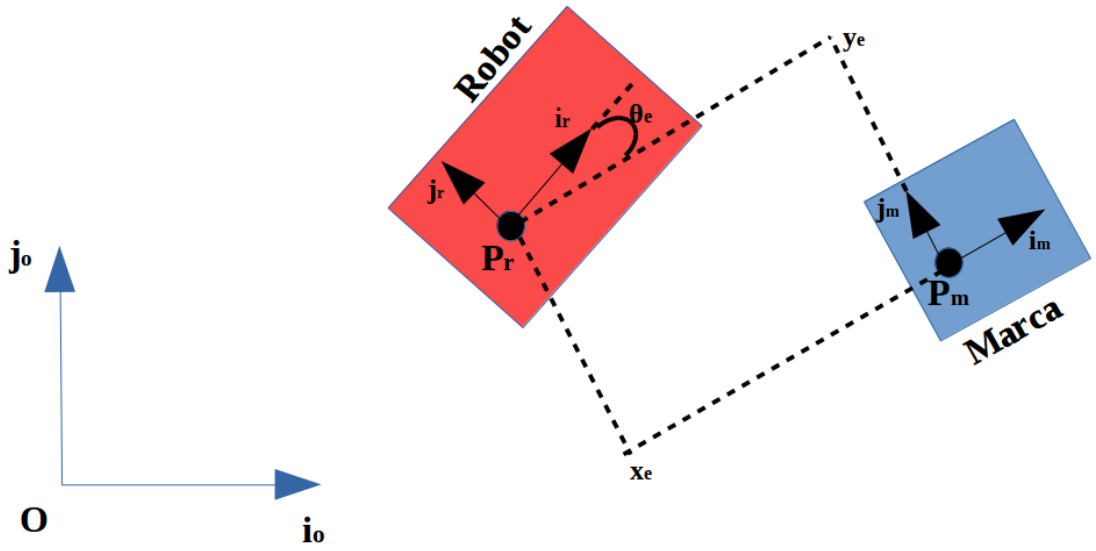

Figura 3.1: Diagrama de las referencias

En esta figura (Figura 3.1) se puede observar a la izquierda del todo el sistema de referencia $F_o = \{O, i_o, j_o\}$, en el cual nos vamos a basar para realizar los cálculos que se explicarán a continuación. Además podemos ver un rectángulo rojo que representa

el robot, en él se visualizan sus propios ejes, a esto le llamaremos sistema del robot $F_r = \{P_r, i_r, j_r\}$. Por otro lado, el rectángulo azul representa la marca ArUco, la cual va a ser seguida por el robot y por lo tanto podríamos decir que es nuestro objeto de referencia, en ella aparecen también representados sus propios ejes, y a todo esto se le denomina sistema de la marca $F_m = \{P_m, i_m, j_m\}$.

El problema de seguimiento suele estar asociado en control con el problema de estabilizar asintóticamente [1] la trayectoria de referencia. En este caso, una condición necesaria para la existencia de una solución de control es que la referencia sea factible.

Las trayectorias factibles t→(x_m(t),y_m(t),θ_m(t)) son funciones de tiempo suaves. Además, son una solución al modelo cinemático del robot para alguna entrada de control específica t→(u_m(t))=($u_{1,m}$(t),$u_{2,m}$(t)), llamado control de referencia.

Para un robot tipo monociclo anteriormente comentado se expresa de la siguiente manera:

$$\dot{x}_m = u_{1,m} \cos\theta_m \tag{3.1}$$

$$\dot{y}_m = u_{1,m} \sin\theta_m \tag{3.2}$$

$$\dot{\theta}_m = u_{2,m} \tag{3.3}$$

Siendo:

- x_m: La coordenada de i_m en F_o
- y_m: La coordenada de j_m en F_o
- θ_m: El ángulo formado entre i e i_m

Es decir, las trayectorias de referencia factibles corresponden al movimiento de un sistema de referencia Fm unido rígidamente a un robot de referencia de tipo monociclo Pm. En este caso suponemos que, en ese robot virtual a seguir va nuestra marca ubicada a una distancia media de las ruedas accionadas.

Una vez conocido esto, el problema será obtener un control en bucle cerrado que permita estabilizar asintóticamente el error de seguimiento en cero. Este error de seguimiento se expresa de la siguiente manera:

$$error = (x - x_m, y - y_m, \theta - \theta_m) \tag{3.4}$$

[1] La estabilidad asintótica significa que soluciones que empiezan suficientemente cerca, no sólo permanecen cercanas sino que eventualmente acaban convergiendo al mismo equilibrio [13].

Los pasos que se van a dar para determinar dicho control son los siguientes:

1. Establecer las ecuaciones de error con respecto al marco F_m
2. Transformar dichas ecuaciones en la forma de cadena mediante un cambio de variable
3. Diseñar las leyes de control estabilizadoras para el sistema transformado.

Paso 1: Obtener el error de seguimiento en la posición x (x_e) e y (y_e) con respecto al marco F_m:

$$\begin{pmatrix} x_e \\ y_e \end{pmatrix} = \begin{pmatrix} \cos\theta_m & \sin\theta_m \\ -\sin\theta_m & \cos\theta_m \end{pmatrix} \begin{pmatrix} x - x_m \\ y - y_m \end{pmatrix} \tag{3.5}$$

A continuación, se calculará la derivada temporal:

$$\begin{pmatrix} \dot{x_e} \\ \dot{y_e} \end{pmatrix} = \dot{\theta_m} \begin{pmatrix} -\sin\theta_m & \cos\theta_m \\ -\cos\theta_m & -\sin\theta_m \end{pmatrix} \begin{pmatrix} x - x_m \\ y - y_m \end{pmatrix} + \begin{pmatrix} \cos\theta_m & \sin\theta_m \\ -\sin\theta_m & \cos\theta_m \end{pmatrix} \begin{pmatrix} \dot{x} - \dot{x_m} \\ \dot{y} - \dot{y_m} \end{pmatrix} \tag{3.6}$$

$$\begin{pmatrix} \dot{x_e} \\ \dot{y_e} \end{pmatrix} = \begin{pmatrix} u_{2,m} y_e + u_1 \cos(\theta - \theta_m) - u_{1,m} \\ -u_{2,m} x_e + u_1 \sin(\theta - \theta_m) \end{pmatrix} \tag{3.7}$$

Si consideramos que

$$\theta_e = \theta - \theta_m \tag{3.8}$$

Entonces el sistema anterior queda de la siguiente manera:

$$\dot{x_e} = u_{2,m} y_e + u_1 \cos\theta_e - u_{1,m} \tag{3.9}$$

$$\dot{y_e} = -u_{2,m} x_e + u_1 \sin\theta_e \tag{3.10}$$

$$\dot{\theta_e} = u_2 - u_{2,m} \tag{3.11}$$

Paso 2: Mediante un cambio de variable se transformarán las ecuaciones anteriores en la forma de cadena.

$$Z_1 = x_e \tag{3.12}$$

$$Z_2 = y_e \tag{3.13}$$

$$Z_3 = \tan\theta_e \tag{3.14}$$

$$w_1 = u_1 \cos\theta_e - u_{1,m} \tag{3.15}$$

$$w_2 = \frac{u_2 - u_{2,m}}{cos^2(\theta_e)} \tag{3.16}$$

Hay que tener en cuenta que el error de orientación entre el robot y la marca ArUco tiene que ser menor que $\pi/2$. Es decir, este controlador solo estará definido cuando el

error de orientación se encuentra entre $-\pi/2$ y $\pi/2$ porque si no el sistema se vuelve inestable.

Por lo tanto, con el cambio de variables el sistema definido anteriormente ahora presenta la siguiente forma:

$$\dot{Z}_1 = u_{2,m} Z_2 + w_1 \tag{3.17}$$

$$\dot{Z}_2 = -u_{2,m} Z_1 + u_{1,m} Z_3 + w_1 Z_3 \tag{3.18}$$

$$\dot{Z}_3 = w_2 \tag{3.19}$$

Paso 3: Si nos fijamos en las anteriores ecuaciones podemos observar que se trata de un sistema encadenado y reordenando obtenemos la ley de control, como se muestra a continuación:

$$w_1 = -k_1 |u_{1,m}| (Z_1 + Z_2 Z_3) \tag{3.20}$$

$$w_2 = -k_2 u_{1,m} Z_2 - k_3 |u_{1,m}| Z_3 \tag{3.21}$$

La ley de control hace que el origen del sistema sea globalmente asintóticamente estable si $u_{1,r}$ es una función diferenciable acotada cuya derivada esta acotada y no tiende a cero cuando t tiende a infinito.

Finalmente, para obtener un control mucho más sencillo, lo que se hace es linealizar la ley de control quedando de la siguiente forma:

$$w_1 = -k_1 |u_{1,m}| Z_1 \tag{3.22}$$

$$w_2 = -k_2 u_{1,m} Z_2 - k_3 |u_{1,m}| Z_3 \tag{3.23}$$

Si consideramos que:

$$w_1 = u_1 - u_{1,m} \tag{3.24}$$

$$w_2 = u_2 - u_{2,m} \tag{3.25}$$

Finalmente, la ley de control simplificada queda de la siguiente forma:

$$u_1 = u_{1,m} - k_1 |u_{1,m}| Z_1 \tag{3.26}$$

$$u_2 = u_{2,m} - k_2 u_{1,m} Z_2 - k_3 |u_{1,m}| Z_3 \tag{3.27}$$

Siendo:

- u_1 la velocidad lineal del robot
- u_2 la velocidad angular del robot

Esta última forma en la que se ha expresado la ley de control es la que ha sido utilizada para llevar a cabo nuestro trabajo.

Capítulo 4

Desarrollo del algoritmo de control

En este capítulo se explicarán los algoritmos utilizados para obtener el objetivo final. La primera parte estará dedicada al algoritmo que se ha llevado a cabo en el entorno de Matlab, mientras que en la segunda parte se expondrá el algoritmo realizado en el entorno de ROS.

4.1. Implementación en Matlab

Lo primero que se realizará después de comprender los fundamentos teóricos necesarios para realizar el control de seguimiento, es el desarrollo del algoritmo que permitirá cumplir el principal objetivo. Para ello se usará el entorno de Matlab, el cual nos va a permitir realizar una simulación para poder ajustar nuestro controlador y comprobar que este funciona como deseamos.

A continuación, se explica el algoritmo de una forma detallada. Para obtener el código completo ir al Anexo B.

Primer paso: Lo primero que se hace es crear un vector de tiempos, esto es necesario para poder llevar a cabo la simulación en un tiempo determinado. A continuación, se muestra el código de Matlab que recoge esto anteriormente contado.

```
% Vector de tiempos
Tmax= 2000
delta = 0.1
tiempo=(0:delta:Tmax)';
t=length(tiempo);
```

Para escoger el tiempo máximo (Tmax) se ha considerado que debía ser un valor elevado para que así el regulador pudiese alcanzar su régimen permanente sin ningún problema. Se comenzó probando con 500, pero se vio que se necesitaba más tiempo y por ello se aumentó hasta los 2000 segundos.

A continuación, para elegir delta se han valorado números bastante bajos. Esta decisión ha sido tomada porque delta representa el valor del paso y por lo tanto cuanto

menor valor tenga, mayor número de valores de simulación tendremos, obteniendo como consecuencia una representación mucho más suave. Si se elige un número elevado como puede ser 100, podremos ver que la representación es mucho más abrupta y por lo tanto mucho menos precisa. Esto que se describe se puede ver en la figura 4.1

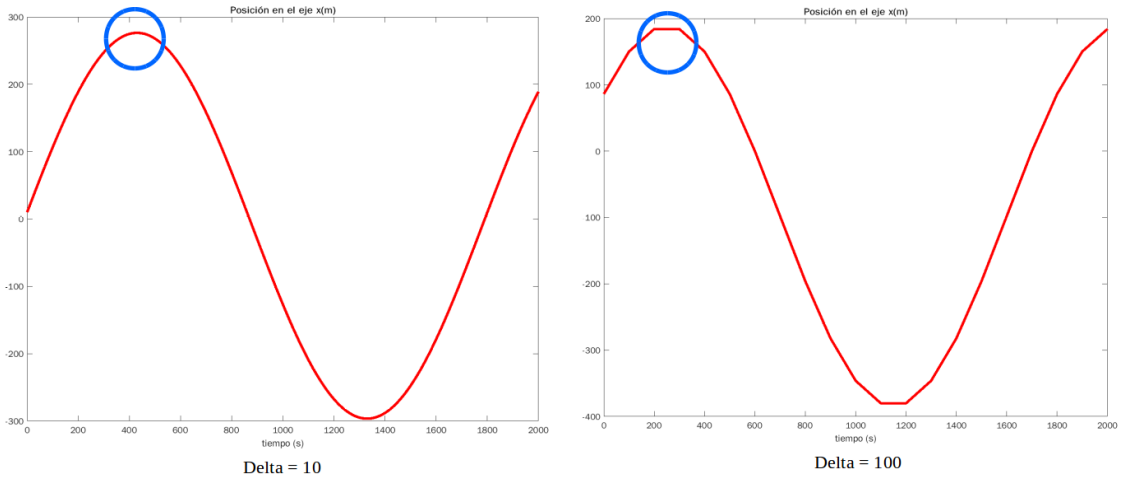

Figura 4.1: Comparación de deltas. En la gráfica de la izquierda el valor de delta es 10, mientras que para la gráfica de la derecha el valor de delta es 100

Finalmente, en la línea 4 lo que se hace es formar el vector de tiempo y en la línea 5 se obtiene la longitud que tendrá dicho vector, que es el valor que necesitamos para llevar a cabo el bucle de control que se explicará un poco más adelante.

Segundo paso: Inicialización tanto de la posición de la marca como la posición del robot. Esto se hace porque para obtener el valor de las variables del controlador tenemos que simular la trayectoria tanto de la marca ArUco como la del robot, y así poder comprobar si realiza el robot perfectamente el seguimiento. Se considera que inicialmente la marca parte del origen (0,0) en dos dimensiones y que el ángulo θ es también 0. Por otro lado, el robot se inicializa en la posición (-1,-0.5) y el ángulo θ es 0. A continuación se muestra la parte de código que define lo anteriormente descrito.

```
%Inicializacion de la posicion de la marca
xMarca = 0;
yMarca = 0;
titaMarca = 0;

%Inicializamos la posicion del robot
xRobot=-1;
yRobot=-0.5;
titaRobot=0;
```

Tercer paso: Creación de los vectores que se van a utilizar con la finalidad de poder almacenar los datos que se obtengan en el bucle de control, y así poder utilizarlos posteriormente para representar gráficamente los resultados.

```
%Vectores para guardar datos
xMarcaIni=zeros(t,1);
yMarcaIni=zeros(t,1);
titaMarcaIni=zeros(t,1);

xRobotIni=zeros(t,1);
yRobotIni=zeros(t,1);
titaRobotIni=zeros(t,1);

v_xe = zeros(t,1);
v_ye =zeros (t,1);
v_titae = zeros (t,1);

xR=zeros(t,1);
yR=zeros(t,1);
titaR=zeros(t,1);

xM=zeros(t,1);
yM=zeros(t,1);
titaM=zeros(t,1);

v_v=zeros(t,1);
v_w=zeros(t,1);

v_vMarca=zeros(t,1);
v_wMarca=zeros(t,1);
```

Cuarto paso: Este paso se puede considerar como el paso principal ya que en él se desarrolla el bucle de control. A continuación, se pasa a explicar detalladamente cada una de las partes que componen dicho bucle de control y el porqué de los valores elegidos en las constantes que se requieren.

Lo primero de todo será definir un bucle *for*, el cual servirá para recorrer todo el vector de tiempos. Dentro de este bucle se realizan distintas tareas, las cuales van a ser desglosadas a continuación:

- **Tarea 1**: Se define la trayectoria de la marca, para ello se necesita una velocidad lineal y una velocidad angular, estos valores se eligen según queramos que se mueva nuestra marca ArUco. En este caso se elige una velocidad lineal de 0.2 metros por segundo y una velocidad angular de 0 grados por segundo.

```
%TRAYECTORIA MARCA
xMarcaIni(it,:)=xMarca;
yMarcaIni(it,:)=yMarca;
titaMarcaIni(it,:)=titaMarca;

vMarca=0.2;
wMarca=0.0;
```

– **Tarea 2**: Se obtiene y guarda la localización actual del robot.

```
%Localizacion Actual del robot
xRobotIni(it,:)=xRobot;
yRobotIni(it,:)=yRobot;
titaRobotIni(it,:)=titaRobot;
```

– **Tarea 3**: Se calculan los errores de posición y orientación. Este cálculo se va a hacer en coordenadas cartesianas, y para ello se necesita la posición en x e y tanto del robot como de la marca y la orientación de la marca.

```
%Error en coordenadas cartesianas
xe = (xRobot-xMarca)*cosd(titaMarca)+(yRobot-yMarca)*sind(titaMarca);
ye =-(xRobot-xMarca)*sind(titaMarca)+(yRobot-yMarca)*cosd(titaMarca);
titae = titaRobot-titaMarca;
```

– **Tarea 4**: Se calcula la posición y orientación de la marca.

```
% Marca
titaMarca=titaMarca + wMarca*delta;
DyMarca= zeros(1,1);
DxMarca= vMarca*delta;

if (wMarca ~= 0)
    L_Marca= vMarca/(wMarca*pi/180);
    DyMarca= L_Marca -L_Marca.*cosd(wMarca*delta);
    DxMarca= L_Marca.*sind(wMarca*delta);
end

xMarca= xMarca +DxMarca .*cosd(titaMarca) -DyMarca .*sind(titaMarca);
yMarca= yMarca +DxMarca .*sind(titaMarca) +DyMarca .*cosd(titaMarca);
```

Como se ve en el código dependiendo de si la marca presenta velocidad angular o no, se calcula de distinta manera. En el primer caso, para calcular la posición de la marca se necesita obtener el desplazamiento en x (Dx) y el desplazamiento en y (Dy). Estos valores se obtienen por trigonometría.

– **Tarea 5**: En esta parte del código se define la ley de control, la cual ha sido explicada en el capítulo 3. Para ello se declaran las tres constantes k_1, k_2 y k_3. Los valores que se han elegido para estas variables han sido obtenidos mediante pruebas en simulación, es decir, se ha partido de un valor bajo y se han ido tanteando valores hasta obtener el resultado deseado.

Como se puede ver en la siguiente parte de código, el parámetro k_1 es el que modifica la velocidad lineal, mientras que los parámetros k_2 y k_3 modifican la velocidad angular. El parámetro k_2 es el que corrige el error en el eje y, siendo este eje en coordenadas cartesianas la suma de dos componentes. Por un lado, el primer componente es la diferencia entre el robot y la marca en el eje x multiplicada por

el seno del ángulo θ de la marca. Por otro lado, el segundo componente es la diferencia entre el robot y la marca en el eje y multiplicada por el coseno del ángulo θ de la marca. El parámetro k_3 corrige el error de la orientación.

```
%Ley de control
k1=0.1;
k2=0.1;
k3=1.0;

Z1=xe;
Z2=ye;
Z3=tand(titae);

v=vMarca-(k1*abs(vMarca)*Z1);
w=wMarca-(k2*vMarca*Z2)-(k3*abs(vMarca)*Z3);
```

- **Tarea 6**: Una vez obtenida la velocidad lineal y angular que debe tener el robot, se calcula la posición y orientación del mismo. Se realiza de la misma forma que hemos obtenido la posición y orientación de la marca.

```
%Calculo de la posicion del robot
titaRobot=titaRobot + w*delta;
Dy= zeros(1,1);
Dx= v*delta;

if (w ~= 0)
    L= v./(w *pi/180);
    Dy= L -L.*cosd(w*delta);
    Dx= L.*sind(w*delta);
end

xRobot= xRobot +Dx .*cosd(titaRobot) -Dy .*sind(titaRobot);
yRobot= yRobot +Dx .*sind(titaRobot) +Dy .*cosd(titaRobot);
```

- **Tarea 7**: Por último, se almacenan todos los datos recogidos para posteriormente poder representarlos.

```
%Vector que almacena los datos del bucle
xR(it,:)=xRobot;
yR(it,:)=yRobot;
titaR(it,:)=titaRobot;

xM(it,:)=xMarca;
yM(it,:)=yMarca;
titaM(it,:)=titaMarca;
v_v(it,:)=v;
v_w(it,:)=w;
v_vMarca(it,:)=vMarca;
v_wMarca(it,:)=wMarca;
v_xe(it,:)=xe;
v_ye(it,:)=ye;
v_titae(it,:)=titae;
```

Quinto paso: El último de los pasos que se ha llevado a cabo es la representación gráfica. En esta parte del código se dibujan nueve gráficas, que pasaré a explicar a continuación.

```
%Posicion tanto de la marca como del robot en el eje x
subplot(3,3,1)
plot(tiempo,xM,'r','LineWidth',3)
hold on
plot(tiempo,xR,'b','LineWidth',3)
xlabel ('tiempo (s)')
title ('Posicion en el eje x(m)')
legend ('Marca','Robot')

%Posicion tanto de la marca como del robot en el eje y
subplot(3,3,2)
plot(tiempo,yM,'r','LineWidth',3)
hold on
plot(tiempo,yR, 'b','LineWidth',3)
xlabel ('tiempo (s)')
title ('Posicion en el eje y')
legend ('Marca','Robot')

%Posicion tanto de la marca como del robot del angulo theta
subplot(3,3,3)
plot(tiempo,titaM,'r','LineWidth',3)
hold on
plot(tiempo,titaR, 'b','LineWidth',3)
xlabel ('tiempo (s)')
title ('Posicion de angulo theta')
legend ('Marca','Robot')

%Velocidad Lineal de la marca y del robot
subplot(3,3,4)
plot (tiempo,v_vMarca,'r','LineWidth',3)
hold on
plot(tiempo,v_v,'b','LineWidth',3)
xlabel ('tiempo (s)')
title ('Velocidad Lineal')
legend ('marca','robot')

%Velocidad Angular de la marca y del robot
subplot(3,3,5)
plot(tiempo,v_wMarca,'r','LineWidth',3)
hold on
plot (tiempo,v_w,'b','LineWidth',3)
xlabel ('tiempo (s)')
title ('velocidad Angular')
legend ('Marca','Robot')

%Error en el eje x
subplot(3,3,6)
plot(xM,yM,'r','LineWidth',3)
hold on
plot(xR,yR,'b','LineWidth',3)
xlabel ('x (m)')
title ('x, y(m)')
legend ('Marca','Robot')
```

```
%Error en el eje y
subplot(3,3,7)
plot(tiempo,v_ye,'g','LineWidth',3)
xlabel('tiempo (s)')
title('Error en el eje y')

%Error en el angulo tita
subplot(3,3,8)
plot(tiempo,v_titae,'g','LineWidth',3)
xlabel('tiempo (s)')
title('Error en el angulo tita')

%Eje x y eje y
subplot(3,3,9)
plot (tiempo,v_xe,'g','LineWidth',3)
xlabel('tiempo (s)')
title ('Error en el eje x')
```

En las tres primeras gráficas se representa la posición en el eje x, en el eje y, y el ángulo theta tanto del robot como de la marca a lo largo del tiempo, el cual ha sido definido anteriormente.

En las tres siguientes se muestra tanto la velocidad lineal como la velocidad angular y la posición en x e y, tanto del robot como de la marca.

En las tres últimas gráficas se representan los errores que se obtienen en el eje x, en el eje y, y en la orientación. Para tener un control adecuado, hay que fijarse que estos errores se estabilicen.

El análisis del comportamiento del control mediante esta implementación se describe en el capítulo 5.

4.2. Implementación en ROS

Una vez que se ha probado el controlador en Matlab, lo siguiente que se va a hacer es probar ese algoritmo en ROS, para ello escribiremos el código en Python. A continuación, se explica de una manera más detallada las partes de código más importantes. El código completo que se ha utilizado en ROS está en el Anexo C.

Para llevar a cabo la simulación en ROS solo se ha generado un fichero .py. Los ficheros de detección de la marca y los de envíos de comandos al robot que también se utilizan son del trabajo de fin de grado del que partimos [2] y por lo tanto no se comentarán en esta sección.

El fichero .py que se ha implementado se llama ArucoTracking.py, y se compone de varias funciones y un *main*. Se comenzará mencionando las funciones de las que se compone y después se comentará el *main*.

Este fichero se compone de 7 funciones y estás son:

- **Conversión de unidades**: se definen tres funciones para poder usar el seno, el coseno y la tangente cuando introducidos los datos en grados.

```
def sen(grados):
    return math.sin(math.radians(grados))

def cos(grados):
    return math.cos(math.radians(grados))

def tan(grados):
    return math.tan(math.radians(grados))
```

- **Detección de marca**: Esta función se ha creado porque nuestro controlador solo tiene sentido que funcione cuando detecta marca, y por lo tanto inicialmente tendremos que saber si detecta marca o no.

```
def deteccionMarcaInicio():

if arucoNav.mark_x==0.0 and arucoNav.mark_y==0.0 and
arucoNav.mark_theta==0.0:
        arucoNav.marca_detectada_ini=0
        print('No detecto inicialmente a la marca')
   else:
        arucoNav.marca_detectada_ini=1
        print('Marca inicialmente detectada')
```

- **Giro completo**: Esta función se ha creado para que el robot, en caso de que no encuentre la marca, comience a girar para intentar buscarla en otra parte del escenario.

```
def girar():
   print ("Comienzo a girar")
   rospy.Timer(rospy.Duration(10),my_callback,oneshot=True)

   while (not arucoNav.girocompleto==1) and not rospy.is_shutdown():
           arucoNav.vel.angular.z=-1.256637061
           pub.publish(arucoNav.vel)
           #print("Estoy girando")

   print("Giro terminado")
   arucoNav.girocompleto=0
   arucoNav.vel.angular.z=0.0

   pub.publish(arucoNav.vel)

def my_callback(event):
    print("Han pasado 5 segundos")
    arucoNav.girocompleto=1
```

– **Avanzar**: Esta función al igual que la anterior se ha creado para que una vez que ha terminado de girar el robot completamente y no ha encontrado marca, avance un metro y si no ve marca vuelva a girar.

```
def avanzar():
  print ("Comienzo a avanzar")
  rospy.Timer(rospy.Duration(1.0),my_callback_1,oneshot=True)
  while (not arucoNav.avancecompleto==1) and not rospy.is_shutdown():
          arucoNav.vel.linear.x=1.0
          pub.publish(arucoNav.vel)
          print ("Estoy avanzando")
  print ("Avance terminado")
  arucoNav.avancecompleto=0
  arucoNav.vel.linear.x=0.0
  pub.publish(arucoNav.vel)

def my_callback_1(event):
    print ("Ha pasado 1 segundo")
        arucoNav.avancecompleto=1
```

– **Posición de la marca**: Con esta función lo que se hace es obtener los valores tanto de posición como de orientación de la marca, y con ellos posteriormente se pueden calcular los errores que se necesitan para el controlador.

```
def arucoCallback(aruco_pose_message):
    arucoNav.mark_x=aruco_pose_message.pose.position.x
    arucoNav.mark_y=aruco_pose_message.pose.position.y
    arucoNav.mark_z=aruco_pose_message.pose.position.z
    rotation=aruco_pose_message.pose.orientation
    angles=tf.transformations.euler_from_quaternion(quaternion=
    (rotation.x, rotation.y, rotation.z, rotation.w))
    arucoNav.mark_alpha=numpy.rad2deg(angles[0])
    arucoNav.mark_beta=numpy.rad2deg(angles[1])
    arucoNav.mark_theta=numpy.rad2deg(angles[2])
```

– **Posición del robot**: Al igual que hemos obtenido la posición de la marca también se necesita conocer la posición y orientación que presenta el robot en cada instante.

```
def robotCallback(robot_pose_message):
    arucoNav.robot_x=robot_pose_message.pose.pose.position.x
    arucoNav.robot_y=robot_pose_message.pose.pose.position.y
    arucoNav.robot_z=robot_pose_message.pose.pose.position.z

    rotation_robot=robot_pose_message.pose.pose.orientation
    angles_robot=tf.transformations.euler_from_quaternion(quaternion=
    (rotation_robot.x, rotation_robot.y, rotation_robot.z,
    rotation_robot.w))

    arucoNav.robot_alpha=numpy.rad2deg(angles_robot[0])
    arucoNav.robot_beta=numpy.rad2deg(angles_robot[1])
    arucoNav.robot_theta=numpy.rad2deg(angles_robot[2])
```

– **Subscripción de nodos**: Esta función se ha creado para subscribir los nodos, los cuáles nos dan el resultado de la posición y orientación de la marca y del robot.

```
def listener():
    rospy.Subscriber('/campero/robotnik_base_control/odom',Odometry,
    robotCallback)
    rospy.Subscriber('aruco_pose', PoseStamped, arucoCallback)
```

– **Representación**: Con esta función se pretende que cuando finalice el programa principal nos devuelva unas gráficas para comprobar que el seguimiento ha sido correcto.

```
def representacion ():
    plt.figure()
    plt.plot(arucoNav.t, arucoNav.xM, 'b')
    plt.plot(arucoNav.t, arucoNav.xR, 'r')
    plt.xlabel('Tiempo')
    plt.title('Posicion en el eje x(m)')
    plt.show()
```

– **Tracking**: es la función principal en la cual se va a calcular el controlador. Para explicar lo que hace esta función se presenta el siguiente diagrama de flujo, figura 4.2.

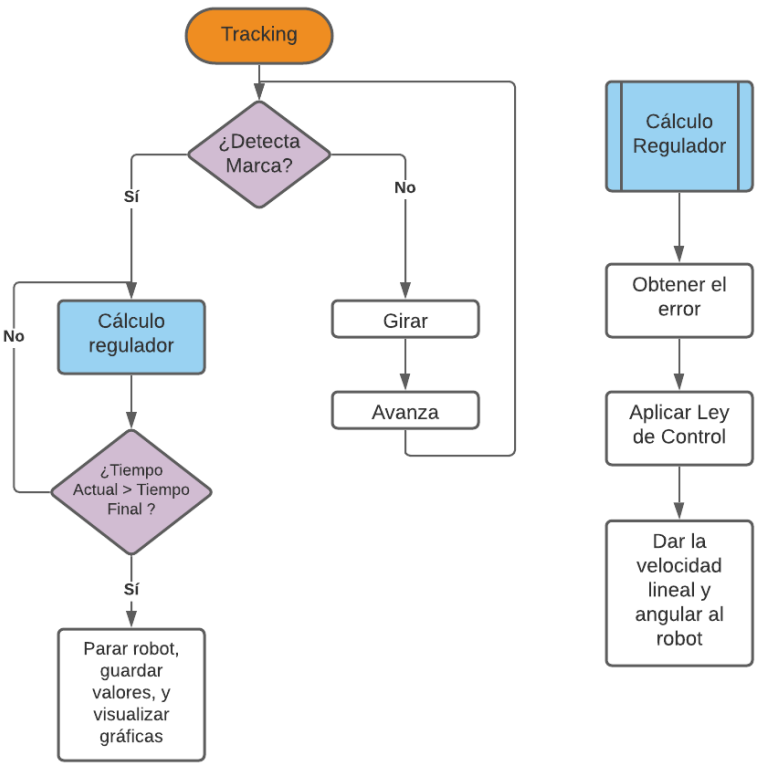

Figura 4.2: Diagrama de flujo de la función Tracking

A continuación, se muestra el código que describe el anterior diagrama de flujo:

```
def tracking():
 deteccionMarcaInicio()
 #print ('Marca_Inicio',arucoNav.marca_detectada_ini)

 if (arucoNav.marca_detectada_ini==1):
   #print ("Estoy dentro de marca detectada y avanzo")
   if (arucoNav.fin==0):
         tiempo = rospy.get_time()
         rate = rospy.Rate(10000)
         arucoNav.mark_x_list.append(arucoNav.mark_x)
         arucoNav.robot_x_list.append(arucoNav.robot_x)
         arucoNav.mark_y_list.append(arucoNav.mark_y)
         arucoNav.robot_y_list.append(arucoNav.robot_y)
         arucoNav.mark_theta_list.append(arucoNav.mark_theta)
         arucoNav.robot_theta_list.append(arucoNav.robot_theta)
         arucoNav.tiempo_list.append(tiempo)
         rate.sleep()

         tiempoActualizado = arucoNav.tiempo_list[0] + 130.0
         tiempoActualizado1 = arucoNav.tiempo_list[0] + 1.0

         # Trayectoria de la marca
         vMarca=0.2
         wMarca=1.8

         # Error relativo
         xe = (arucoNav.robot_x-arucoNav.mark_x)*cos(arucoNav.mark_theta)
         +(arucoNav.robot_y-arucoNav.mark_y)*sen(arucoNav.mark_theta)
         #print ("error en x", xe)
         ye = -(arucoNav.robot_x-arucoNav.mark_x)*sen(arucoNav.mark_theta)
         +(arucoNav.robot_y-arucoNav.mark_y)*cos(arucoNav.mark_theta)
         #print ("error en y",ye)
         titae = arucoNav.robot_theta-arucoNav.mark_theta
         #print ("error en el angulo tita",titae)

         #Almacenar los valores de los errores
         rate = rospy.Rate(10000)
         arucoNav.error_x_list.append(xe)
         #print ("error en x",arucoNav.error_x_list)
         arucoNav.error_y_list.append(ye)
         #print ("error en y",arucoNav.error_y_list)
         arucoNav.error_theta_list.append(titae)
         #print ("error en tita",arucoNav.error_theta_list)
         rate.sleep()

         # Ley de control
         k1 = 0.2
         k2 = 60.0
         k3 = 80.0
         Z1 = xe
         Z2 = ye
         Z3 = tan(titae)

         #print (Z1)
         #print (Z2)
         #print (Z3)
```

```
        arucoNav.velocidad_linear=(vMarca-(k1*abs(vMarca)*Z1))
        #print ("velocidad lineal",arucoNav.velocidad_linear)
        arucoNav.velocidad_angular=wMarca-(k2*vMarca*Z2)-
        (k3*abs(vMarca)*Z3)
        #print ('velocidad_angular',arucoNav.velocidad_angular)

        arucoNav.vel.linear.x= (arucoNav.velocidad_linear)
        arucoNav.vel.angular.z= arucoNav.velocidad_angular

        #Almacenar los valores de las velocidades
        rate = rospy.Rate(10000)
        arucoNav.velocidad_linear_list.append(arucoNav.vel.linear.x)
        arucoNav.velocidad_angular_list.append(arucoNav.vel.angular.z)
        rate.sleep()

        #deteccionMarca()

    if (tiempo > tiempoActualizado):
       #print ('dentro_del_if')
        arucoNav.vel.angular.z = 0.0
        arucoNav.vel.linear.x = 0.0
        arucoNav.vel.linear.y = 0.0
        a = len(arucoNav.tiempo_list)
        arucoNav.longTiempo.append(a)

        arucoNav.t = arucoNav.tiempo_list[0:arucoNav.longTiempo[0]]
        a = len(arucoNav.t)

        arucoNav.xM = arucoNav.mark_x_list[0:arucoNav.longTiempo[0]]
        arucoNav.xR = arucoNav.robot_x_list[0:arucoNav.longTiempo[0]]
        arucoNav.yM = arucoNav.mark_y_list[0:arucoNav.longTiempo[0]]
        arucoNav.yR = arucoNav.robot_y_list[0:arucoNav.longTiempo[0]]
        arucoNav.titaM = arucoNav.mark_theta_list
        [0:arucoNav.longTiempo[0]]
        arucoNav.titaR = arucoNav.robot_theta_list
        [0:arucoNav.longTiempo[0]]

        arucoNav.errorX=arucoNav.error_x_list[0:arucoNav.longTiempo[0]]
        arucoNav.errorY=arucoNav.error_y_list[0:arucoNav.longTiempo[0]]
        arucoNav.errorTita=arucoNav.error_theta_list
        [0:arucoNav.longTiempo[0]]

        arucoNav.velocidad_linear=arucoNav.velocidad_linear_list
        [0:arucoNav.longTiempo[0]]
        arucoNav.velocidad_angular=arucoNav.velocidad_angular_list
        [0:arucoNav.longTiempo[0]]

        arucoNav.diferencia_eje_x= arucoNav.dif_x_list
        [0:arucoNav.longTiempo[0]]
        arucoNav.diferencia_eje_y= arucoNav.dif_y_list
        [0:arucoNav.longTiempo[0]]
        arucoNav.coseno_tita = arucoNav.cos_tita_list
        [0:arucoNav.longTiempo[0]]
        arucoNav.seno_tita = arucoNav.sen_tita_list
        [0:arucoNav.longTiempo[0]]

        representacion()
```

```
        pub.publish(arucoNav.vel)
  else:
    print ("Estoy dentro de marca no detectada y giro")
    girar()
    avanzar()
```

Con respecto al *main* decir que es principalmente donde se publican los nodos que se necesitan y donde se manda la función principal para que se ejecute mientras el robot este activo. El código empleado es el siguiente:

```
if __name__=='__main__':
  try:
        rospy.init_node('arucoTracking')
        pub = rospy.Publisher('/campero/cmd_vel',Twist,queue_size=10)
        arucoNav=arucoTracking()
        listener()
        rospy.sleep(1)
        print ("Inicio del seguimiento de la marca")
        #while not arucoNav.fin and not rospy.is_shutdown():
        while not rospy.is_shutdown():
                tracking()
        if rospy.is_shutdown():
                rospy.loginfo("Programa cancelado")
        else:
                rospy.loginfo("El robot ha llegado")
  except rospy.ROSInterruptException:
        rospy.loginfo("Test de navegacion finalizado")
```

Capítulo 5

Resultados y Análisis

En este capítulo se analizarán los resultados obtenidos en las simulaciones. Primero se comentarán aquellos que han sido realizados en Matlab y posteriormente los obtenidos en ROS.

5.1. Resultados en Matlab

Con el código explicado en el capítulo 4 se llevan a cabo una serie de experimentos para poner a prueba el algoritmo y ver que el controlador funciona adecuadamente.

A continuación, se van a exponer las diferentes simulaciones. Entre ellas tenemos un movimiento lineal, un movimiento circular, un movimiento sinusoidal, un movimiento en forma de espiral y una trayectoria en forma de ocho.

5.1.1. Movimiento lineal

Se ha comenzado por la simulación de un movimiento lineal, ya que se trata de una trayectoria bastante sencilla que solo presenta un desplazamiento en el eje x y por lo tanto la marca solo tiene velocidad lineal.

Para esta simulación los parámetros elegidos para k_1, k_2 y k_3 son 0.1, 30.0 y 35.0 respectivamente. Además se simulará para un tiempo máximo de 2000 segundos, con un valor de delta de 0.1.

A continuación en la figura 5.1, se van a mostrar las distintas gráficas obtenidas de las simulaciones y seguidamente se analizarán y explicarán detalladamente cada una de las gráficas.

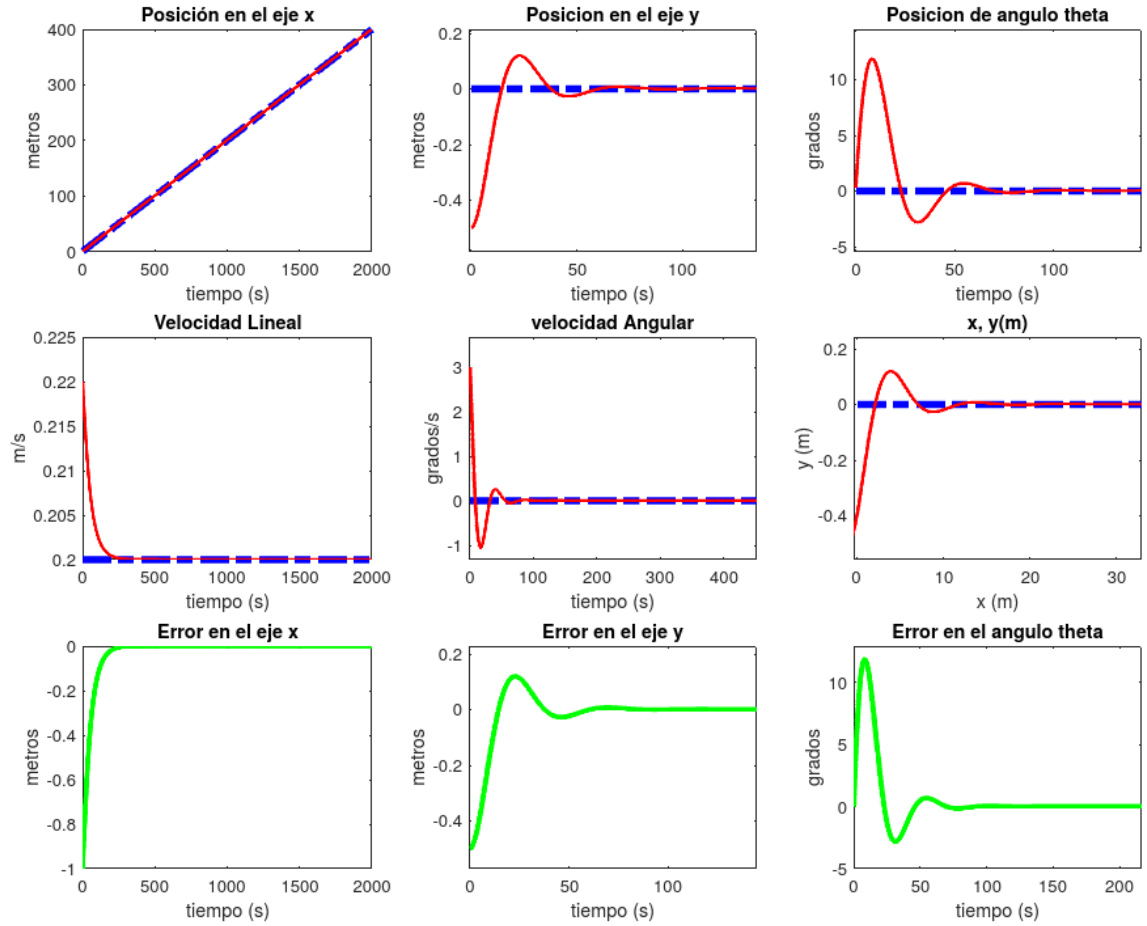

Figura 5.1: Ejemplo de simulación de la trayectoria lineal, donde la línea azul corresponde a la marca y la línea roja corresponde al robot. Se presenta una evolución en el tiempo de distintas variables.

En la figura 5.1 que se va a analizar, el trazado rojo corresponde al robot y el trazado azul corresponde a la marca. Se comienza explicando la primera fila de la figura, en ella hay tres gráficas, que pasaran a ser explicadas a continuación más detalladamente.

– **Gráfica en x**: Esta gráfica (figura 5.2) representa la posición en el eje x (expresada en metros) a lo largo del tiempo. En ella podemos ver cómo la marca empieza en 0 y el robot empieza en -1; sin embargo, con el paso del tiempo estas dos líneas se encuentran superpuestas. También se puede observar cómo la longitud total recorrida es de 400 metros, ya que se simula durante 2000 segundos con una velocidad lineal de 0.2 m/s.

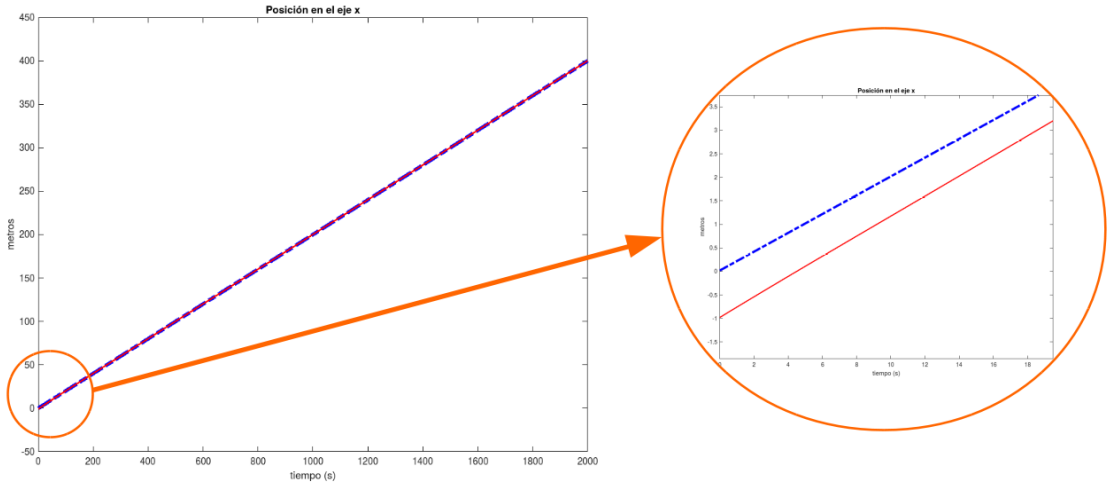

Figura 5.2: Ampliación de la posición en x

– **Gráfica en y**: Corresponde a la evolución de la posición en el eje y. En ella se puede ver como la marca inicialmente se encuentra en 0 y ese valor se mantendrá en el tiempo, debido a que esta solo se desplaza a lo largo del eje x. Sin embargo, el robot comienza en -0.5 metros y se puede ver como a lo largo del tiempo esta posición se va corrigiendo. Inicialmente se produce una sobreoscilación para posteriormente mantenerse constante al valor de cero igual que esta la marca.

– **Gráfica en θ**: Representa la posición del ángulo theta. Como se puede ver para la marca este ángulo será constante a cero, mientras que con respecto al robot este ángulo comenzará en cero y sobreoscilara un poco hasta estabilizarse en cero, igual que ha sucedido con la posición en el eje y. Este giro se debe a que el robot tiene que corregir el error en el eje y, ya que como hemos dicho antes en el eje y comienza en -0.5.

En la segunda fila nos encontramos con:

– **Gráfica velocidad lineal y Gráfica velocidad angular**: Representan la velocidad lineal y la velocidad angular respectivamente. Para la marca podemos observar como la primera de ellas se mantiene constante en 0.2 m/s que es el valor que le hemos impuesto y en la segunda de ellas se mantiene constante en cero ya que no presenta movimiento angular. Con respecto al robot vemos que la velocidad lineal comienza en 0.22 m y con el paso del tiempo va descendiendo hasta mantenerse constante en 0.2 m/s, valor acorde a la velocidad lineal de la marca. Para la velocidad angular del robot, también podemos observar que comienza en 3 grados y sobreoscila hasta mantenerse constante en cero.

– **Gráfica x-y**: Representa la evolución del eje x con respecto al eje y, que presentan tanto la marca como el robot. Podemos observar que la marca al ser un movimiento lineal en el eje x se mantiene constante en cero para el eje y, mientras que para el eje x va aumentando su valor. Con respecto al robot vemos que como se ha comentado en las gráficas anteriores al iniciarse con una posición en el eje y distinta de cero, se produce una sobreoscilación para compensar ese error y finalmente se estabiliza en cero siguiendo a la marca.

En la tercera fila de la figura 5.1, nos encontramos con tres gráficas.

– **Gráfica error en x**: Representa el error en el eje x. En ella se puede observar que comienza existiendo un error de -1 m, y con el paso del tiempo ese error tiende a cero y se mantiene constante en dicho valor. De las variables del controlador, k_1 es la que hará que ese error tienda más rápidamente a cero. A continuación, se muestra gráficamente como para distintos valores de k_1 ese error tiende con mayor o menor rapidez a cero.

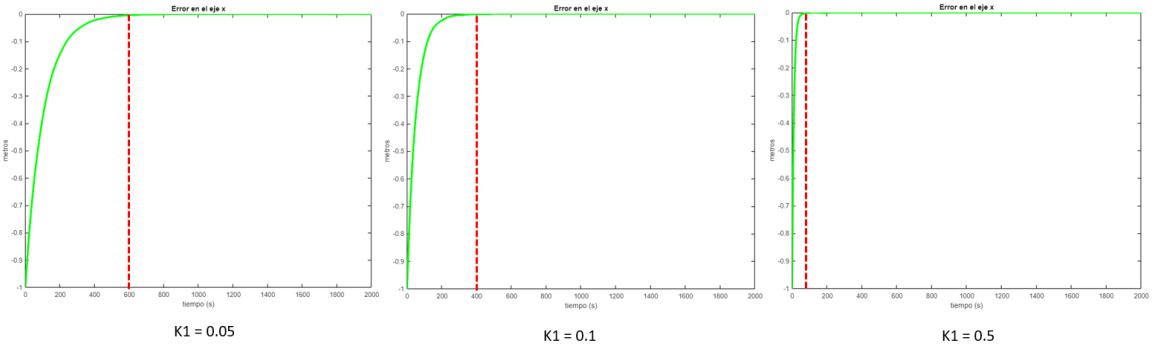

Figura 5.3: Error en x con distintos valores de k1

De las gráficas (Figura 5.3) podemos observar que contra menor sea el valor de k_1, más se tardará en alcanzar el régimen permanente. Así para un valor de k_1=0.05 alcanzamos un error en x constante a cero en 600 segundos, con $k_1 = 0.1$ tardará 400 segundos y para $k_1 = 0.5$ tardará 100 segundos.

– **Gráfica error en y**: Representa el error en el eje y. En este caso se puede observar cómo comienza con un error de -0.5 metros, ya que es el parámetro que se ha puesto como condición al inicio, para después sobreoscilar un poco hasta estabilizarse en cero de manera permanente. El error en el eje y se corrige mediante la constante del regulador k_2.

– **Gráfica error en θ**: Representa el error en el ángulo theta. En esta gráfica podemos ver como parte de un error de 0°, pero al existir una variación en el eje y, el error en el ángulo theta sobreoscila igual que sucede en el error en el eje y hasta estabilizarse en cero. Este error que se produce en el ángulo theta se puede controlar con el parámetro k_3 del controlador.

5.1.2. Movimiento circular

La siguiente trayectoria que se ha simulado es un movimiento circular. A continuación se muestra la figura 5.4 donde se recogen todas las gráficas que demuestran que el controlador funciona adecuadamente para un círculo.

Los valores elegidos para representar este movimiento circular son una velocidad lineal de 0.2 m/s y una velocidad angular de 1.8 grados/segundo. Los valores escogidos para las ganancias de los reguladores son 0.5 para k_1, 0.5 para k_2 y 1.0 para k_3. Además, la marca partirá de una orientación 0 y una posición 0 para los ejes x e y; mientras que el robot se iniciará con una orientación de cero y una posición de -1 para el eje x y una posición de 0 para el eje y.

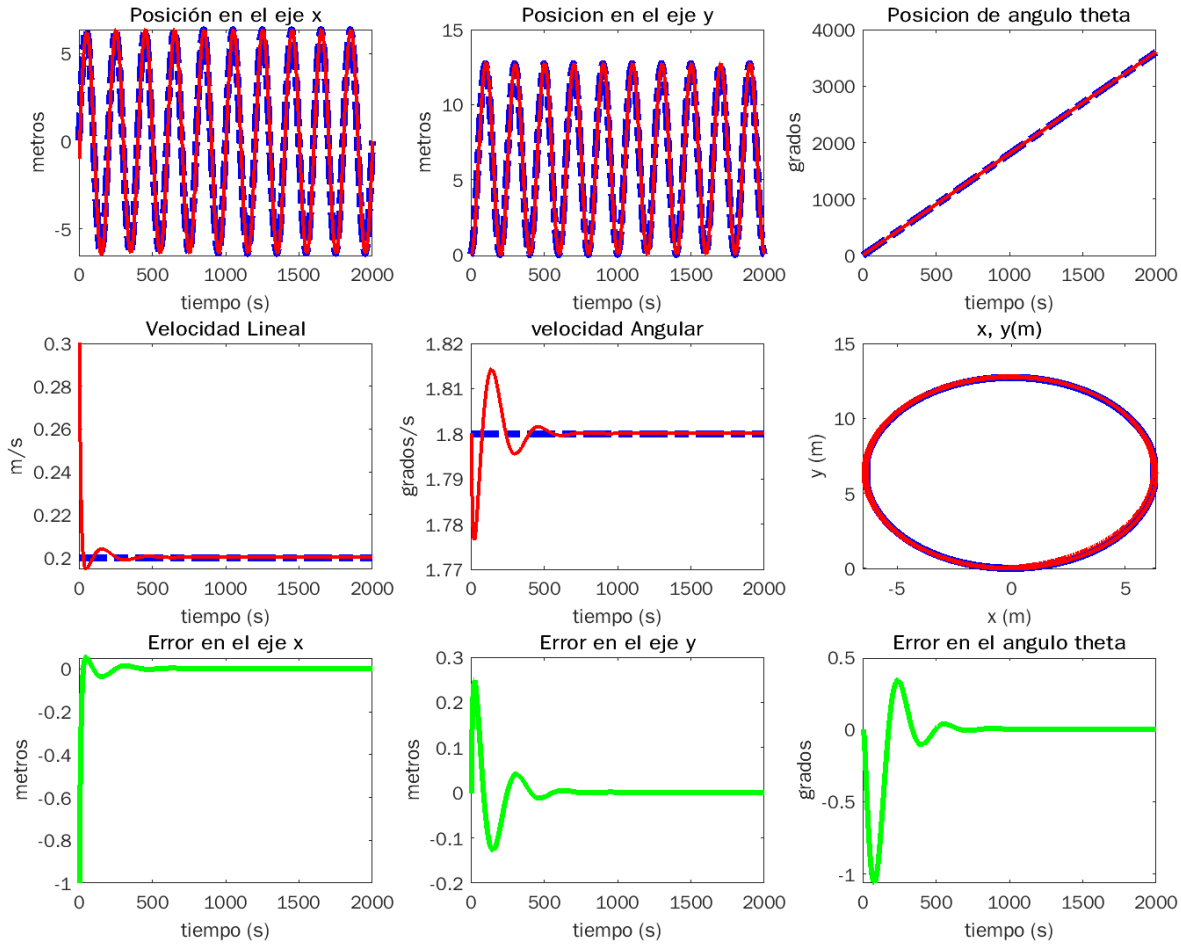

Figura 5.4: Trayectoria Circular

En la figura 5.4 nos encontramos con 9 gráficas, las cuales representan lo mismo que las explicadas en el movimiento lineal y por lo tanto en este caso se destacará lo más relevante de ellas.

Observando las tres últimas gráficas, en las que se muestran los errores de posición y orientación, se puede decir que los parámetros elegidos resultan adecuados para cumplir con el objetivo deseado, ya que los tres errores tienden a cero y se mantienen constantes en ese valor.

También se puede ver que el robot realiza un seguimiento adecuado de la trayectoria en la gráfica tercera de la segunda fila, ya que esta representa la evolución del eje x con respecto al eje y, en la cual no se ve que la línea roja se salga de la trayectoria que mantiene la línea azul.

5.1.3. Movimiento sinusoidal

Otra de las trayectorias que se van a probar es la de un movimiento sinusoidal. A continuación, se muestra el correcto seguimiento del robot a dicha trayectoria con los siguientes valores de k_1, k_2 y k_3: 0.01, 0.5, 0.5 respectivamente.

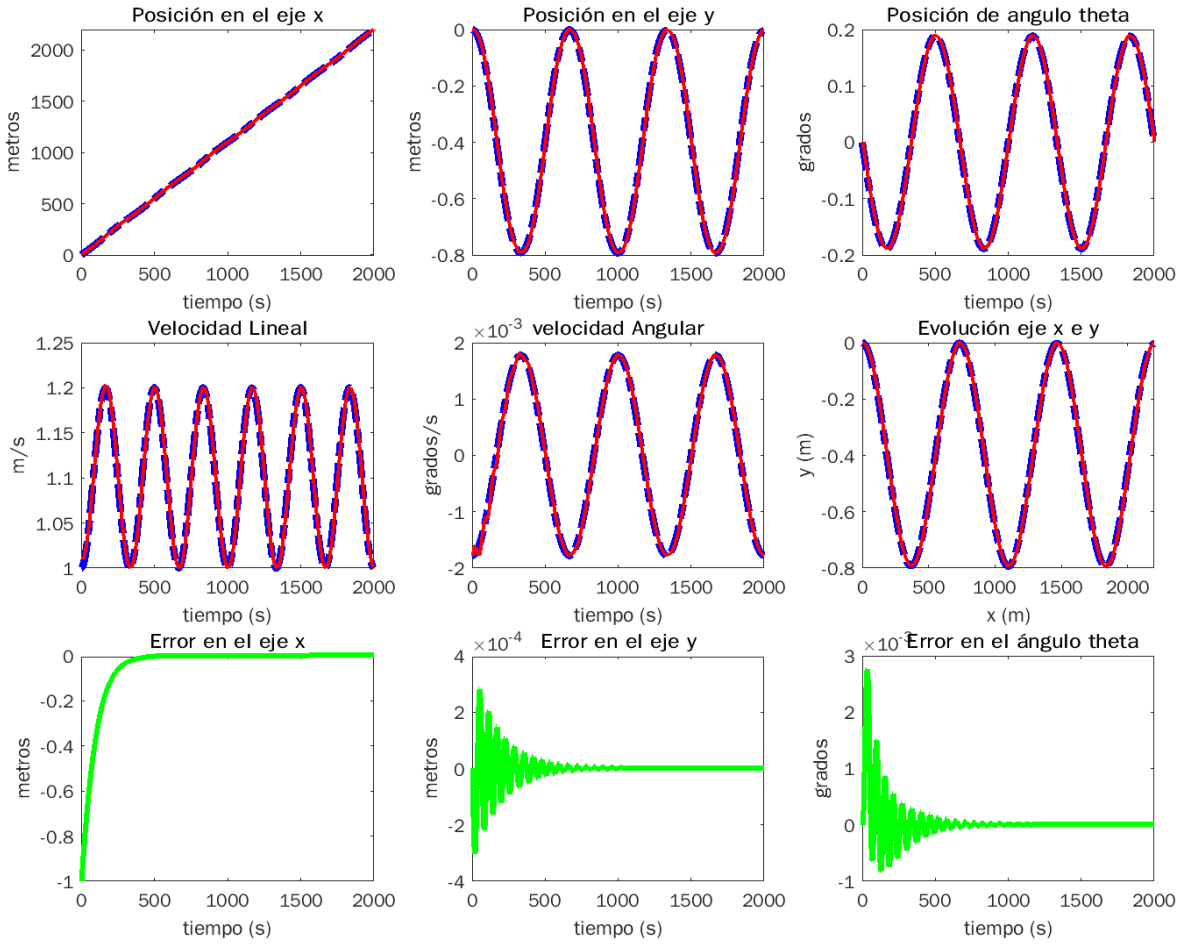

Figura 5.5: Trayectoria sinusoidal

En la figura 5.5 se puede ver como en la gráfica de la evolución del eje x con respecto al eje y, la línea roja (robot) sigue correctamente a la línea azul (marca) y por lo tanto podemos decir que el robot sigue correctamente a la marca ArUco. Esto que se comenta también se puede confirmar con las gráficas de los errores en las que se puede ver como los tres errores se estabilizan en cero. Además, se puede destacar que en los errores en el ángulo θ y en el eje y, se aprecia inicialmente una sobreoscilación. Esta es debida a que la trayectoria de referencia va de un lado a otro sinusoidalmente.

5.1.4. Movimiento en forma de espiral

A continuación, se explica un movimiento en forma de espiral. Para llevar a cabo un buen seguimiento se han probado diversos valores para los parámetros k_1,k_2 y k_3 y finalmente los valores elegidos han sido 0.01,1.0 y 1.0 respectivamente. A continuación, se muestra la figura en la que se encuentran las nueve gráficas que ayudan a visualizar si el seguimiento es el correcto.

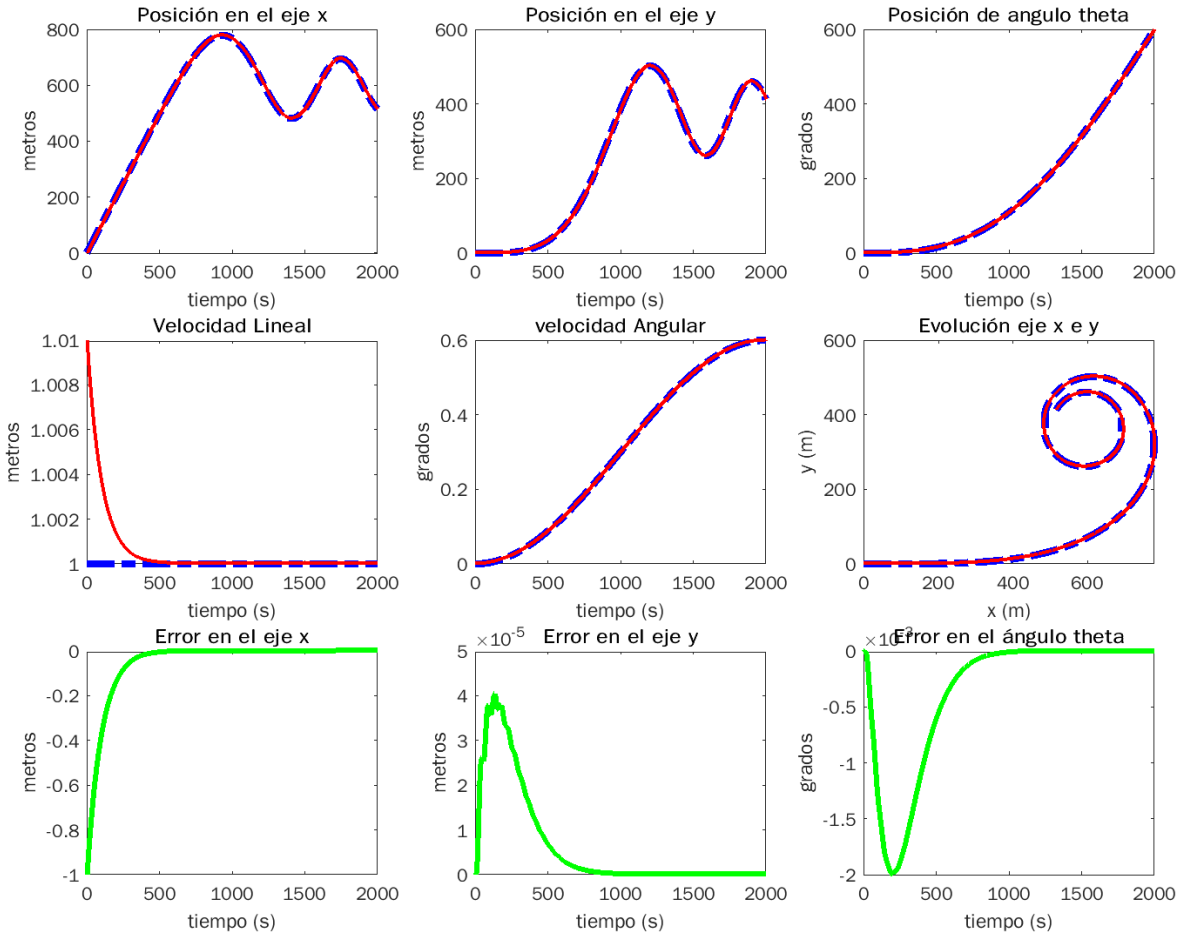

Figura 5.6: Trayectoría en forma de espiral

En la figura 5.6 se puede observar, al igual que lo que ocurría en las trayectorias anteriores, que el robot sigue perfectamente a la marca y que los errores se estabilizan para un valor de cero.

5.1.5. Movimiento en forma de ocho

Finalmente se mostrará la simulación que se ha obtenido para una trayectoria en forma de ocho. Para realizar el seguimiento de dicha trayectoria, los valores de las constantes k_1, k_2 y k_3 son 0.5, 3 y 4 respectivamente. A continuación, se muestra la figura que recoge un conjunto de representaciones gráficas.

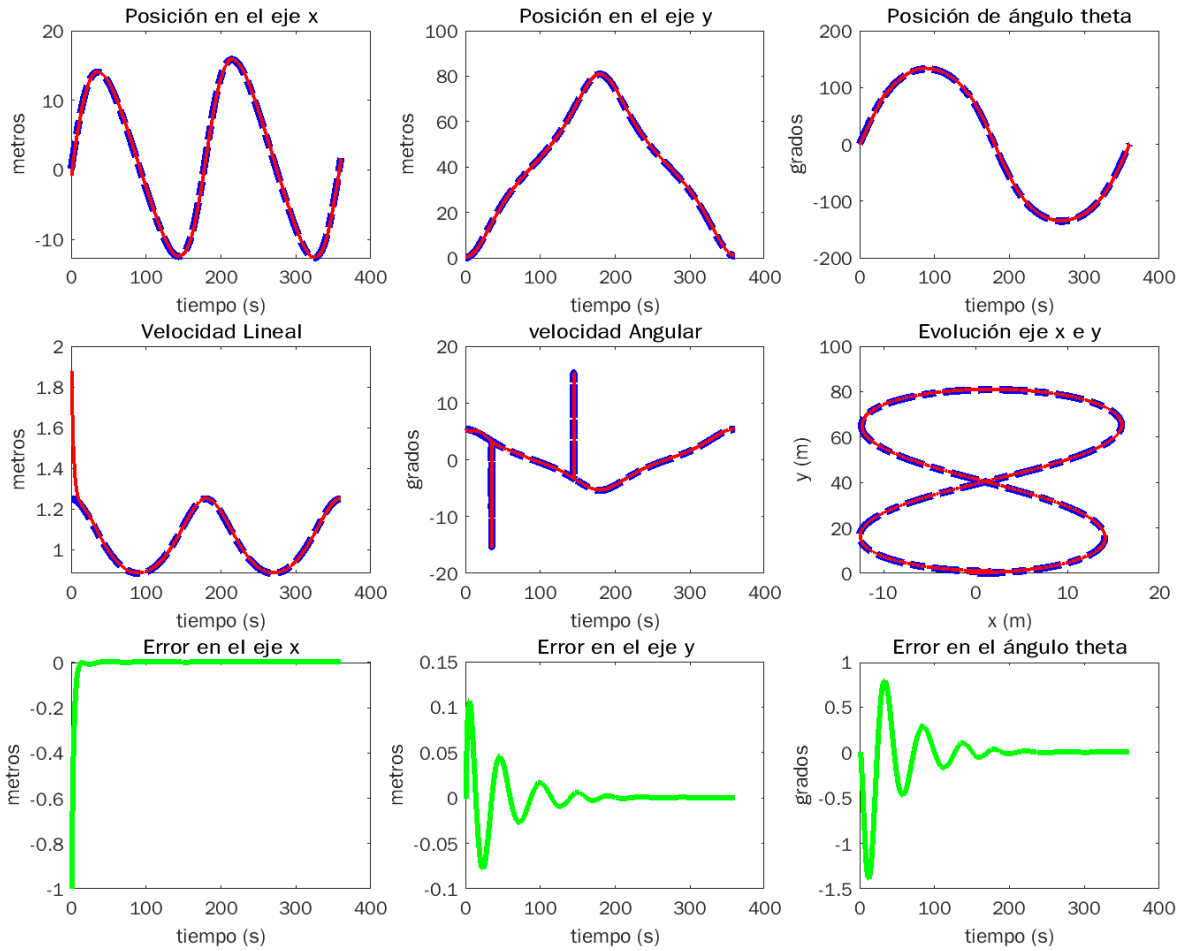

Figura 5.7: Trayectoria en forma de ocho

En esta figura 5.7 se pueden ver cómo evoluciona en posición y en orientación el robot (línea roja) y la marca (línea azul) con respecto al tiempo. También se puede ver cómo evoluciona el eje x con respecto al eje y dando lugar a la trayectoria que está describiendo en forma de ocho, y en esta gráfica se comprueba como con los parámetros seleccionados se obtiene un buen resultado.

Finalmente, se recogen en la siguiente tabla 5.1 los valores elegidos para los parámetros k_1, k_2 y k_3 en las distintas trayectorias.

Trayectorias	k_1	k_2	k_3
Lineal	0.1	30.0	35.0
Circular	0.5	0.5	1.0
Sinusoidal	0.01	0.5	0.5
Espiral	0.01	1.0	1.0
En forma de ocho	0.5	3.0	4.0

Tabla 5.1: Parámetros para las distintas trayectorias

Como se puede observar en la tabla, los parámetros elegidos para las distintas trayectorias son de un orden bastante similar, excepto los valores de k_2 y k_3 para la trayectoria lineal que son más elevados. Esto es debido a que en la simulación de dicha trayectoria el robot comenzaba desplazado 0.5 metros en el eje y con respecto a la marca, y por lo tanto para obtener en el tiempo de simulación un buen resultado en el que no existieran muchas sobreoscilaciones se han elegido esos valores.

Como conclusión podemos decir que, si las condiciones iniciales para las trayectorias son iguales, los parámetros elegidos podrán ser los mismos. Aunque si queremos una mayor precisión de seguimiento deberemos ajustar los parámetros.

5.2. Resultados en ROS

En esta sección se mostrarán los resultados obtenidos tras realizar diversas simulaciones para distintos tipos de movimientos. Estas trayectorias serán las mismas que se han llevado a cabo en el apartado anterior de Matlab.

Para simular dichas trayectorias en ROS hemos tenido que crear una marca con movimiento. Al principio dicha marca fue creada con un tamaño de 12 cm, aunque se vio que tenía un poco de ruido y se decidió probar con otros tamaños. Finalmente, se observó que para un tamaño de 20 cm presentaba un poco de menos ruido y por lo tanto se decidió que las simulaciones se realizaran con ese tamaño. Además, otra de las ventajas que presenta el tamaño elegido es que es muy similar al que tienen las marcas en la ejecución real.

A continuación se muestra dos figuras, la primera de ellas (Figura 5.8) es cómo se ve la marca de 12 cm a una distancia de 1 metro de la cámara y la posición que se obtiene del movimiento de la marca en el eje y. En la segunda (Figura 5.9) se muestra lo mismo pero para una marca de 20 cm.

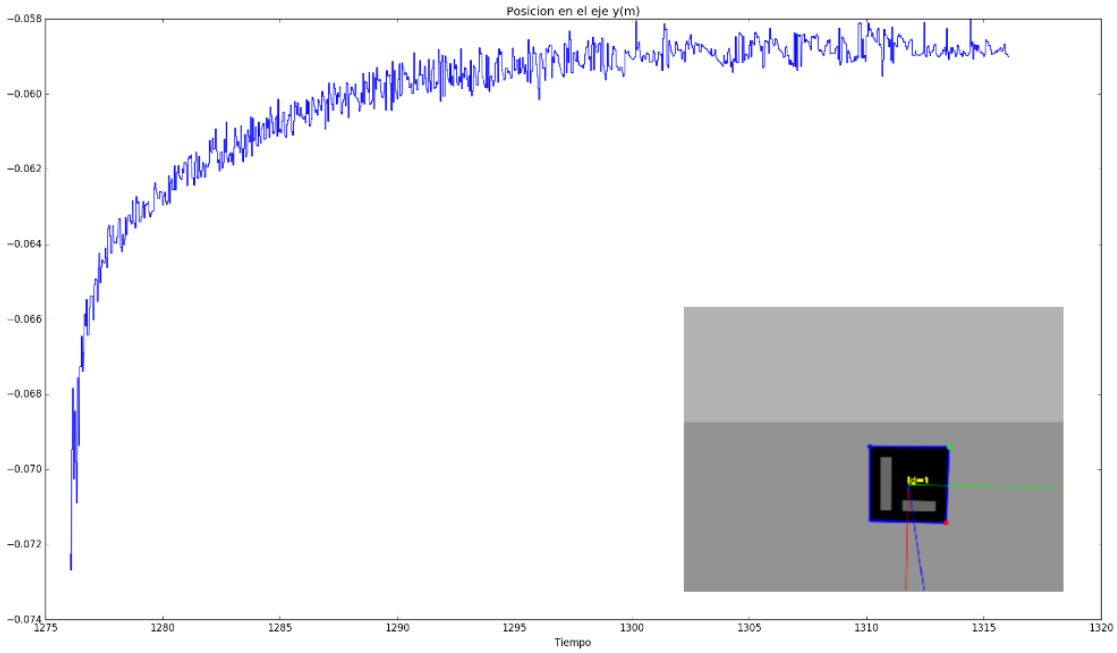

Figura 5.8: Marca de 12 cm

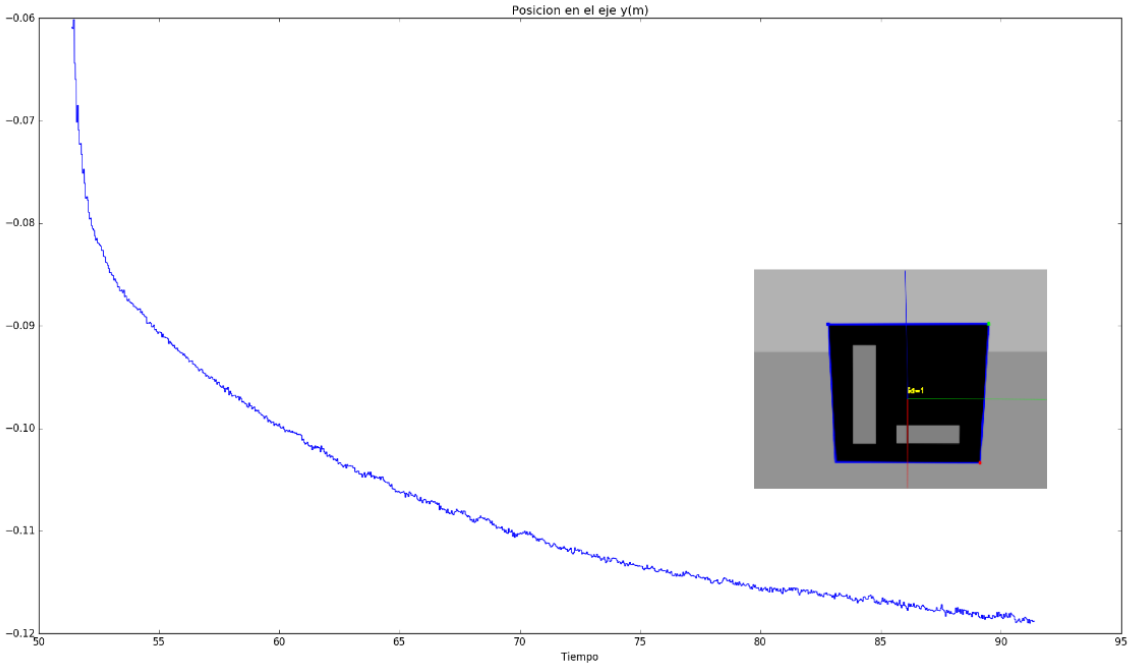

Figura 5.9: Marca de 20 cm

Una vez elegido el tamaño de la marca, quedará ajustar los parámetros del controlador para encontrar los valores que se ajusten correctamente para cumplir con el objetivo deseado.

A continuación, se muestran los resultados obtenidos y los valores elegidos.

5.2.1. Movimiento lineal

Para la trayectoria lineal se han llevado a cabo diferentes pruebas de simulación con el objetivo de encontrar los parámetros que mejor cumplan los objetivos.

Los experimentos que se han llevado a cabo se recogen en esta tabla:

Número Prueba	k_1	k_2	k_3
Prueba 1	0.1	50.0	60.0
Prueba 2	0.2	40.0	60.0
Prueba 3	0.2	20.0	60.0
Prueba 4	0.2	20.0	30.0
Prueba 5	0.25	30.0	60.0

Tabla 5.2: Parámetros movimiento lineal

De las cinco pruebas realizadas se decide quedarse con la prueba 3, por ello los valores de los parámetros elegidos son: k_1= 0.2, k_2=20.0 y k_3=60.0.

A continuación, se muestran las gráficas y se explica el porqué de quedarse con esta solución.

La primera de las gráficas (Figura 5.10) es la evolución tanto del robot (línea roja), como de la marca (línea azul) en el eje x con respecto al eje y.

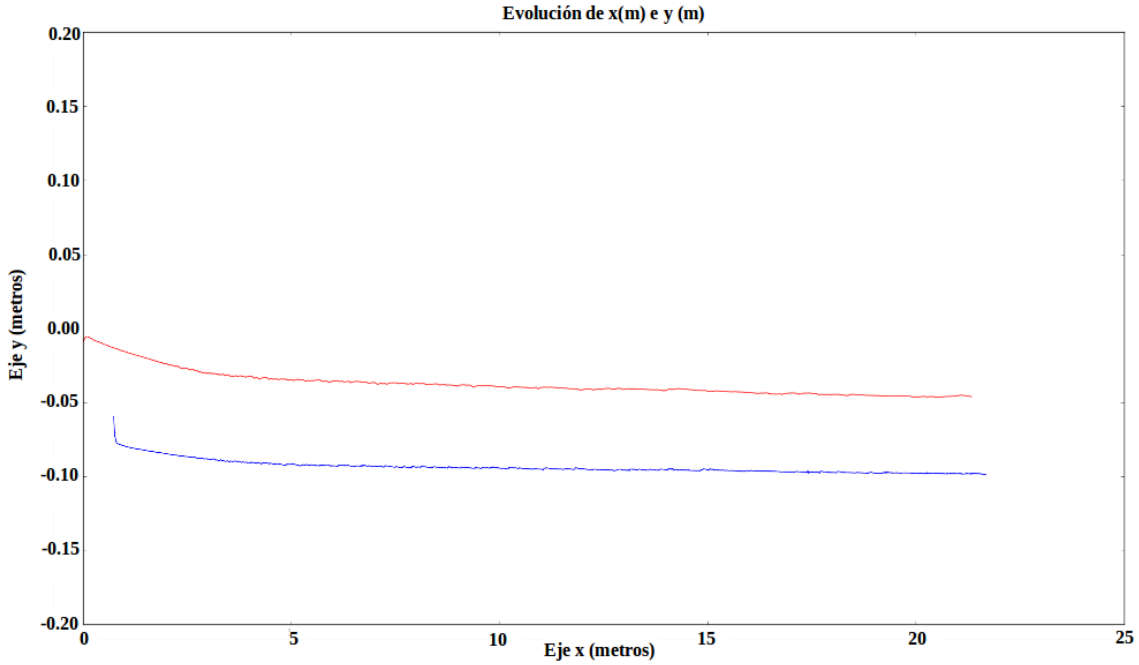

Figura 5.10: Evolución del eje x con respecto al eje y en el movimiento lineal

En esta figura 5.10 podemos ver como el robot comienza en cero en el eje x, mientras la marca comienza a un metro. Por otro lado, vemos como ambas líneas evolucionan de una forma constante lo que nos indica que el robot va por detrás de la marca en todo momento.

En las siguientes gráficas se recogen los errores tanto en el eje x, como en el eje y; y el error en la orientación.

En la figura 5.11 vemos como el error en el eje x se mantiene constante entre -0.8 y -0.9 metros.

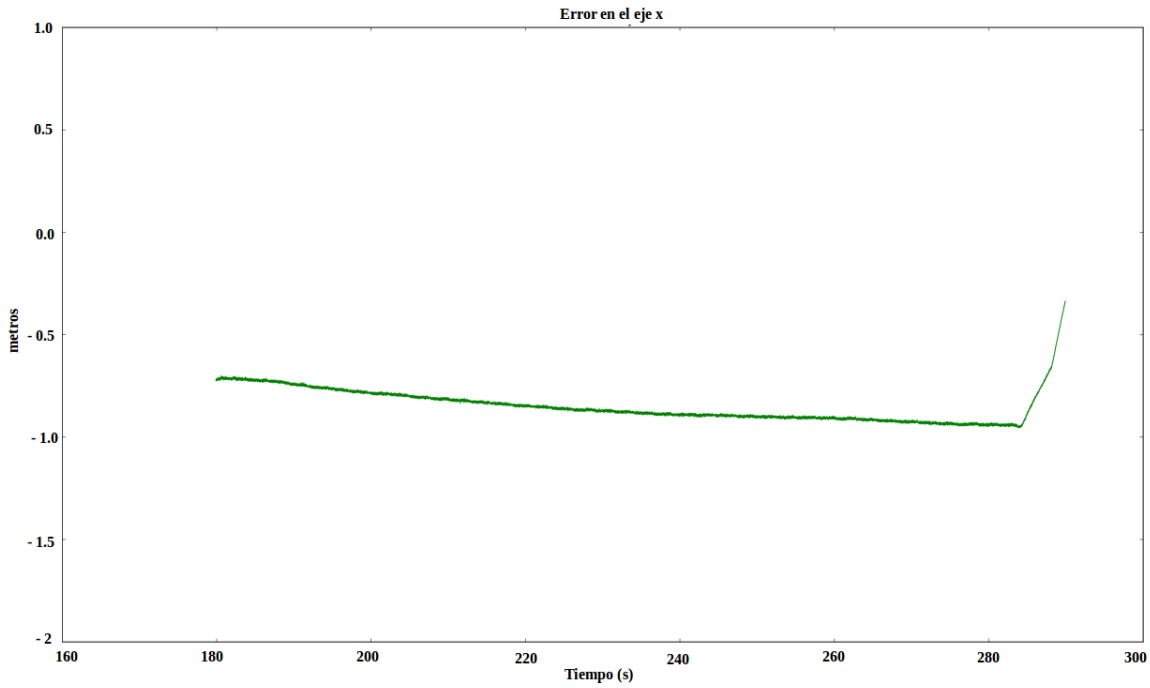

Figura 5.11: Error en el eje x (m) para una trayectoria lineal

En la figura 5.12 se ve el error en el eje y. Podemos apreciar como dicho error se mantiene constante en un valor de 0.07 metros. En dicha gráfica se puede ver que hay unos picos, a los que llamaremos ruido y que se consideran despreciables al no variar unas cantidades muy elevadas.

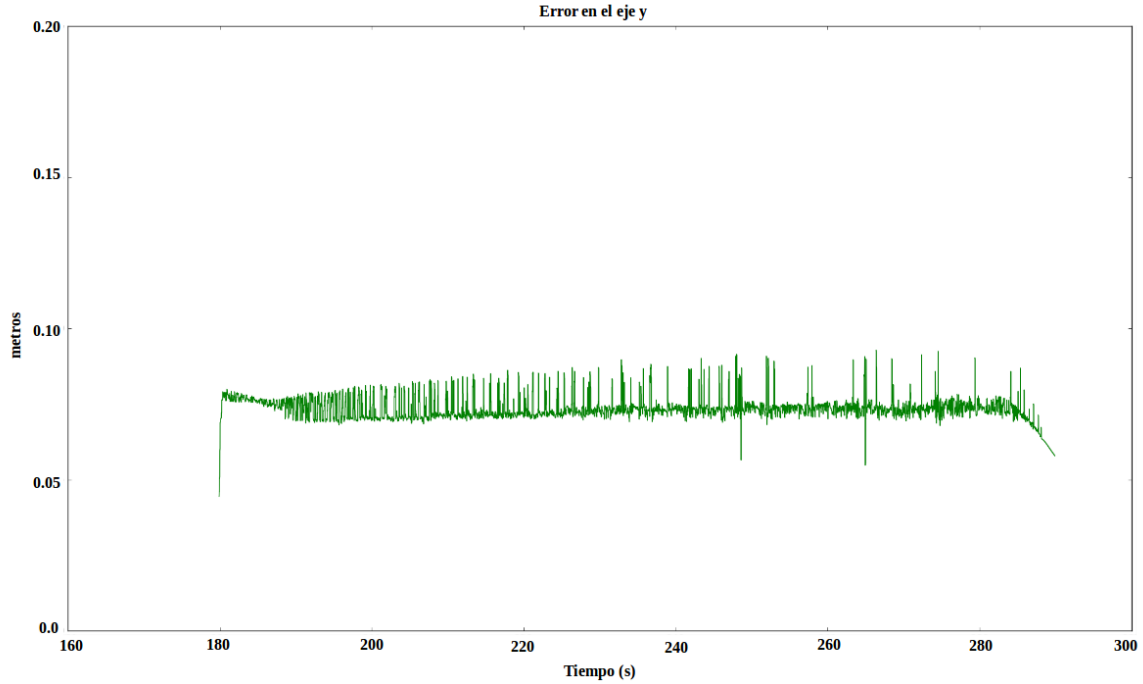

Figura 5.12: Error en el eje y (m) para una trayectoria lineal

En la figura 5.13 podemos ver el error en el ángulo theta, vemos cómo éste también se mantiene constante en torno a -1 grado. En este caso también se aprecia ruido como en el eje y.

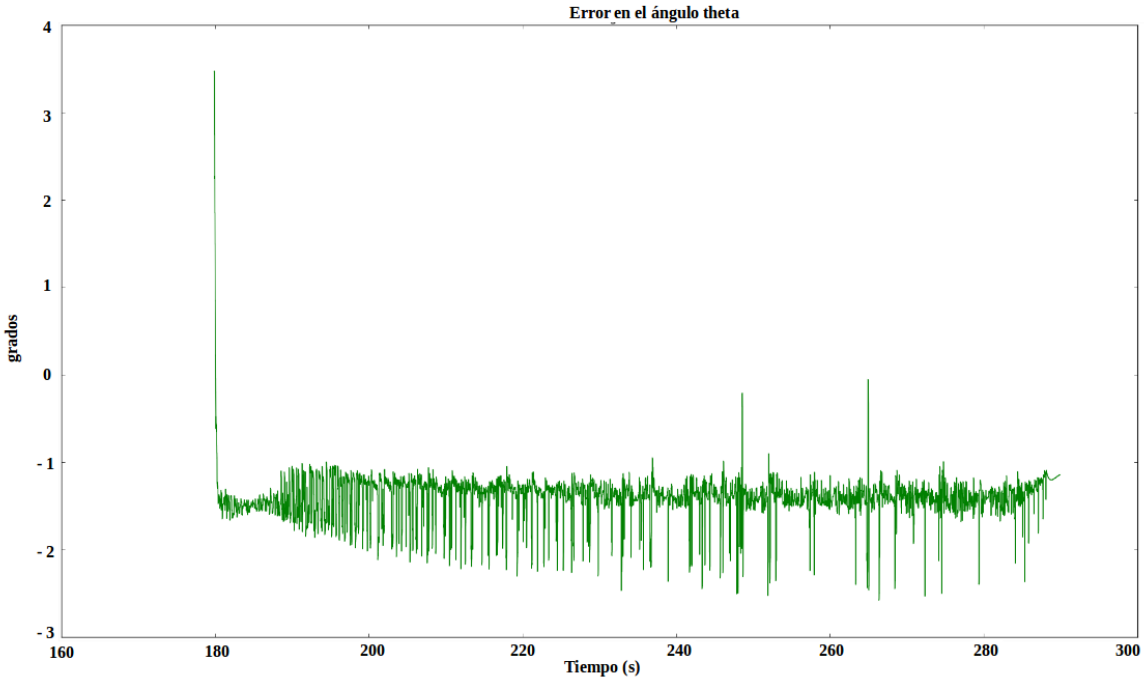

Figura 5.13: Error en la orientación para una trayectoria lineal

Como conclusión de estas gráficas se obtiene que con los valores de los parámetros

elegidos se obtiene un buen resultado para el algoritmo de control. Esto se puede ver en que los tres errores se mantienen constantes y además en la gráfica de la evolución del eje x con respecto al eje y también se aprecia que el robot sigue a la marca.

Finalmente se muestran unas imágenes (figura 5.14,5.15 y 5.16) tomadas en distintos puntos de la trayectoria, en Gazebo.

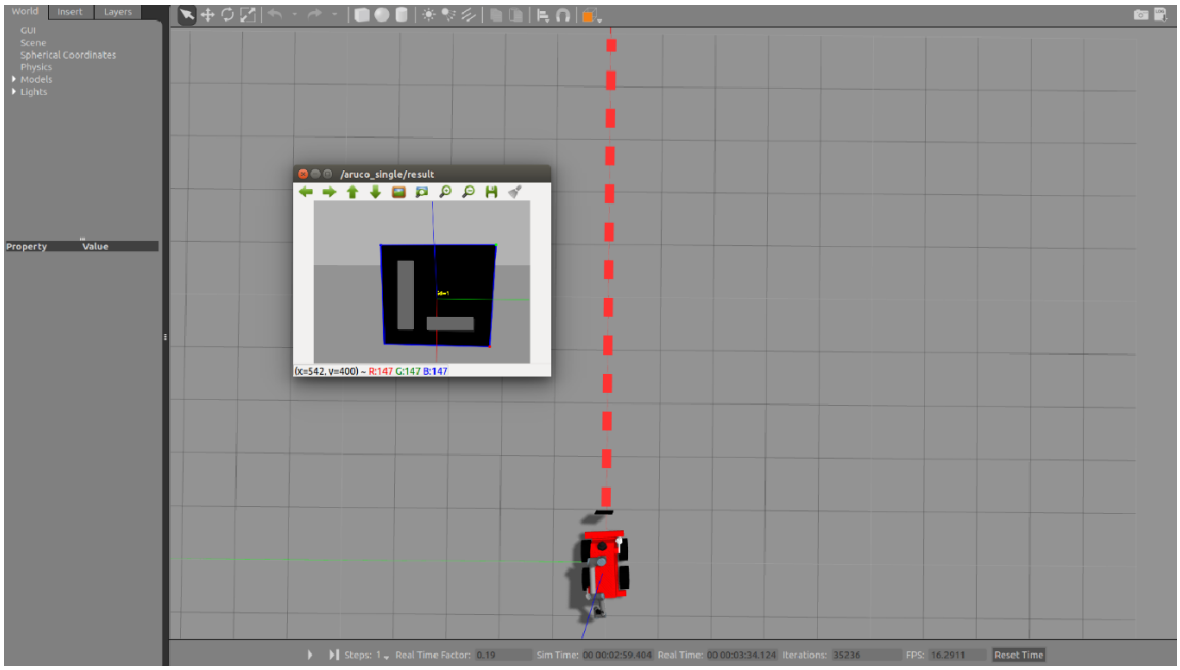

Figura 5.14: Posición inicial del movimiento lineal

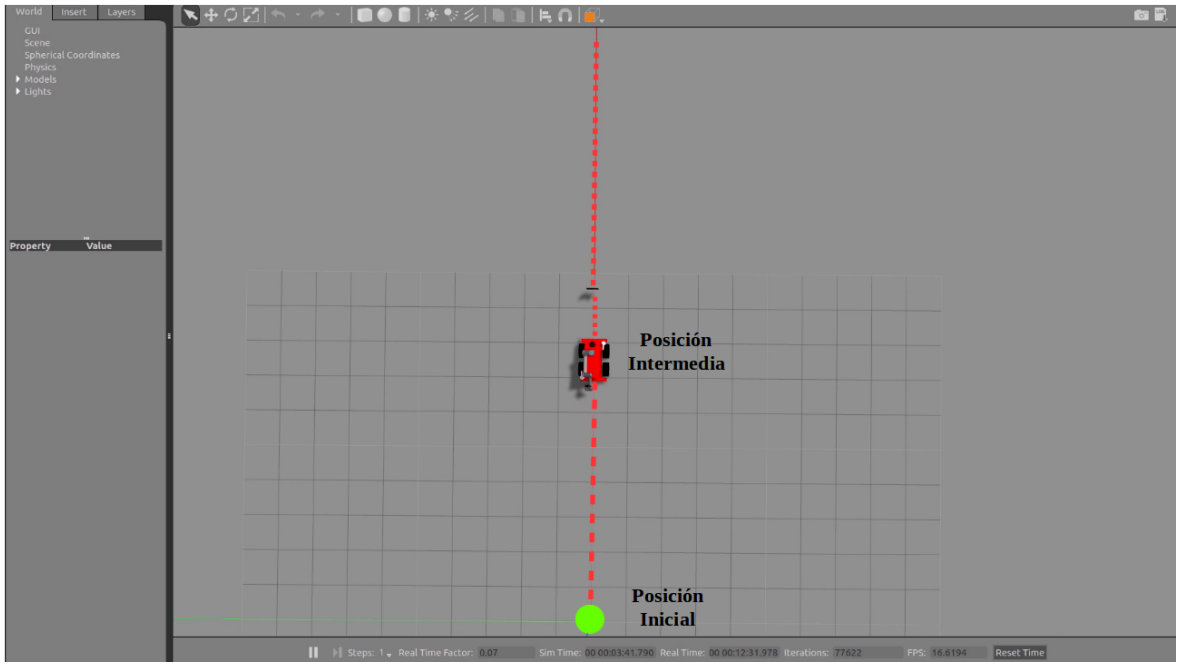

Figura 5.15: Posición intermedia del movimiento lineal

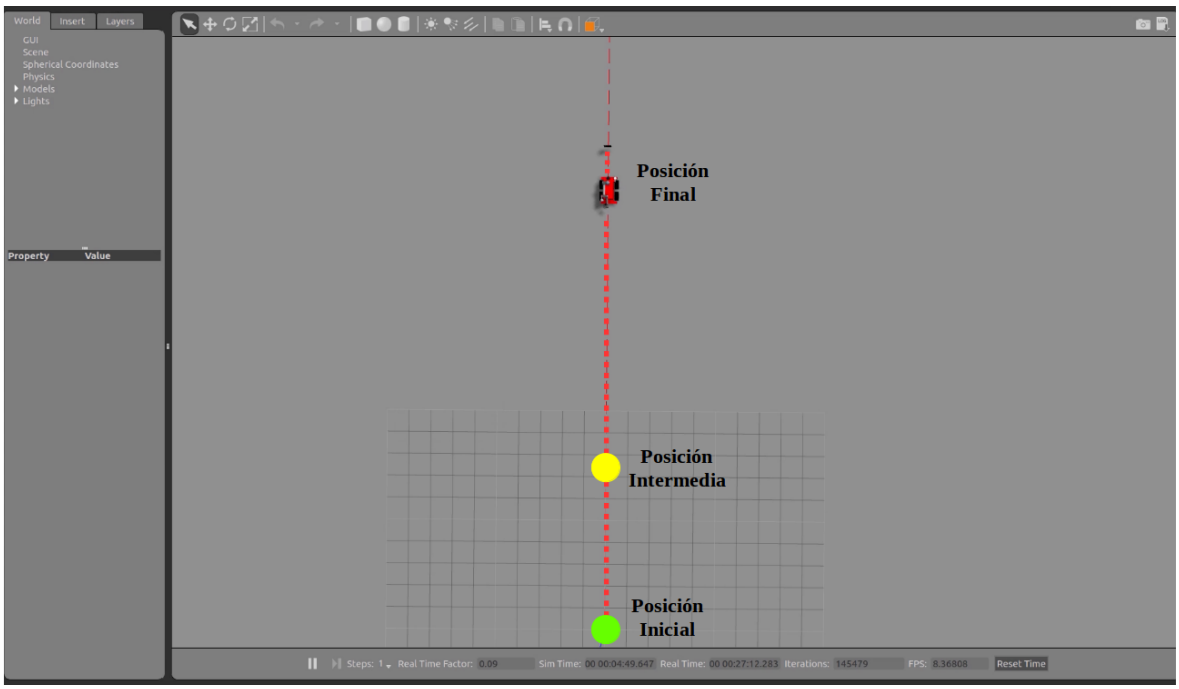

Figura 5.16: Posición final del movimiento lineal

5.2.2. Movimiento circular

Para una trayectoria circular se han llevado a cabo los siguientes experimentos, los cuales se recogen en esta tabla 5.3:

Número Prueba	k_1	k_2	k_3
Prueba 1	0.2	20.0	30.0
Prueba 2	0.2	70.0	80.0
Prueba 3	0.2	60.0	80.0
Prueba 4	0.2	50.0	70.0
Prueba 5	0.2	80.0	90.0
Prueba 6	0.2	70.0	95.0

Tabla 5.3: Parámetros movimiento circular

De las seis pruebas realizadas, se elige la prueba 3. Los valores de los parámetros son los siguientes: k_1= 0.2, k_2=60.0 y k_3=80.0.

A continuación, se muestran las gráficas que se han obtenido. En la figura 5.17 se puede ver la evolución de eje x con respecto al eje y. De esta figura se puede decir que el robot (línea roja) comienza un metro más atrás que la marca (línea azul). También observamos como el robot sigue perfectamente a la marca describiendo un círculo y finalmente cuando la marca se para el robot también, ya que este algoritmo no tiene ningún sentido si la marca no está en movimiento.

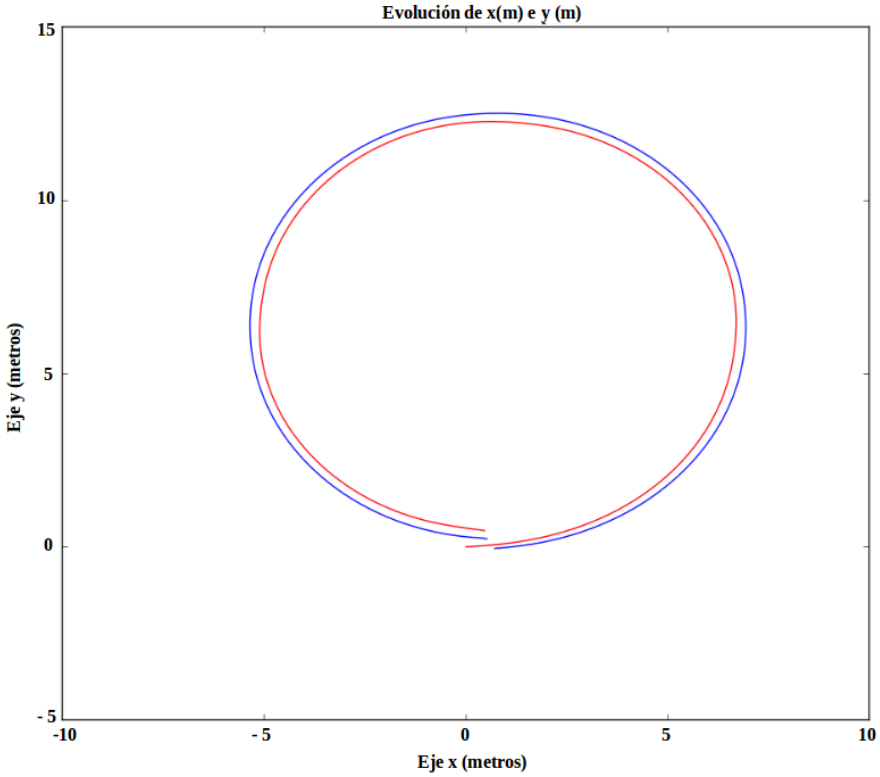

Figura 5.17: Evolución del eje x con respecto al y en la trayectoria circular

En las figuras 5.18 y 5.19 se pueden ver el error en el eje x y en el eje y respectivamente. Visualizando dichas figuras se puede decir que ambos errores se mantienen constantes y por lo tanto el controlador funciona de la manera deseada.

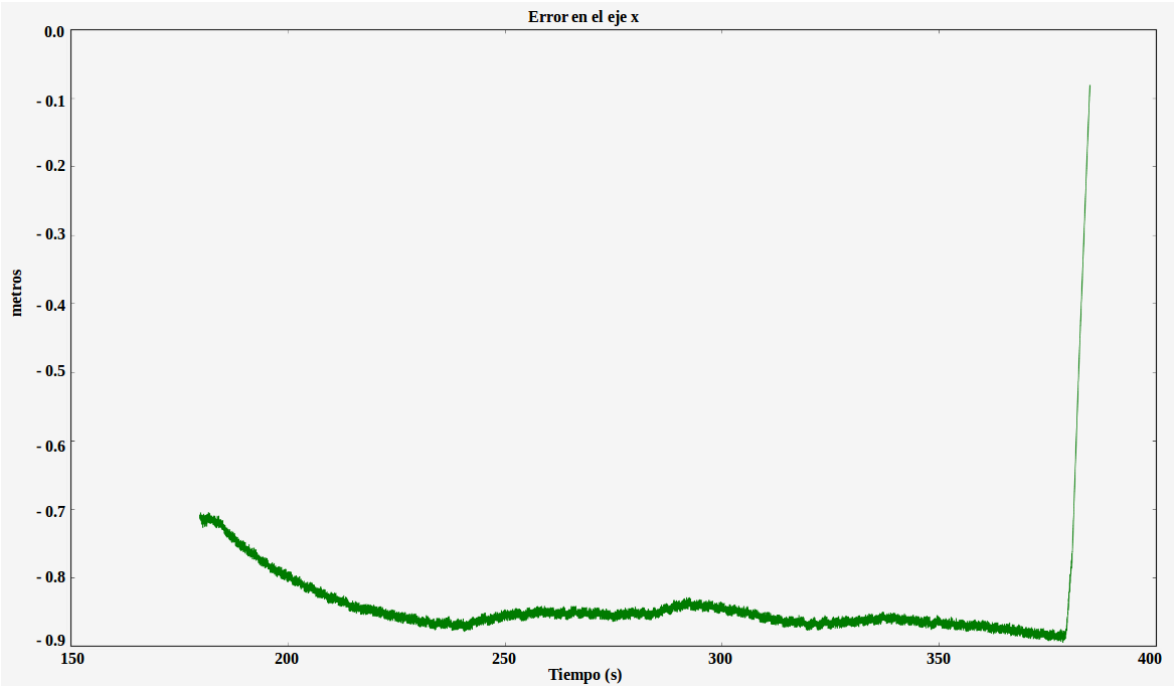

Figura 5.18: Error en el eje x en la trayectoria circular

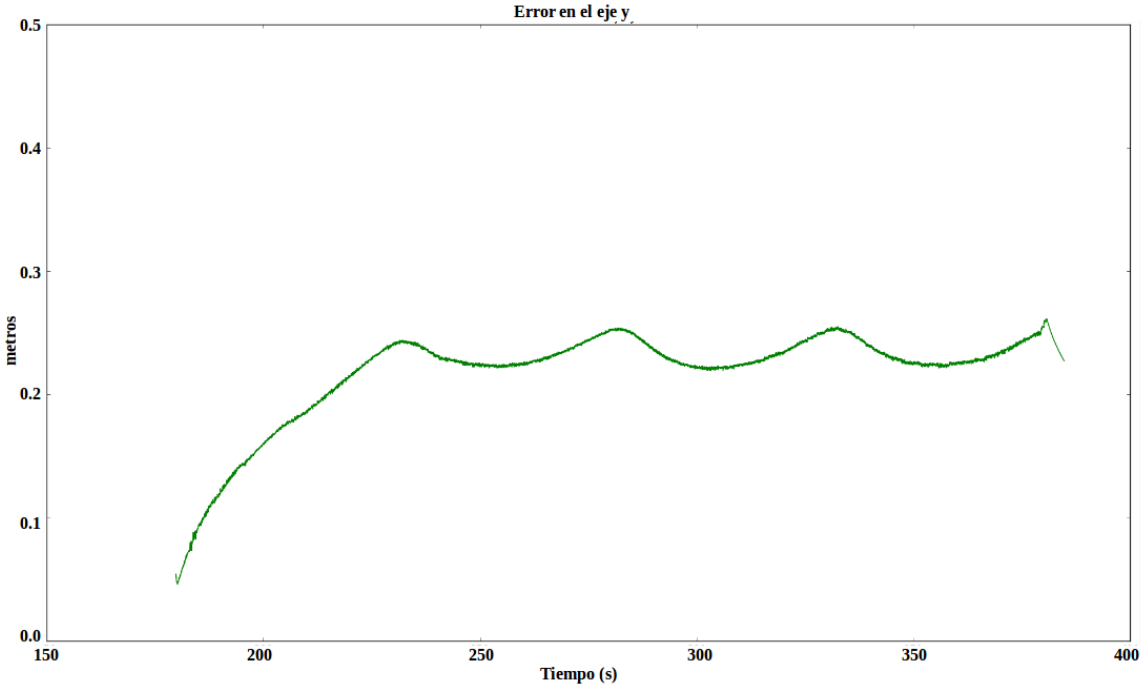

Figura 5.19: Error en el eje y en la trayectoria circular

A continuación, en la figura 5.20 se muestra la trayectoria circular realizada por el robot y la marca en el entorno de Gazebo.

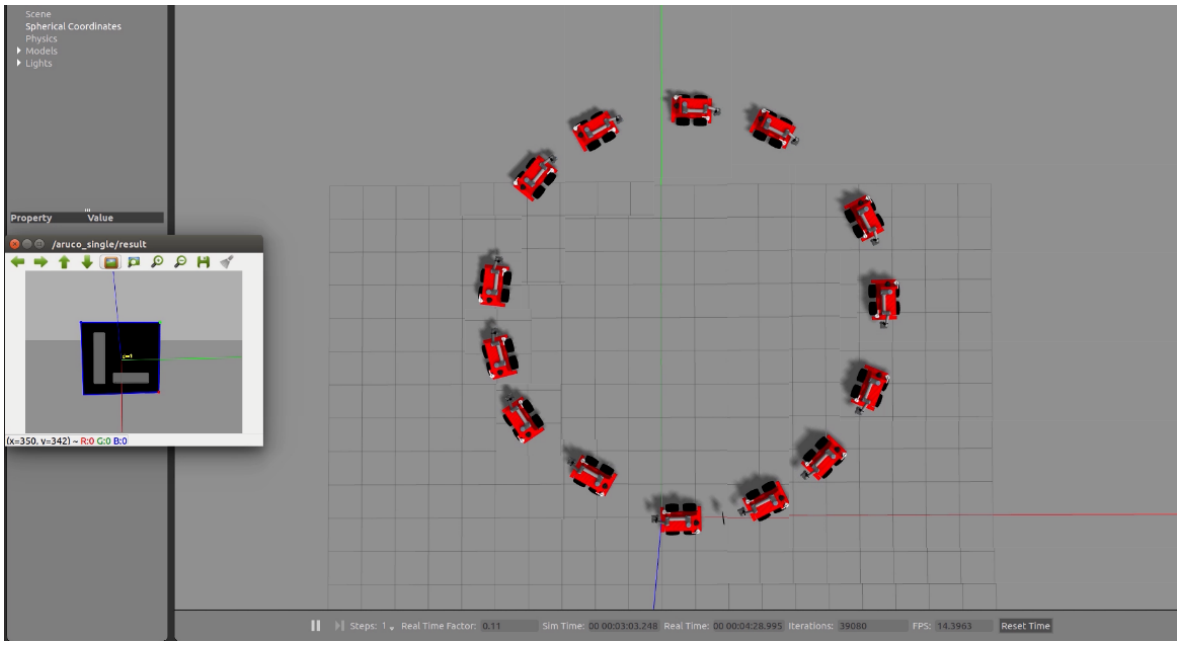

Figura 5.20: Trayectoria circular en gazebo

5.2.3. Movimiento sinusoidal

Los experimentos que se han llevado a cabo se recogen en esta tabla:

Número Prueba	k_1	k_2	k_3
Prueba 1	0.25	70.0	80.0
Prueba 2	0.30	70.0	80.0
Prueba 3	0.40	60.0	70.0
Prueba 4	0.40	70.0	80.0

Tabla 5.4: Parámetros movimiento sinusoidal

De las pruebas que se han realizado se elige la prueba 4 y por lo tanto los valores de los parámetros elegidos para k_1, k_2 y k_3 son 0.4, 70.0 y 80.0, respectivamente.

En la figura 5.21 se puede ver como el robot tiene una trayectoria sinusoidal igual que la marca y por lo tanto se puede decir que el robot sigue correctamente a la marca.

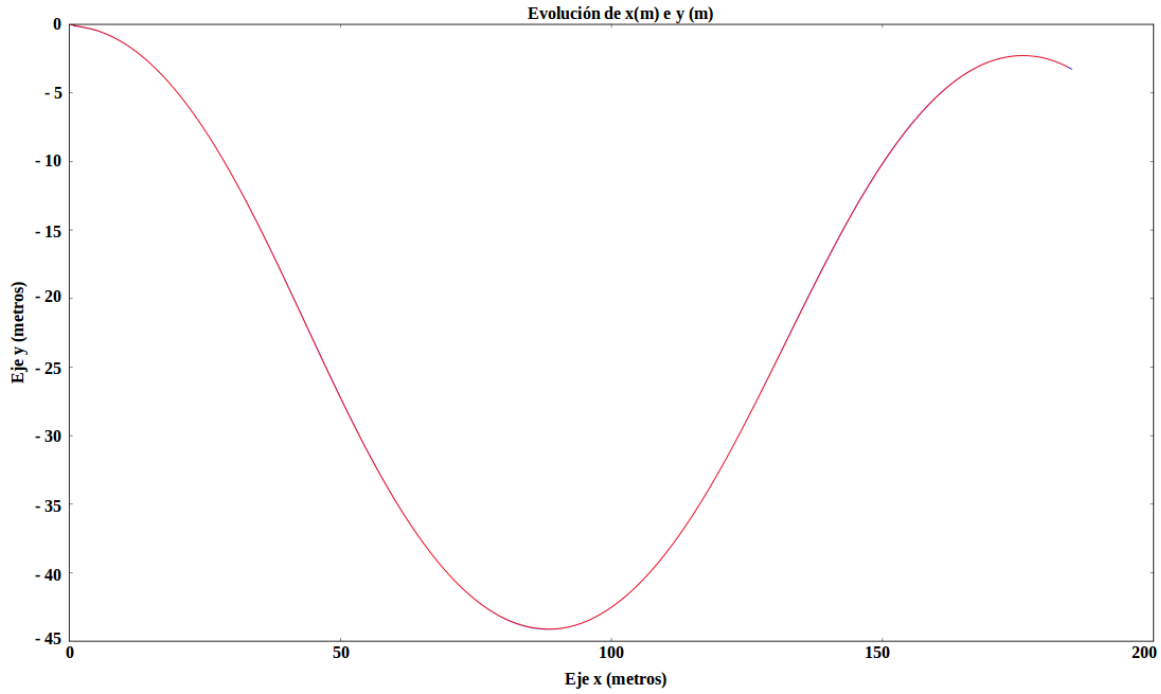

Figura 5.21: Evolución en el eje x y en el eje y en la trayectoria sinusoidal

Con respecto a las figuras 5.22 y 5.23 se puede decir que los errores tanto en el eje x como en el eje y, se mantienen constantes a lo largo del tiempo. Esto nos hace pensar que el algoritmo utilizado junto con los parámetros elegidos es correcto.

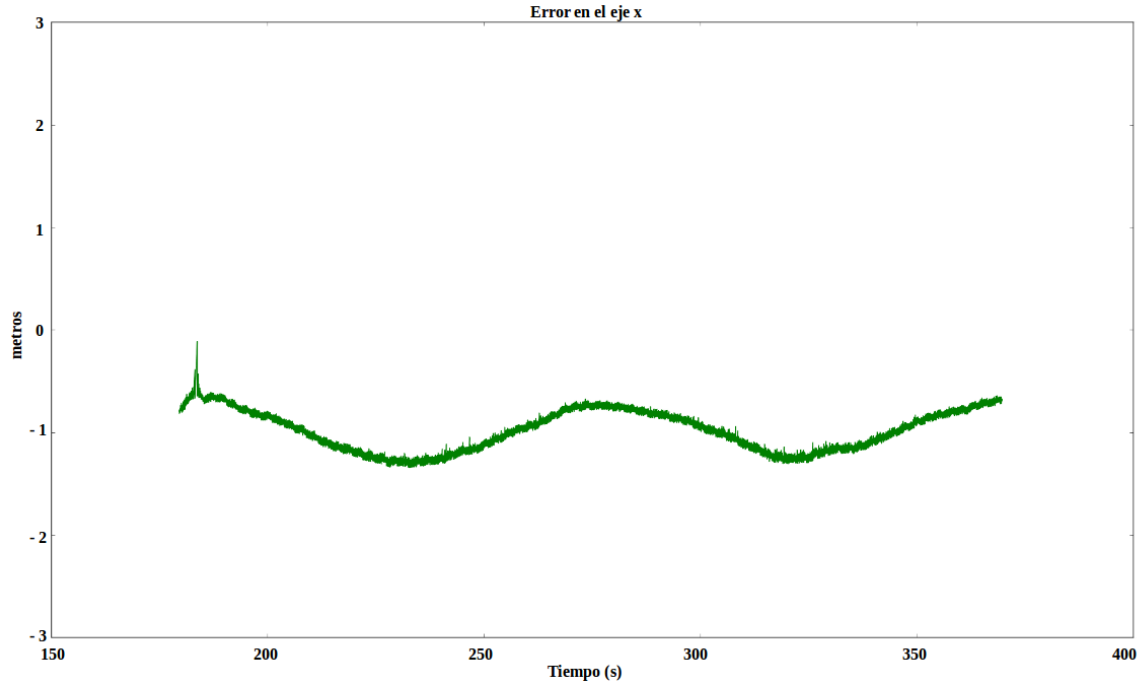

Figura 5.22: Error en el eje x en la trayectoria sinusoidal

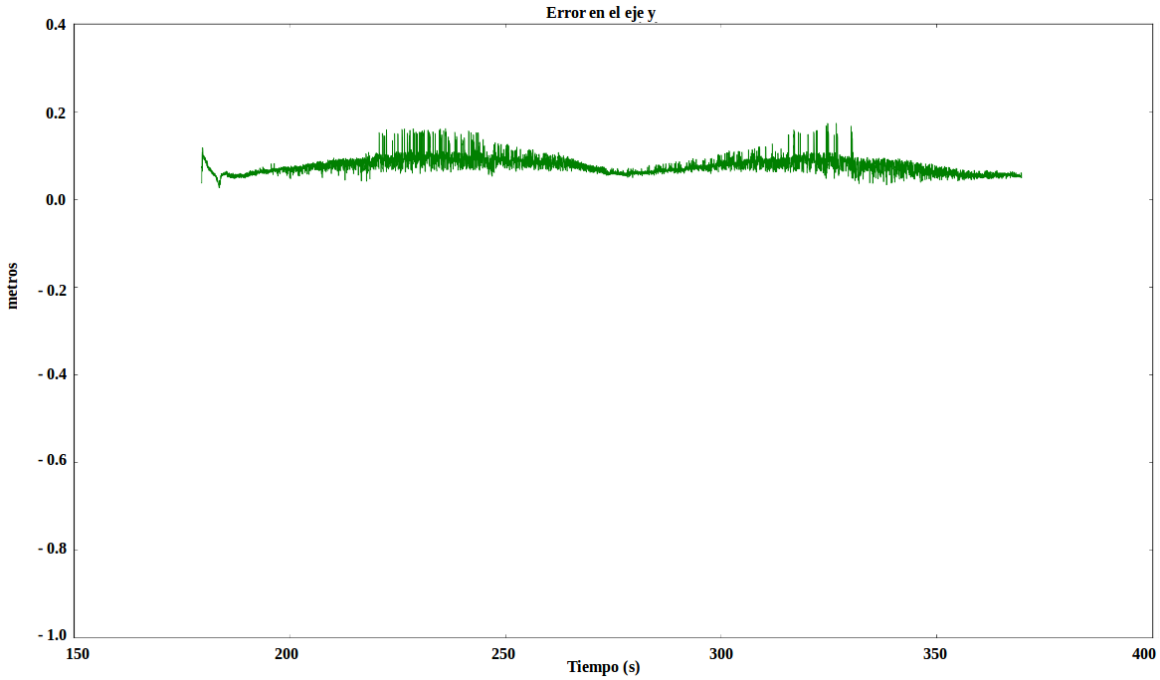

Figura 5.23: Error en el eje y en la trayectoria sinusoidal

Finalmente se muestra la figura 5.24 en la que se recogen una sucesión de fotogramas tomados en el entorno de Gazebo para dicha trayectoria.

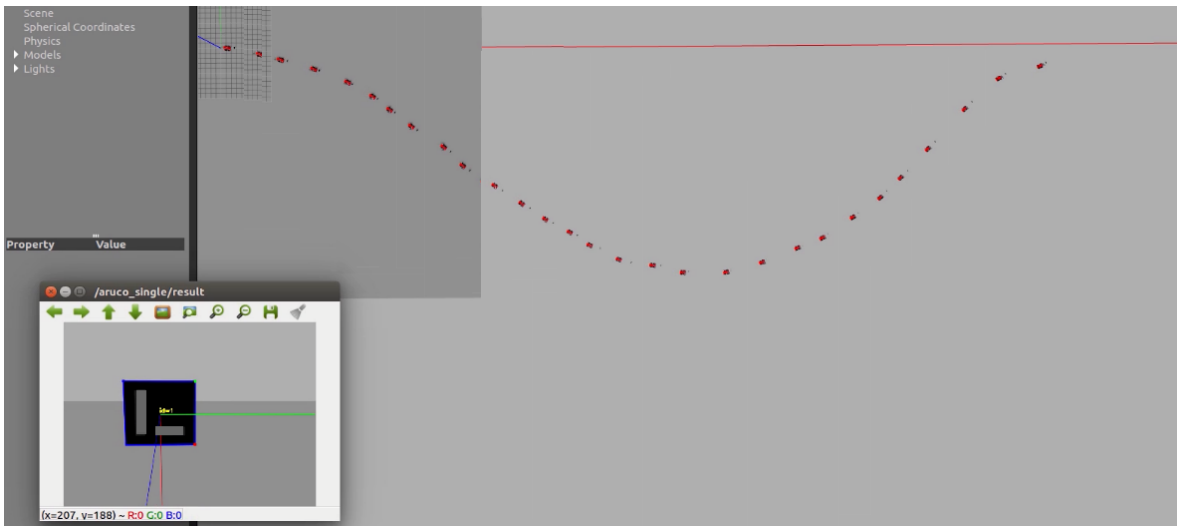

Figura 5.24: Trayectoria sinusoidal en el entorno de Gazebo

5.2.4. Movimiento en forma de espiral

Los experimentos que se han llevado a cabo se recogen en esta tabla:

Número Prueba	k_1	k_2	k_3
Prueba 1	0.2	80.0	80.0
Prueba 2	0.25	70.0	90.0
Prueba 3	0.25	90.0	70.0
Prueba 4	0.25	90.0	100.0
Prueba 5	0.25	50.0	90.0

Tabla 5.5: Parámetros movimiento espiral

De las cinco pruebas que se han realizado, se ha elegido la prueba 2 y por lo tanto los valores de las ganancias k_1, k_2 y k_3 son 0.25,70.0 y 90.0 respectivamente. A continuación, se mostrarán las gráficas más relevantes al igual que se ha hecho en las trayectorias anteriores.

La figura 5.25 representa la evolución del eje x e y. De esta gráfica se puede observar que el robot sigue a la marca y ambos presentan una trayectoria sinusoidal.

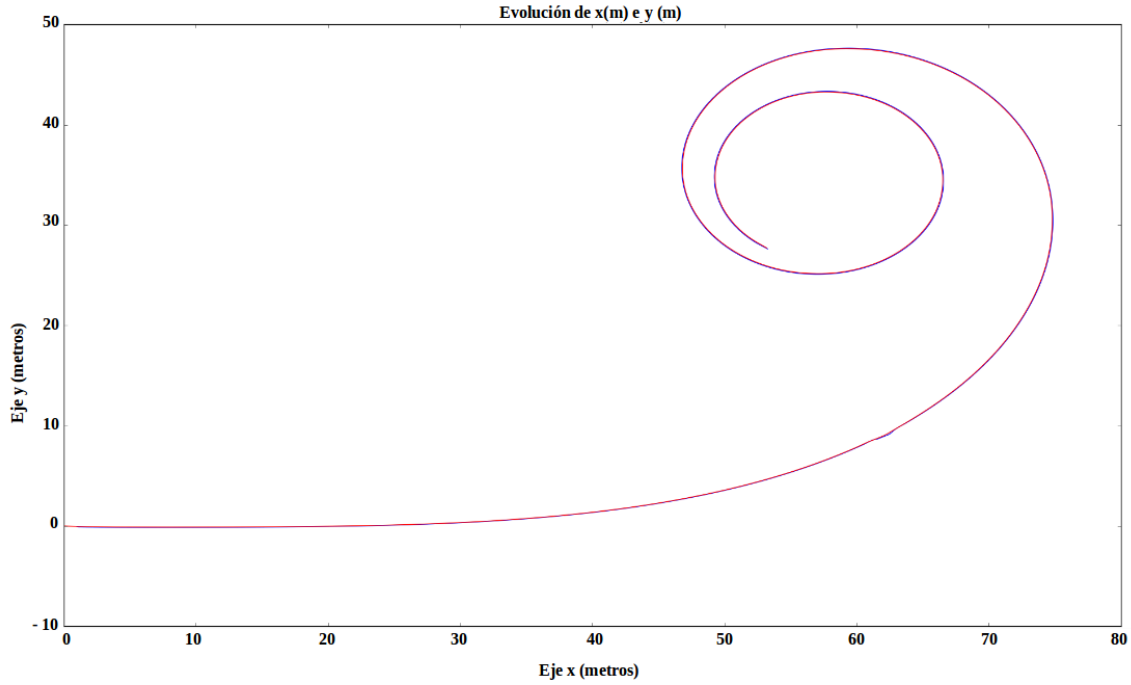

Figura 5.25: Evolución x e y para una trayectoria con forma de espiral

Las figuras 5.26 y 5.27 representan los errores en el eje x y en el eje y. De ambas gráficas podemos deducir que el controlador funciona adecuadamente ya que estos errores se mantienen constantes.

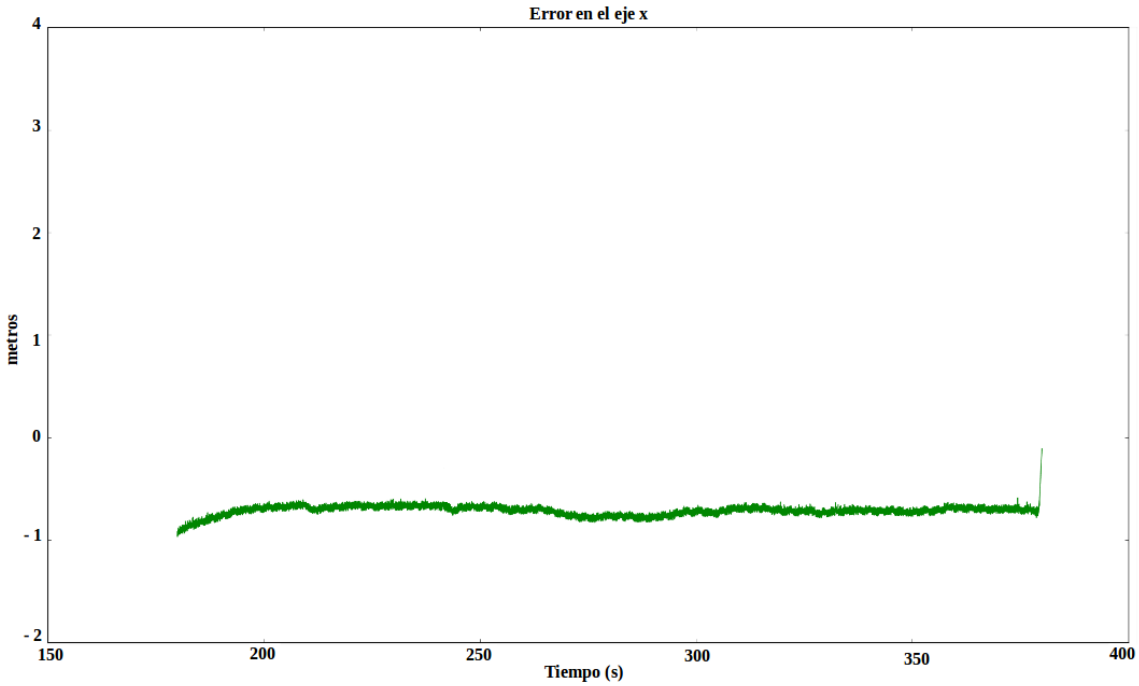

Figura 5.26: Error en el eje x para una trayectoria con forma de espiral

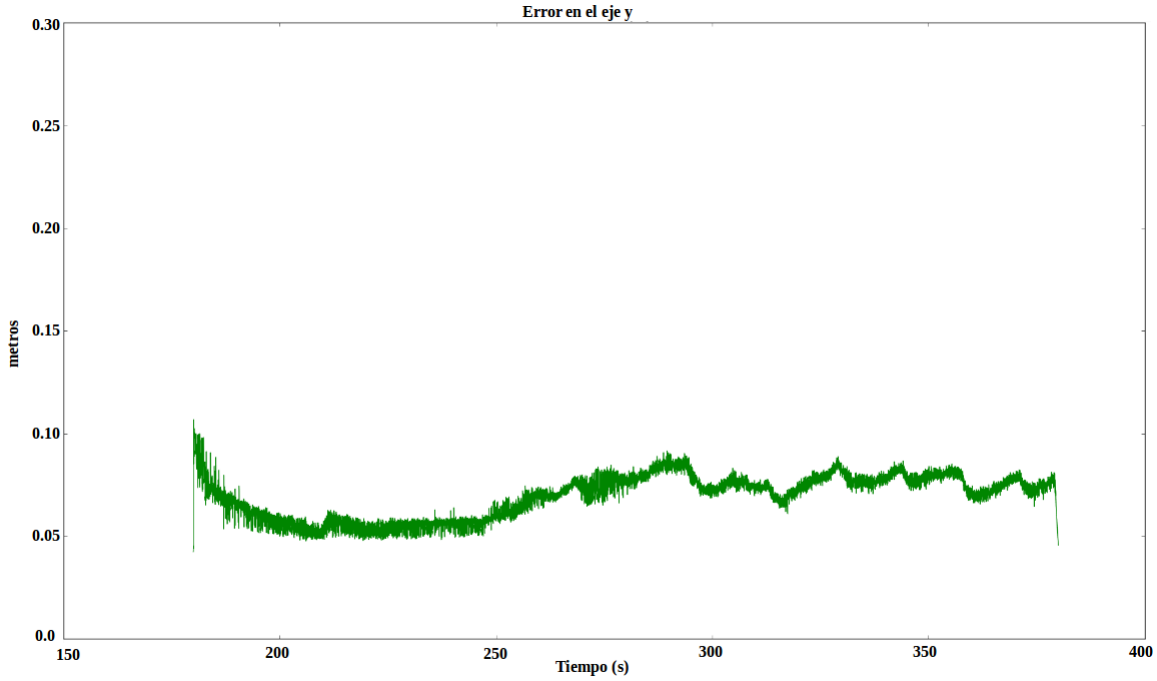

Figura 5.27: Error en el eje y para una trayectoria con forma de espiral

Finalmente se muestra una figura (figura 5.28), la cual recoge una sucesión de imágenes tomadas en Gazebo y que muestran el seguimiento de la trayectoria. En ella podemos apreciar como el robot sigue a la marca describiendo una trayectoria en forma de espiral.

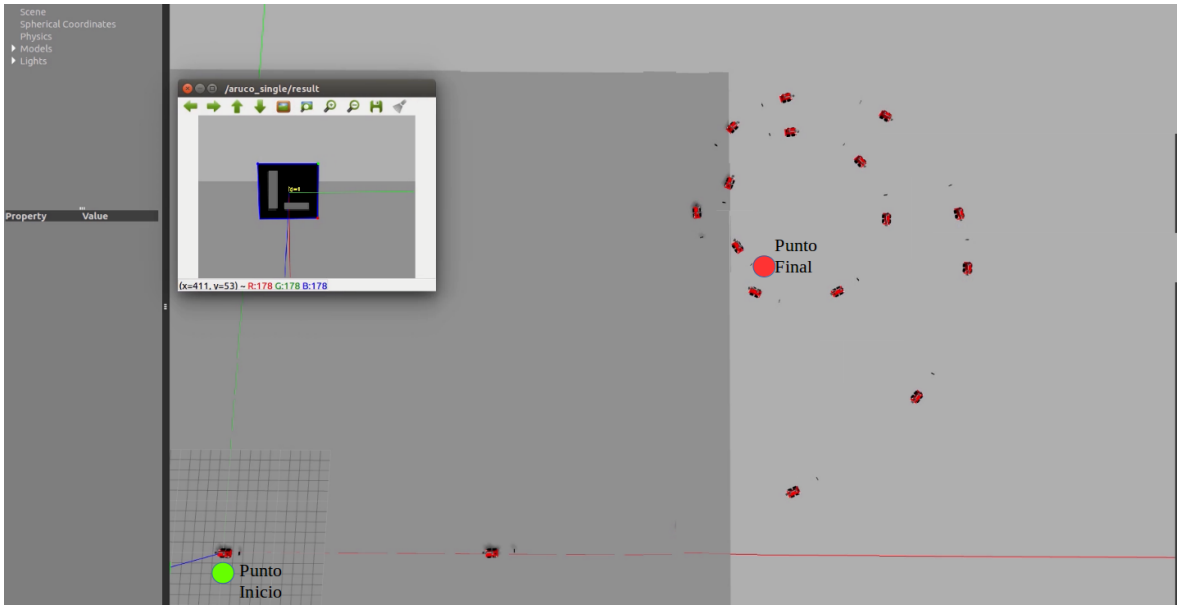

Figura 5.28: Trayectoria en forma de espiral en Gazebo

5.2.5. Movimiento en forma de ocho

Los experimentos que se han llevado a cabo se recogen en esta tabla:

Número Prueba	k_1	k_2	k_3
Prueba 1	0.2	80.0	90.0
Prueba 2	0.3	100.0	110.0
Prueba 3	0.25	110.0	120.0
Prueba 4	0.22	140.0	150.0

Tabla 5.6: Parámetros movimiento en forma de ocho

De todos los experimentos anteriores se ha elegido la prueba 1. Por lo tanto, los valores de las ganancias k_1, k_2 y k_3 serán 0.2, 80.0 y 90.0 respectivamente.

A continuación, se muestran las gráficas de la evolución de x respecto de y (Figura 5.29), el error en x (Figura 5.30) y el error en y (Figura 5.31). En las tres gráficas se comprueba el correcto funcionamiento del regulador.

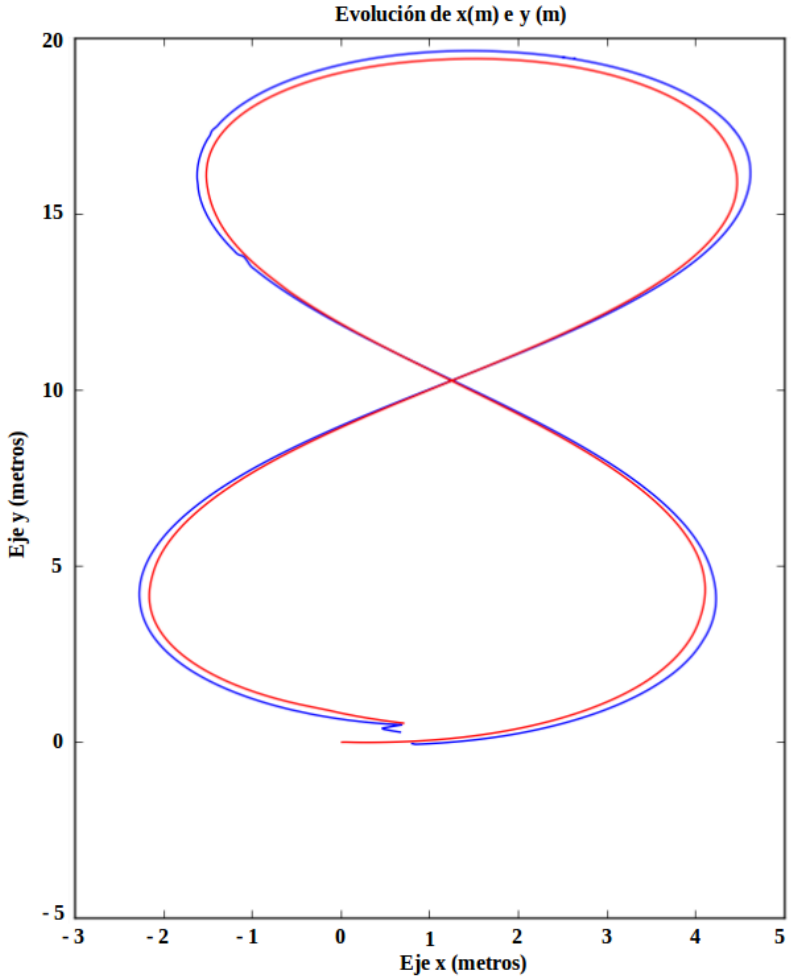

Figura 5.29: Evolución x e y para una trayectoria en forma de ocho

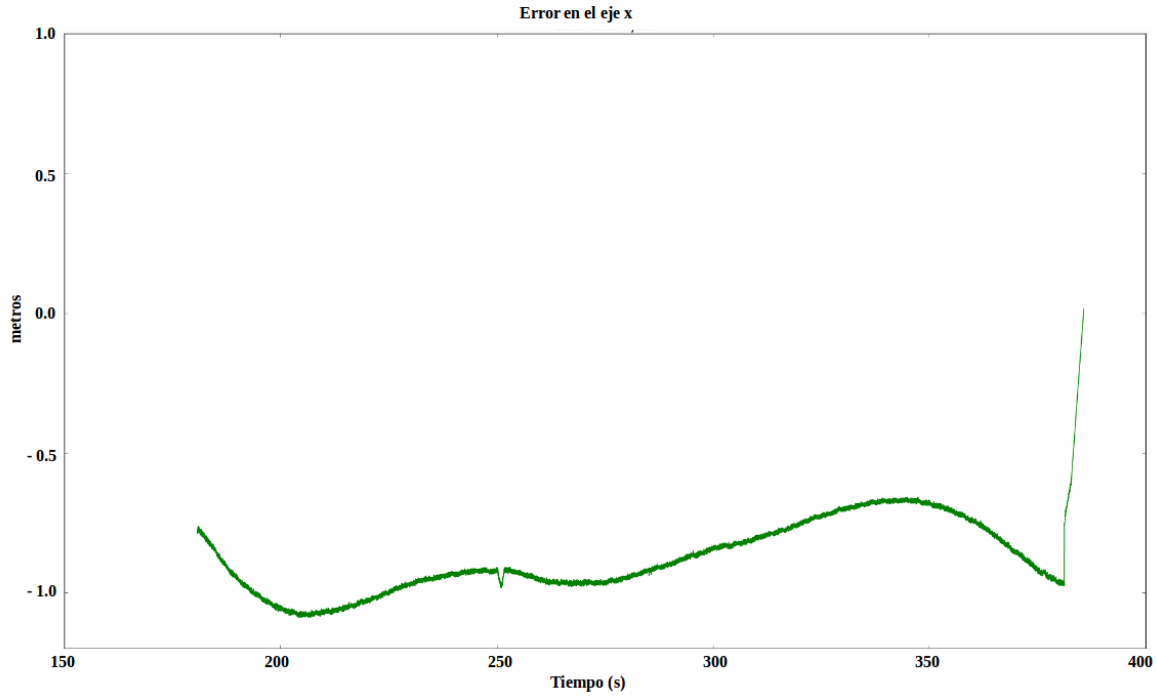

Figura 5.30: Error en el eje x para una trayectoria en forma de ocho

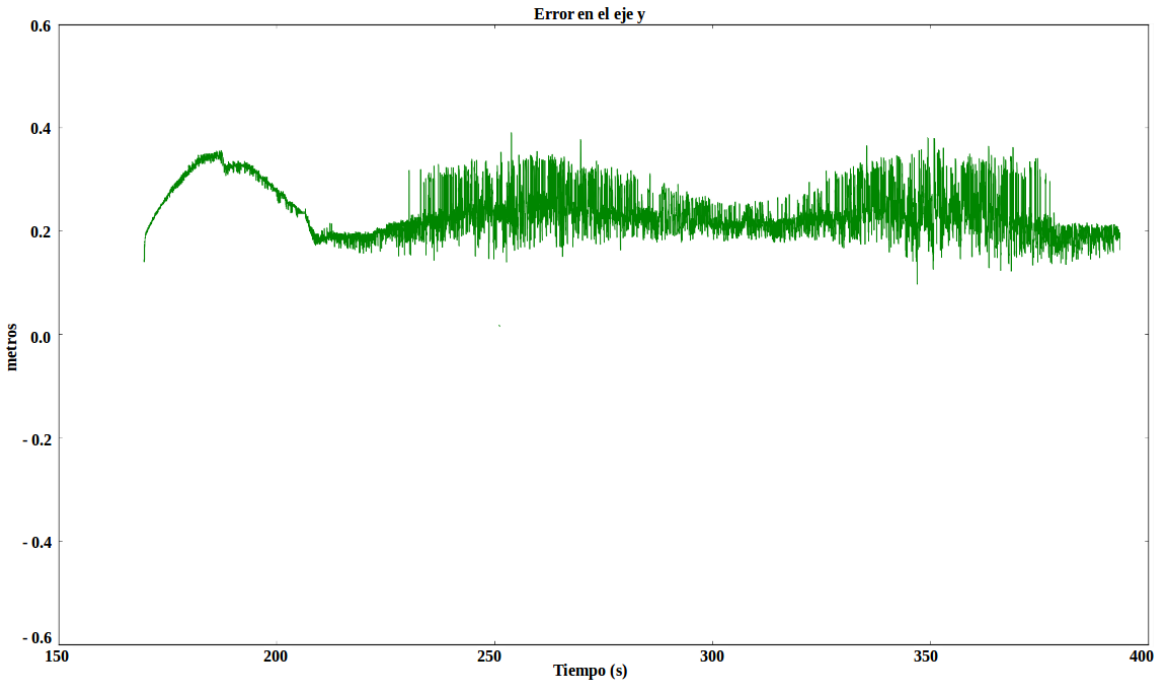

Figura 5.31: Error en el eje y para una trayectoria en forma de ocho

Tanto en la gráfica del error en x (Figura 5.30) como en la del error en y (Figura 5.31) se ve como estas se mantienen constantes. En el primer caso, el valor oscilará entre -0.5 y -1.0 metro. Mientras que, para el segundo caso el valor se mantendrá en torno a 0.2 metros.

Finalmente, al igual que en el resto de las trayectorias se muestra la figura 5.32 que es una sucesión de fotogramas tomados en Gazebo y que muestra el recorrido del robot en su seguimiento a la marca ArUco.

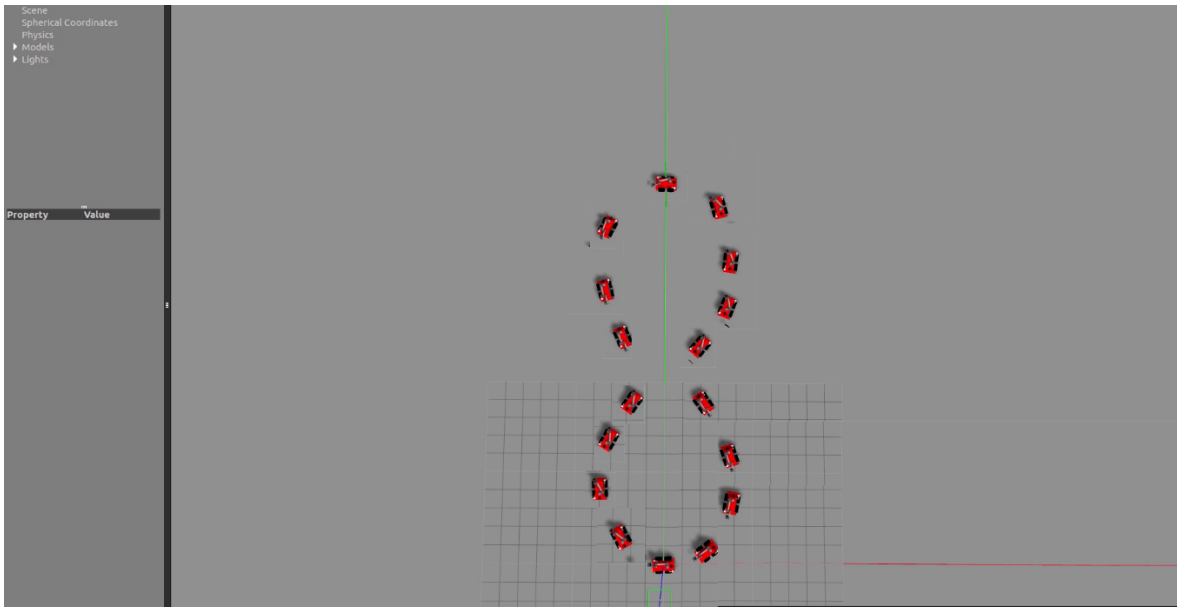

Figura 5.32: Trayectoria en forma de ocho en Gazebo

5.3. Planteamiento de experimentos reales

En este apartado se explicarán los pasos y cambios que se realizarán para poder llevar a cabo las simulaciones mostradas anteriormente, a una prueba real con el robot campero.

En los apartados anteriores, las simulaciones han sido realizadas suponiendo que la velocidad lineal y angular que presenta la marca es ideal. Sin embargo, para realizar experimentos reales se necesita que esas velocidades sean obtenidas a través del movimiento que presenta realmente la marca. En definitiva, lo que se necesita es cambiar las velocidades lineales y angulares ideales por las reales y a continuación se muestran dichos cambios.

Por un lado, se necesita calcular la velocidad lineal. Esta presenta la siguiente formula:

$$v\_lineal = \frac{Posición}{Tiempo} \tag{5.1}$$

Por lo tanto, para obtener la velocidad lineal, se necesita conocer la posición y el tiempo. Esto se va a obtener de la siguiente forma:

$$Posición = Posición\_Actual - Posición\_Anterior \tag{5.2}$$

$$Tiempo = Tiempo\_Actual - Tiempo\_Anterior \tag{5.3}$$

Esto comentado anteriormente en el código se escribiría de la siguiente forma:

```
arucoNav.longitudX = len (arucoNav.mark_x_list)
arucoNav.longitudY = len (arucoNav.mark_y_list)
arucoNav.longitudT = len (arucoNav.tiempo_list)

x_actual = arucoNav.mark_x_list[arucoNav.longitudX-1]
x_anterior = arucoNav.mark_x_list[arucoNav.longitudX-2]

y_actual = arucoNav.mark_y_list[arucoNav.longitudY-1]
y_anterior = arucoNav.mark_y_list[arucoNav.longitudY-2]

t_actual =arucoNav.tiempo_list[arucoNav.longitudT-1]
t_anterior =arucoNav.tiempo_list[arucoNav.longitudT-2]

v_x = (x_actual-x_anterior)/(t_actual-t_anterior)
v_y = (y_actual-y_anterior)/(t_actual-t_anterior)

v_x_2 = v_x*v_x
v_y_2 = v_y*v_y
numero = v_x_2 + v_y_2
v_t = math.sqrt(numero)

vMarca= v_t
```

Por otro lado, se necesita calcular la velocidad angular. Esta velocidad presenta la siguiente formula:

$$v\_angular = w = \frac{Orientación}{Tiempo} \tag{5.4}$$

Por lo tanto, para obtener la velocidad angular, se necesita conocer la orientación y el tiempo. Esto se va a obtener de la siguiente forma:

$$Orientación = Orientación\_Actual - Orientación\_Anterior \tag{5.5}$$

$$Tiempo = Tiempo\_Actual - Tiempo\_Anterior \tag{5.6}$$

Esto comentado anteriormente en el código se escribiría de la siguiente forma:

```
arucoNav.longitudTheta = len (arucoNav.mark_theta_list)
arucoNav.longitudT = len (arucoNav.tiempo_list)

theta_actual =arucoNav.mark_theta_list[arucoNav.longitudTheta-1]
```

```
theta_anterior=arucoNav.mark_theta_list[arucoNav.longitudTheta-2]

t_actual =arucoNav.tiempo_list[arucoNav.longitudT-1]
t_anterior =arucoNav.tiempo_list[arucoNav.longitudT-2]

wMarca= (theta_actual-theta_anterior)/(t_actual-t_anterior)
```

Finalmente, si en el fichero ArucoTracking.py explicado anteriormente, se cambian los valores de vMarca y wMarca ideales por estas líneas de código se obtendrán los valores reales del movimiento de la marca. Y con esos valores, se podrá realizar un correcto seguimiento del robot para cualquier trayectoria que describa la marca.

Para finalizar este capítulo 5, se deja un enlace en el cual se puede encontrar material multimedia adicional. `https://drive.google.com/drive/folders/1OyLXsQkyvg6xdAc6WowUPfvV-JqptdJs?usp=sharing`

Capítulo 6

Conclusiones y trabajo futuro

En este capítulo, se recogen las conclusiones a las que se han llegado con este trabajo y se menciona una posible continuación del mismo.

6.0.1. Conclusiones

Una vez terminado el trabajo se puede decir que se han cumplido de una forma satisfactoria los objetivos que se plantearon inicialmente. Se ha conseguido obtener un algoritmo, el cual permite que el robot pueda seguir perfectamente a una marca ArUco. Para conseguir dicho objetivo principal nos hemos tenido que familiarizar con la distribución de Linux ubuntu 16.04, aprender a utilizar el entorno de ROS y sus herramientas (RViz y Gazebo), crear marcas ArUco con movimiento en Gazebo y desarrollar código en Matlab y Python para el seguimiento. Con todo esto realizado se han obtenido las siguientes conclusiones:

- La filosofía de ROS funciona muy bien, ya que gracias a que en la red hay una gran cantidad de código abierto se han podido ir solucionando los errores con los que nos íbamos encontrando en el trabajo.

- Las simulaciones y la visualización de los resultados en distintas herramientas, como en este caso ha sido Gazebo, es de gran ayuda para poder ajustar los parámetros correctamente y saber si en la realidad funcionará el código. Esto evita que se puedan producir daños en el modelo real.

- El seguimiento de unas marcas mediante un robot manipulador, presenta una gran ventaja. Gracias a esto el robot puede trabajar de manera autónoma transportando cargas de un lugar a otro en una empresa.

- La metodología utilizada y los conceptos aprendidos en este trabajo pueden ser utilizados para otros modelos de robots que no sean el robot campero.

6.0.2. Trabajo futuro

Después de realizar el trabajo y obtener las conclusiones se puede decir que los robots móviles manipuladores tienen una gran cantidad de aplicaciones y que éstas son necesarias para favorecer las condiciones de los trabajadores en las empresas.

En definitiva, se deben seguir desarrollando nuevos algoritmos. Por lo tanto, para futuros trabajos se podría pensar en la idea de que el robot en lugar de seguir a un objeto pudiese seguir a una persona, y para ello sería necesario aplicar los conceptos de visión por computación.

Capítulo 7

Bibliografía

[1] Robotnik. https://robotnik.eu/es/productos/manipuladores-moviles/rb-eken/, 2021.

[2] David Barrera Gracia. Navegación autónoma de robot manipulador móvil con cámara y láser en el entorno ros. *Trabajo de Fin de Grado, Universidad de Zaragoza*, 2020.

[3] Commandia. http://commandia.unizar.es/es/lo-basico-de-commandia/, 2021.

[4] Rafael Muñoz-Salinas, Sergio Garrido-Jurado, Francisco Madrid-Cuevas y Rafael Medina-Carnicer. https://www.uco.es/investiga/grupos/ava/node/26, 2021.

[5] Sergio Garrido-Jurado; Rafael Muñoz-Salinas; Francisco Madrid-Cuevas and Rafael Medina-Carnicer. Generation of fiducial marker dictionaries using mixed integer linear programming. *Pattern Recognition:51,481-491*, 2016.

[6] Francisco Romero-Ramirez; Rafael Muñoz-Salinas and Rafael Medina-Carnicer. Speeded up detection of squared fiducial markers. *Image and Vision Computing,76, 06*, 2018.

[7] Wikipedia. https://es.wikipedia.org/wiki/Licencia_BSD, 2021.

[8] ROS. http://wiki.ros.org/, 2021.

[9] Gazebo. http://gazebosim.org/, 2021.

[10] RViz. http://wiki.ros.org/rviz, 2021.

[11] Matlab. https://es.mathworks.com/, 2021.

[12] Oussama Khatib Bruno Siciliano. *SPRINGER HANDBOOK OF ROBOTICS*. SPRINGER-VERLAG, 2008.

[13] Wikipedia. https://es.wikipedia.org/wiki/Teoria_de_la_estabilidad.

[14] Terminator. https://terminator-gtk3.readthedocs.io/en/latest/, 2021.

[15] Animated box. http://gazebosim.org/tutorials?tut=animated_box, Version 8.0.

Lista de Figuras

Lista de Tablas

Anexos

Anexos A

Como crear un escenario con marcas ArUco móviles

En este anexo se explica cómo crear un escenario con marcas ArUco móviles. Para ello se parte de que dichas marcas ya han sido creadas en Blender siguiendo los pasos necesarios, los cuales están indicados en el trabajo de Navegación autónoma de robot manipulador móvil con cámara y láser en el entorno ROS [2], y por lo tanto esto no se explica en este anexo. A continuación, se explica el proceso que se ha seguido para crear dicho escenario en Gazebo y después los pasos a seguir para darle movimiento a la marca.

A.1. Creación del escenario en Gazebo

Primero se crea una carpeta que contenga la imagen de la marca (aruco_mark_1.png), el modelo 3D de la marca creado en Blender (aruco_visual_marker_1.dae) y el material (aruco_visual_marker_1_marker.material). Esta carpeta será creada en el siguiente directorio: campero_ws/src/campero.

Segundo se debe modificar el archivo que contiene el modelo 3D de la marca (aruco_visual_marker_1.dae). Lo abrimos con Gedit y buscamos la línea 44, es decir, donde aparece la imagen que usamos para crear la marca. En dicha línea solo aparecerá el nombre del archivo y se debe poner en esa línea la dirección completa donde se encuentra. Esto se puede ver a continuación, en la figura A.1

```
<image id="aruco_mark_1_png" name="aruco_mark_1_png">
  <init_from>home/alba/campero_ws_Alba/src/campero/Marca1_14</init_from>
</image>
```

Figura A.1: Archivo que contiene el modelo 3D de la marca

Una vez realizados estos pasos, abrimos Gazebo para crear nuestro escenario. Para ello escribimos $Gazebo y se abrirá un entorno vacío como se muestra en la figura A.2

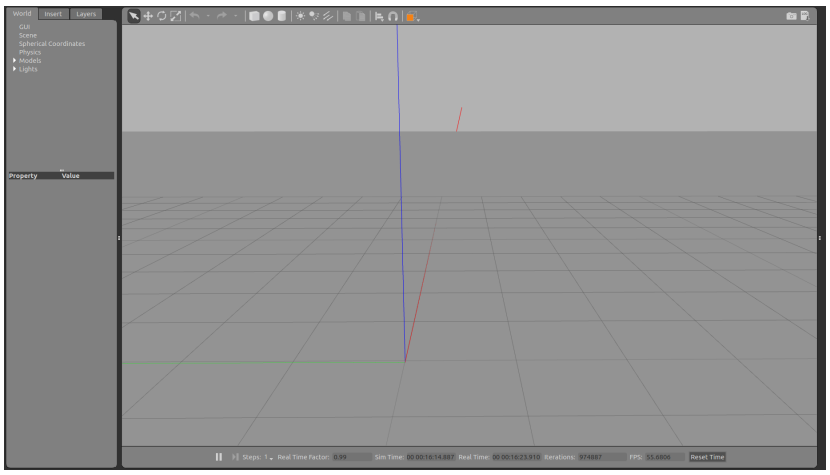

Figura A.2: Entorno de Gazebo vacío

En el entorno vacío de Gazebo crearemos un objeto, en este caso un cubo, el cual hará de marca. Una vez creado el cubo lo seleccionamos y con el botón derecho del ratón nos aparece un desplegable. En dicho desplegable se elige “Edit Model”. Esto se recoge en la figura A.3

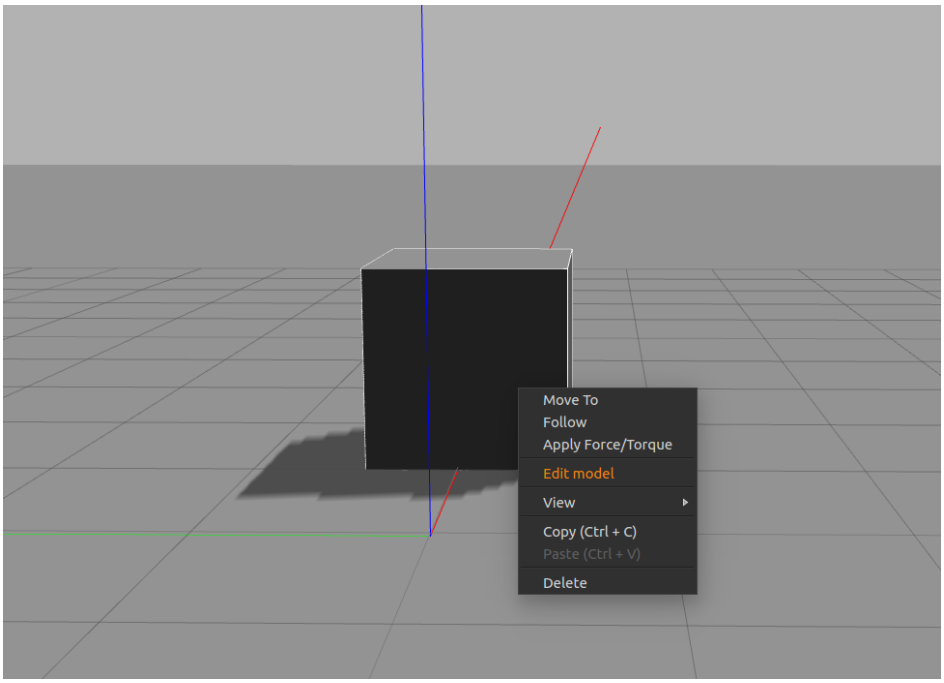

Figura A.3: Entorno de Gazebo con la caja

Se nos abrirá el modelo de edición. Una vez dentro volvemos a seleccionar el cubo y en el nuevo desplegable elegimos la opción "Open Link Inspector". Entonces, se abre una ventana con tres pestañas. Elegimos la pestaña "Visual", y dentro de esta el apartado "Geometry", donde seleccionaremos mesh y nos dará la opción de seleccionar nuestro archivo con extensión .dae (aruco_visual_marker_1.dae). Después pulsamos ok. Esto mencionado se recoge en la figura A.4

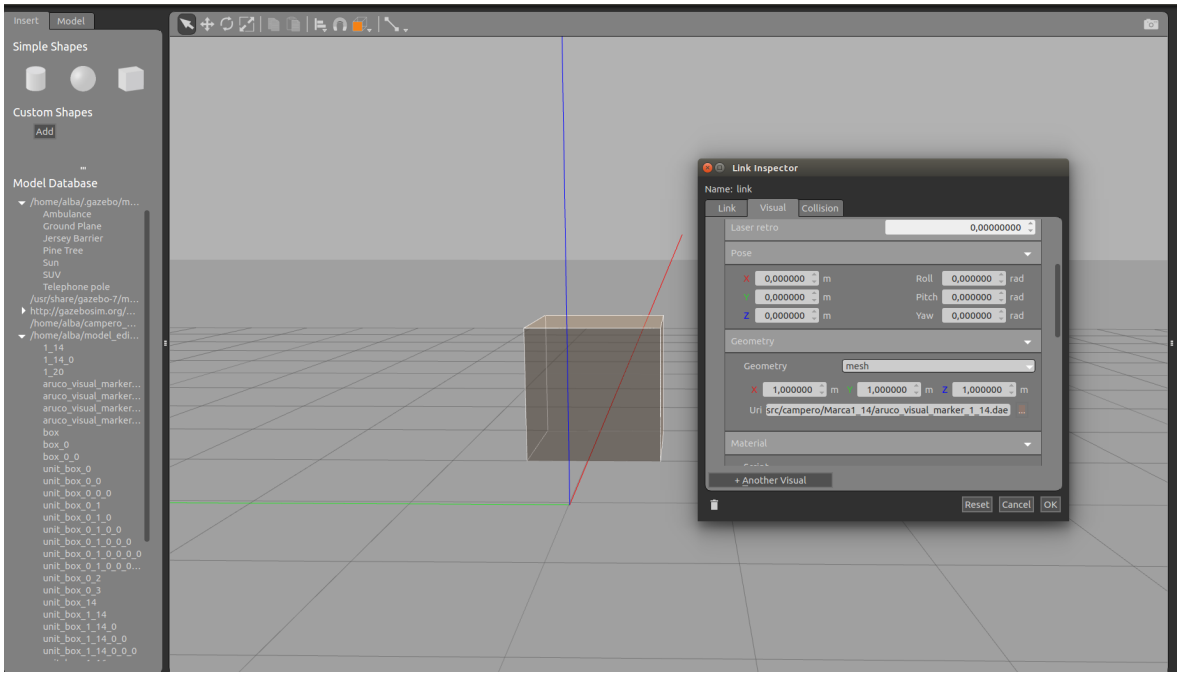

Figura A.4: Modelo de edición en el entorno de Gazebo

A continuación, se guarda el objeto que se ha creado para ello le damos a "File". Una vez guardado para salir del editor le volvemos a dar a "File", y "Exit Model Editor". Una vez hecho esto, se ve como se tiene la forma de la marca, pero no el material. Lo que se va a hacer es guardar el entorno que hemos creado, con el nombre en mi caso de "marca.world", y cerrar Gazebo.

Para poder visualizar la marca deberemos modificar el fichero del mundo (marca.world) que acabamos de crear y para ello lo abrimos con Gedit. Una vez dentro, se busca en el apartado $< visual >$, dentro de este el apartado $< material >$. Ahí tenemos que modificar las dos líneas, la que pone $< name >$ y $< uri >$. En la primera tenemos que poner el nombre que tiene nuestro material y en la segunda tenemos que poner la dirección donde se encuentra el archivo del material. Esto se muestra en la siguiente figura A.5

```
</geometry>
<material>
  <script>
    <name>ArucoVisualMarker1_20/Marker</name>
    <uri>/home/alba/campero_ws_Alba/src/campero/Marca1_20/aruco_visual_marker_1_20_marker.material</uri>
  </script>
  <ambient>0.3 0.3 0.3 1</ambient>
  <diffuse>0.7 0.7 0.7 1</diffuse>
  <specular>0.01 0.01 0.01 1</specular>
  <emissive>0 0 0 1</emissive>
  <shader type='vertex'>
    <normal_map>__default__</normal_map>
  </shader>
</material>
```

Figura A.5: Fichero marca.world

Finalmente, al abrir de nuevo el mundo (marca.world) con Gazebo se puede ver correctamente la marca creada (figura A.6). Una vez tenemos esto se debe dar movimiento a dicha marca y es lo que se pasa a explicar a continuación.

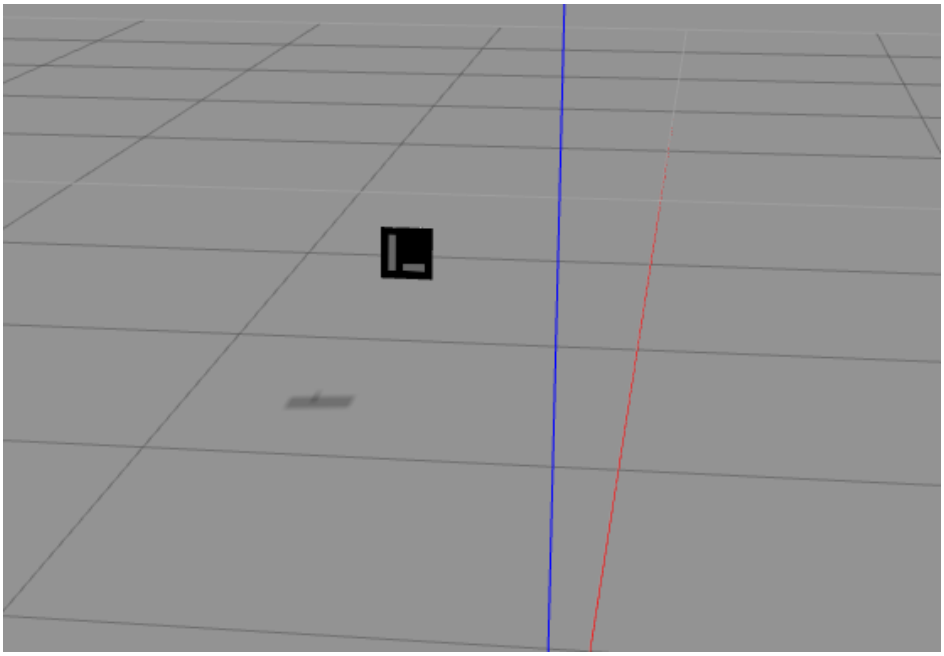

Figura A.6: Escenario con la marca ArUco

A.2. Plugin para el movimiento de la marca

Tanto los ficheros como los pasos a seguir para obtener el plugin han sido obtenido de un tutorial de Gazebo [15]. A continuación, se explica de una manera un poco más detallada los pasos a seguir.

Para obtener el plugin, el cual nos dará el movimiento de la marca, lo primero que se hará será crear una carpeta donde se recogerá el fichero del espacio, en mi caso marca.world. En esta carpeta también se incluyen 4 ficheros más, que a continuación se explican.

1. **animated_box.cc**: Fichero que define el movimiento del objeto que queremos definir, en este caso la marca ArUco.

2. **independent_listener.cc**: Ejecutable que recibe la información de la posición y nos devuelve dicho valor por pantalla.

3. **integrated_main.cc**: Ejecutable que crea una simulación, recibe información de la posición e imprime dicho valor.

4. **CMakeLists.txt**:Script de compilación de CMake.

De los cuatro ficheros mencionados, el que se va a modificar será animated_box.cc. En él se darán los valores de la posición y orientación de la marca. Esto se muestra en la figura A.7 .

```
gazebo::common::PoseKeyFrame *key;

// set starting location of the box

key = anim->CreateKeyFrame(0);
key->Translation(ignition::math::Vector3d(1.0, 0.0, -1.5));
key->Rotation(ignition::math::Quaterniond(0, 0, 0));

key = anim->CreateKeyFrame(1);
key->Translation(ignition::math::Vector3d(1.0, 0.0, -1.5));
key->Rotation(ignition::math::Quaterniond(0, 0, 0));

key = anim->CreateKeyFrame(2);
key->Translation(ignition::math::Vector3d(1.0, 0.0, -1.5));
key->Rotation(ignition::math::Quaterniond(0, 0, 0));

key = anim->CreateKeyFrame(3);
key->Translation(ignition::math::Vector3d(1.0, 0.0, -1.5));
key->Rotation(ignition::math::Quaterniond(0, 0, 0));

key = anim->CreateKeyFrame(4);
key->Translation(ignition::math::Vector3d(1.0, 0.0, -1.5));
key->Rotation(ignition::math::Quaterniond(0, 0, 0));

key = anim->CreateKeyFrame(5);
key->Translation(ignition::math::Vector3d(1.0, 0.0, -1.5));
key->Rotation(ignition::math::Quaterniond(0, 0, 0));

key = anim->CreateKeyFrame(6);
key->Translation(ignition::math::Vector3d(1.0, 0.0, -1.5));
key->Rotation(ignition::math::Quaterniond(0, 0, 0));
```

Figura A.7: Fichero animated_box.cc

Una vez que se tiene la carpeta con los ficheros necesarios, los pasos que se van a dar son: abrimos una terminal nueva y se ejecutan los siguientes comandos:

$ cd campero_ws
$ cd src/campero/campero_sim/campero_gazebo/marca
$ mkdir build
$ cd build
$ cmake ../
$ make
$ export GAZEBOPLUGINPATH=‘pwd‘:$GAZEBO_PLUGIN_PATH

Anexos B

Código utilizado en Matlab

El código que se muestra en este anexo ha sido utilizado en el entorno de Matlab para llevar a cabo la simulación del seguimiento del robot a una marca ArUco.

```
clear all
close all
%-------------------------INICIALIZACION PARAMETROS-------------------------%
% Vector de tiempos
Tmax= 2000;
delta = 0.1;
tiempo=(0:delta:Tmax)';
t=length(tiempo);

% Inicializacion de la posicion de la marca
xMarca = 0;
yMarca = 0;
titaMarca = 0;

% Inicializacion de la posicion del robot
xRobot=-1;
yRobot=-0.5;
titaRobot=0;

% Vectores para guardar datos
xMarcaIni=zeros(t,1);
yMarcaIni=zeros(t,1);
titaMarcaIni=zeros(t,1);

xRobotIni=zeros(t,1);
yRobotIni=zeros(t,1);
titaRobotIni=zeros(t,1);

v_xe = zeros(t,1);
v_ye =zeros (t,1);
v_titae = zeros (t,1);

xR=zeros(t,1);
yR=zeros(t,1);
titaR=zeros(t,1);

xM=zeros(t,1);
yM=zeros(t,1);
titaM=zeros(t,1);
```

```
40
41  v_v=zeros(t,1);
42  v_w=zeros(t,1);
43
44  v_vMarca=zeros(t,1);
45  v_wMarca=zeros(t,1);
46
47  for it=1:t
48      %-------------------------------TRAYECTORIA MARCA-------------------------------%
49      xMarcaIni(it,:)=xMarca;
50      yMarcaIni(it,:)=yMarca;
51      titaMarcaIni(it,:)=titaMarca;
52
53      vMarca=0.2;
54      wMarca=0.0;       % Movimiento Lineal (0.0)   %Movimiento Circular (1.8)
55
56      %-----------------------------------CONTROL-------------------------------------%
57
58      % Localizacion Actual del robot
59      xRobotIni(it,:)=xRobot;
60      yRobotIni(it,:)=yRobot;
61      titaRobotIni(it,:)=titaRobot;
62
63      % Calculo del error en coordenadas cartesianas
64      xe = (xRobot-xMarca)*cosd(titaMarca)+(yRobot-yMarca)*sind(titaMarca);
65      ye = -(xRobot-xMarca)*sind(titaMarca)+(yRobot-yMarca)*cosd(titaMarca);
66      titae = titaRobot-titaMarca;
67
68      % Calculo de la posicion de la marca
69      titaMarca=titaMarca + wMarca*delta;
70      DyMarca= zeros(1,1);
71      DxMarca= vMarca*delta;
72
73      if (wMarca ~= 0)
74          L_Marca= vMarca/(wMarca*pi/180); %Curvature radius (radians)
75          DyMarca= L_Marca -L_Marca.*cosd(wMarca*delta);
76          DxMarca= L_Marca.*sind(wMarca*delta);
77      end
78
79      xMarca= xMarca +DxMarca .*cosd(titaMarca) -DyMarca .*sind(titaMarca);
80      yMarca= yMarca +DxMarca .*sind(titaMarca) +DyMarca .*cosd(titaMarca);
81
82      % Ley de control
83      k1=0.1;
84      k2=0.1;
85      k3=1;
86
87      Z1=xe;
88      Z2=ye;
89      Z3=tand(titae);
90
91      v=vMarca-(k1*abs(vMarca)*Z1);
92      w=wMarca-(k2*vMarca*Z2)-(k3*abs(vMarca)*Z3);
93
94      %Calculo de la posicion del robot
95      titaRobot=titaRobot + w*delta;
96      Dy= zeros(1,1);
97      Dx= v*delta;
```

```
98
99
100
101         if (w ~= 0)
102             L= v./(w *pi/180);
103             Dy= L -L.*cosd(w*delta);
104             Dx= L.*sind(w*delta);
105         end
106
107         xRobot= xRobot +Dx .*cosd(titaRobot) -Dy .*sind(titaRobot);
108         yRobot= yRobot +Dx .*sind(titaRobot) +Dy .*cosd(titaRobot);
109
110         %Vector que almacena los datos del bucle
111         xR(it,:)=xRobot;
112         yR(it,:)=yRobot;
113         titaR(it,:)=titaRobot;
114
115         xM(it,:)=xMarca;
116         yM(it,:)=yMarca;
117         titaM(it,:)=titaMarca;
118
119         v_v(it,:)=v;
120         v_w(it,:)=w;
121
122         v_vMarca(it,:)=vMarca;
123         v_wMarca(it,:)=wMarca;
124
125         v_xe(it,:)=xe;
126         v_ye(it,:)=ye;
127         v_titae(it,:)=titae;
128
129 end
130
131 %-------------------------REPRESENTACION GRAFICA -------------------------%
132 %Posicion tanto de la marca como del robot en el eje x
133 subplot(3,3,1)
134 plot(tiempo,xM,'-.b','LineWidth',4)
135 hold on
136 plot(tiempo,xR,'r','LineWidth',2)
137 xlabel ('tiempo (s)')
138 ylabel ('metros')
139 title ('Posicion en el eje x(m)')
140 legend ('Marca','Robot')
141
142 %Posicion tanto de la marca como del robot en el eje y
143 subplot(3,3,2)
144 plot(tiempo,yM,'-.b','LineWidth',4)
145 hold on
146 plot(tiempo,yR, 'r','LineWidth',2)
147 xlabel ('tiempo (s)')
148 ylabel ('metros')
149 title ('Posicion en el eje y')
150 legend ('Marca','Robot')
151
152 %Posicion tanto de la marca como del robot del angulo tita
153 subplot(3,3,3)
154 plot(tiempo,titaM,'-.b','LineWidth',4)
155 hold on
```

```
156 plot(tiempo,titaR, 'r','LineWidth',2)
157 xlabel ('tiempo (s)')
158 ylabel ('grados')
159 title ('Posicion de angulo theta')
160 legend ('Marca','Robot')
161
162 %Velocidad Lineal de la marca y del robot
163 subplot(3,3,4)
164 plot (tiempo,v_vMarca,'-.b','LineWidth',4)
165 hold on
166 plot(tiempo,v_v,'r','LineWidth',2)
167 xlabel ('tiempo (s)')
168 title ('Velocidad Lineal')
169 legend ('marca','robot')
170
171 %Velocidad Angular de la marca y del robot
172 subplot(3,3,5)
173 plot(tiempo,v_wMarca,'-.b','LineWidth',4)
174 hold on
175 plot (tiempo,v_w,'r','LineWidth',2)
176 xlabel ('tiempo (s)')
177 title ('velocidad Angular')
178 legend ('Marca','Robot')
179
180 %Evolucion de los ejes x e y
181 subplot(3,3,6)
182 plot(xM,yM,'-.b','LineWidth',4)
183 hold on
184 plot(xR,yR,'r','LineWidth',2)
185 xlabel ('x (m)')
186 ylabel ('y (m)')
187 title ('x, y(m)')
188 legend ('Marca','Robot')
189
190 %Error en el eje x
191 subplot(3,3,7)
192 plot (tiempo,v_xe,'g','LineWidth',3)
193 xlabel ('tiempo (s)')
194 ylabel ('metros')
195 title ('Error en el eje x')
196
197 %Error en el eje y
198 subplot(3,3,8)
199 plot(tiempo,v_ye,'g','LineWidth',3)
200 xlabel ('tiempo (s)')
201 ylabel ('metros')
202 title('Error en el eje y')
203
204 %Error en el angulo tita
205 subplot(3,3,9)
206 plot(tiempo,v_titae,'g','LineWidth',3)
207 xlabel ('tiempo (s)')
208 ylabel ('grados')
209 title('Error en el angulo theta')
```

Este código anterior sirve para representar tanto una trayectoria lineal como una trayectoria circular. Sin embargo, para representar el resto de trayectorias que se han expuesto en el capítulo 5, lo que se debe hacer es modificar de este código genérico las líneas 53 y 54. Los cambios que hay que realizar se muestran a continuación para cada una de las trayectorias.

Para la **trayectoria sinusoidal**:

```
a = 0.3/100;
v_vt=1+0.1*(1-cosd(2*180*tiempo*a));
vMarca=v_vt(it,1);
v_wt =(180*(pi/180)*a)^2*(-cosd(1*180*tiempo*a))*20;
wMarca =v_wt(it,1);
```

Para la **trayectoria en forma de espiral**:

```
vMarca=1.0;
a=0.3;
b=180/Tmax;
v_wt=a*(1+cosd(b*tiempo-180));
wMarca = v_wt(it,1);
```

Para la **trayectoria en forma de ocho**:

```
Tmax= 200+1;%+1 and later -1 to keep size with diff
delta=0.1;
t=(0:delta:Tmax)';
nt = length(t);

dt=t*2*180/Tmax;

scale = 40 * ( 2 ./ (3 - cosd(2*dt)) );
x = scale .* cosd(dt);
y = scale .* sind(2*dt) / 2;

dx=diff(x./delta);
dx=[dx; dx(end,:)];
dy=diff(y./delta);
dy=[dy; dy(end,:)];
fi=(atan2(dy,dx))*(180/pi);
fi=unwrap(fi);

v_wt=diff(fi./delta);
v_wt(end,:)=v_wt(end-1,:);
v_wt=[v_wt; v_wt(end,:)];

v_vt=sqrt(dx.^2 + dy.^2);

Tmax=Tmax-1;
t=(0:delta:Tmax)';
vMarca=v_vt(1:length(t));
wMarca=v_wt(1:length(t));
vMarca = v_vt(it,1);
wMarca = v_wt(it,1);
```

Anexos C

Código utilizado en ROS

El código que se muestra en este anexo ha sido utilizado en el entorno de ROS para llevar a cabo la simulación del control de seguimiento.

```
#!/usr/bin/env python

import rospy
import math
import time
import numpy
import tf
import matplotlib.pyplot as plt

from geometry_msgs.msg import Quaternion, PoseWithCovarianceStamped, PoseStamped
from tf.transformations import quaternion_from_euler
from geometry_msgs.msg import Twist
from nav_msgs.msg import Odometry

class arucoTracking():
        def __init__(self):
                # Para la velocidad del robot
                self.vel = Twist()
                self.vel.linear.x=0.0
                self.vel.linear.y=0.0
                self.vel.linear.z=0.0

                self.vel.angular.x=0.0
                self.vel.angular.y=0.0
                self.vel.angular.z=0.0

                # Para la posicion de la marca
                self.mark_x=0.0
                self.mark_y=0.0
                self.mark_z=0.0

                self.mark_alpha=0.0
                self.mark_beta=0.0
                self.mark_theta=0.0

                self.mark_theta_list=[]
                self.mark_theta_median=0.0
```

```
#Para la posicion del robot
self.robot_x=0.0
self.robot_y=0.0
self.robot_z=0.0

self.robot_alpha=0.0
self.robot_beta=0.0
self.robot_theta=0.0

self.robot_theta_list=[]
self.robot_theta_median=0.0

#Vectores para almacenar datos
self.mark_x_list=[]
self.robot_x_list=[]
self.mark_y_list=[]
self.robot_y_list=[]
self.marca_x_anterior=[]
self.tiempo_list=[]
self.error_x_list=[]
self.error_y_list=[]
self.error_theta_list=[]
self.xM=[]
self.xR=[]
self.yM=[]
self.yR=[]
self.titaM=[]
self.titaR=[]
self.errorX=[]
self.errorY=[]
self.errorTita=[]
self.longTiempo=[]
self.longTiempo1=[]
self.t=[]
self.t1=[]
self.list_a=[]
self.list_b=[]
self.list_c=[]
self.velocidad_linear_list=[]
self.velocidad_angular_list=[]
self.velocidad_linear=[]
self.velocidad_angular=[]
self.dif_x_list=[]
self.dif_y_list=[]
self.cos_tita_list=[]
self.sen_tita_list=[]
self.diferencia_eje_x=[]
self.diferencia_eje_y=[]
self.coseno_tita=[]
self.seno_tita=[]
self.tiempo1_list=[]

#Inicializacion de variables
self.marca_detectada_ini=0
self.marca_detectada=0
self.girocompleto=0
self.avancecompleto=0
self.marca_x_anterior=0
```

```
                self.marca_y_anterior=0
                self.marca_theta_anterior=0
                self.velocidad_linear=0.0
                self.velocidad_angular=0.0
                self.delta=10
                self.orientado=0
                self.obtenerTiempo=0
                self.fase2=0
                self.fase3=0

                # Para saber si el robot ha llegado a la posicion objetivo
                self.fin=0

###-----------------FUNCIONES DE CONVERSION DE UNIDADES -----------------###

def sen(grados):
    return math.sin(math.radians(grados))

def cos(grados):
    return math.cos(math.radians(grados))

def tan(grados):
    return math.tan(math.radians(grados))

def atan(grados):
    return math.atan(math.radians(grados))

###-----------------DETECCION MARCA INICIALMENTE -----------------###
def deteccionMarcaInicio():

   if arucoNav.mark_x==0.0 and arucoNav.mark_y==0.0 and arucoNav.mark_theta==0.0:
                arucoNav.marca_detectada_ini=0
                #print('No detecto inicialmente a la marca')
   else:
                arucoNav.marca_detectada_ini=1
                #print('Marca inicialmente detectada')


###-----------------DETECCION MARCA -----------------###
def deteccionMarca():
        rate = rospy.Rate(58)    #12
        arucoNav.mark_x_list.append(arucoNav.mark_x)
        arucoNav.mark_y_list.append(arucoNav.mark_y)
        arucoNav.mark_theta_list.append(arucoNav.mark_theta_median)
        rate.sleep()
        if len(arucoNav.mark_x_list)>10 and len(arucoNav.mark_y_list)>10
        and (arucoNav.mark_theta_list)>10:
                arucoNav.mark_x_list.remove(arucoNav.mark_x_list[0])
                #print ('eje x',arucoNav.mark_x_list)
                arucoNav.mark_y_list.remove(arucoNav.mark_y_list[0])
                #print ('eje y',arucoNav.mark_y_list)
                arucoNav.mark_theta_list.remove(arucoNav.mark_theta_list[0])
                #print ('tita',arucoNav.mark_y_list)

                if (arucoNav.mark_x_list[8] == arucoNav.mark_x_list[2]) or
                (arucoNav.mark_y_list[8]== arucoNav.mark_y_list[2]):
                        arucoNav.marca_detectada=0
                        print ('No detecto la marca')
```

```
                else:
                        arucoNav.marca_detectada=1
                        print ('Marca detectada')


###-------------------FUNCION PARA GIRAR-------------------------------###

def girar():
        print ("Comienzo a girar")
        rospy.Timer(rospy.Duration(10),my_callback,oneshot=True)

        while (not arucoNav.girocompleto==1) and not rospy.is_shutdown():
                arucoNav.vel.angular.z=-1.256637061
                pub.publish(arucoNav.vel)
                #print("Estoy girando")

        print("Giro terminado")
        arucoNav.girocompleto=0
        arucoNav.vel.angular.z=0.0
        pub.publish(arucoNav.vel)

def my_callback(event):
        print("Han pasado 5 segundos")
        arucoNav.girocompleto=1

###-------------------FUNCION PARA AVANZAR-----------------------------###

def avanzar():
        print ("Comienzo a avanzar")
        rospy.Timer(rospy.Duration(1.0),my_callback_1,oneshot=True)
        while (not arucoNav.avancecompleto==1) and not rospy.is_shutdown():
                arucoNav.vel.linear.x=1.0
                pub.publish(arucoNav.vel)
                print ("Estoy avanzando")
        print ("Avance terminado")
        arucoNav.avancecompleto=0
        arucoNav.vel.linear.x=0.0
        pub.publish(arucoNav.vel)

def my_callback_1(event):
        print ("Ha pasado 1 segundo")
        arucoNav.avancecompleto=1

###-------------------FUNCION DE SEGUIMIENTO---------------------------###

def tracking():

  deteccionMarcaInicio()
  #print ('Marca Inicio',arucoNav.marca_detectada_ini)

  if (arucoNav.marca_detectada_ini==1):
        #print ("Estoy dentro de marca detectada y avanzo")
        if (arucoNav.fin==0):
          tiempo = rospy.get_time()
          rate = rospy.Rate(10000)
          arucoNav.mark_x_list.append(arucoNav.mark_x)
          arucoNav.robot_x_list.append(arucoNav.robot_x)
          arucoNav.mark_y_list.append(arucoNav.mark_y)
```

```
arucoNav.robot_y_list.append(arucoNav.robot_y)
arucoNav.mark_theta_list.append(arucoNav.mark_theta)
arucoNav.robot_theta_list.append(arucoNav.robot_theta)
arucoNav.tiempo_list.append(tiempo)
rate.sleep()

tiempoActualizado = arucoNav.tiempo_list[0] + 130.0
tiempoActualizado1 = arucoNav.tiempo_list[0] + 1.0

# Trayectoria de la marca
vMarca=0.2
wMarca=1.8

# Error relativo
xe = (arucoNav.robot_x-arucoNav.mark_x)*cos(arucoNav.mark_theta)+
(arucoNav.robot_y-arucoNav.mark_y)*sen(arucoNav.mark_theta)
#print ("error en x", xe)
ye = -(arucoNav.robot_x-arucoNav.mark_x)*sen(arucoNav.mark_theta)+
(arucoNav.robot_y-arucoNav.mark_y)*cos(arucoNav.mark_theta)
#print ("error en y",ye)
titae = arucoNav.robot_theta-arucoNav.mark_theta
#print ("error en el angulo tita",titae)

#Almacenar los valores de los errores
rate = rospy.Rate(10000)
arucoNav.error_x_list.append(xe)
#print ("error en x",arucoNav.error_x_list)
arucoNav.error_y_list.append(ye)
#print ("error en y",arucoNav.error_y_list)
arucoNav.error_theta_list.append(titae)
#print ("error en tita",arucoNav.error_theta_list)
rate.sleep()

# Ley de control
k1 = 0.2
k2 = 60.0
k3 = 80.0
Z1 = xe
Z2 = ye
Z3 = tan(titae)

#print (Z1)
#print (Z2)
#print (Z3)

arucoNav.velocidad_linear=(vMarca-(k1*abs(vMarca)*Z1))
#print ("velocidad lineal",arucoNav.velocidad_linear)
arucoNav.velocidad_angular=wMarca-(k2*vMarca*Z2)-
(k3*abs(vMarca)*Z3)
#print ('velocidad angular',arucoNav.velocidad_angular)

arucoNav.vel.linear.x= (arucoNav.velocidad_linear)
arucoNav.vel.angular.z= arucoNav.velocidad_angular

#Almacenar los valores de las velocidades
rate = rospy.Rate(10000)
arucoNav.velocidad_linear_list.append(arucoNav.vel.linear.x)
arucoNav.velocidad_angular_list.append(arucoNav.vel.angular.z)
```

```
            rate.sleep()

            #deteccionMarca()

            if (tiempo > tiempoActualizado):
              #print ('dentro del if')
              arucoNav.vel.angular.z = 0.0
              arucoNav.vel.linear.x = 0.0
              arucoNav.vel.linear.y = 0.0
              a = len(arucoNav.tiempo_list)
              arucoNav.longTiempo.append(a)

              arucoNav.t = arucoNav.tiempo_list[0:arucoNav.longTiempo[0]]
              a = len(arucoNav.t)

              arucoNav.xM = arucoNav.mark_x_list[0:arucoNav.longTiempo[0]]
              arucoNav.xR = arucoNav.robot_x_list[0:arucoNav.longTiempo[0]]
              arucoNav.yM = arucoNav.mark_y_list[0:arucoNav.longTiempo[0]]
              arucoNav.yR = arucoNav.robot_y_list[0:arucoNav.longTiempo[0]]
              arucoNav.titaM = arucoNav.mark_theta_list[0:arucoNav.longTiempo[0]]
              arucoNav.titaR = arucoNav.robot_theta_list[0:arucoNav.longTiempo[0]]

              arucoNav.errorX=arucoNav.error_x_list[0:arucoNav.longTiempo[0]]
              arucoNav.errorY=arucoNav.error_y_list[0:arucoNav.longTiempo[0]]
              arucoNav.errorTita=arucoNav.error_theta_list[0:arucoNav.longTiempo[0]]

              arucoNav.velocidad_linear=arucoNav.velocidad_linear_list
              [0:arucoNav.longTiempo[0]]
              arucoNav.velocidad_angular=arucoNav.velocidad_angular_list
              [0:arucoNav.longTiempo[0]]

              arucoNav.diferencia_eje_x= arucoNav.dif_x_list[0:arucoNav.longTiempo[0]]
              arucoNav.diferencia_eje_y= arucoNav.dif_y_list[0:arucoNav.longTiempo[0]]
              arucoNav.coseno_tita = arucoNav.cos_tita_list[0:arucoNav.longTiempo[0]]
              arucoNav.seno_tita = arucoNav.sen_tita_list[0:arucoNav.longTiempo[0]]

              representacion()

            pub.publish(arucoNav.vel)
            else:
                print ("Estoy dentro de marca no detectada y giro")
                girar()
                avanzar()

###----------POSICION DE LA MARCA (desde origen [0,0])----------###

def arucoCallback(aruco_pose_message):
        arucoNav.mark_x=aruco_pose_message.pose.position.x
        arucoNav.mark_y=aruco_pose_message.pose.position.y
        arucoNav.mark_z=aruco_pose_message.pose.position.z
        rotation=aruco_pose_message.pose.orientation
        angles=tf.transformations.euler_from_quaternion(quaternion=(rotation.x,
        rotation.y, rotation.z, rotation.w))
        arucoNav.mark_alpha=numpy.rad2deg(angles[0])
        arucoNav.mark_beta=numpy.rad2deg(angles[1])
        arucoNav.mark_theta=numpy.rad2deg(angles[2])
```

```
###----------POSICION DEL ROBOT (desde origen [0,0])----------###
def robotCallback(robot_pose_message):
        arucoNav.robot_x=robot_pose_message.pose.pose.position.x
        arucoNav.robot_y=robot_pose_message.pose.pose.position.y
        arucoNav.robot_z=robot_pose_message.pose.pose.position.z

        rotation_robot=robot_pose_message.pose.pose.orientation
        angles_robot=tf.transformations.euler_from_quaternion
        (quaternion=(rotation_robot.x, rotation_robot.y, rotation_robot.z,
        rotation_robot.w))

        arucoNav.robot_alpha=numpy.rad2deg(angles_robot[0])
        arucoNav.robot_beta=numpy.rad2deg(angles_robot[1])
        arucoNav.robot_theta=numpy.rad2deg(angles_robot[2])


def listener():
        rospy.Subscriber('/campero/robotnik_base_control/odom',Odometry,
        robotCallback)
        rospy.Subscriber('aruco_pose', PoseStamped, arucoCallback)


###----------REPRESENTACION GRAFICA----------###
def representacion ():

        plt.figure()
        plt.plot(arucoNav.t, arucoNav.xM,'b')
        plt.plot(arucoNav.t, arucoNav.xR,'r')
        plt.xlabel('Tiempo')
        plt.title('Posicion en el eje x(m)')

        plt.figure()
        plt.plot(arucoNav.t, arucoNav.yM, 'b')
        plt.plot(arucoNav.t, arucoNav.yR, 'r')
        plt.xlabel('Tiempo')
        plt.title('Posicion en el eje y(m)')

        plt.figure()
        plt.plot(arucoNav.t,arucoNav.titaM, 'b')
        plt.plot(arucoNav.t,arucoNav.titaR,'r')
        plt.xlabel('Tiempo')
        plt.title('Orientacion del angulo theta')

        plt.figure()
        plt.plot(arucoNav.xM, arucoNav.yM, 'b')
        plt.plot(arucoNav.xR, arucoNav.yR, 'r')
        plt.xlabel('Eje x(m)')
        plt.title('Evolucion de x(m) e y(m)')

        plt.figure()
        plt.plot(arucoNav.t,arucoNav.velocidad_linear,'r')
        plt.title('Velocidad Lineal (m/s)')

        plt.figure()
        plt.plot(arucoNav.t,arucoNav.velocidad_angular,'r')
        plt.title('Velocidad angular')

        plt.figure()
```

```
        plt.plot(arucoNav.t,arucoNav.errorX,'g')
        plt.title("Error en el eje x")

        plt.figure()
        plt.plot(arucoNav.t,arucoNav.errorY,'g')
        plt.title("Error en el eje y")

        plt.figure()
        plt.plot(arucoNav.t,arucoNav.errorTita,'g')
        plt.title("Error en el angulo tita")

        plt.show()


###-------------------------------------MAIN-------------------------------------###

if __name__=='__main__':
        try:
                rospy.init_node('arucoTracking')
                pub = rospy.Publisher('/campero/cmd_vel',Twist,queue_size=10)
                arucoNav=arucoTracking()
                listener()
                rospy.sleep(1)
                print ("Inicio del seguimiento de la marca")
                #while not arucoNav.fin and not rospy.is_shutdown():
                while not rospy.is_shutdown():
                        tracking()
                if rospy.is_shutdown():
                        rospy.loginfo("Programa cancelado")
                else:
                        rospy.loginfo("El robot ha llegado")
        except rospy.ROSInterruptException:
                rospy.loginfo("Test de navegacion finalizado")
```

Anexos D

Manual de usuario

En este anexo se explican los pasos que hay que seguir para poder llevar a cabo la simulación del seguimiento de la marca ArUco.

Para llevar a cabo dicha simulación se usa Terminator [14], el cual es un emulador que permite crear múltiples terminales en una sola ventana. En este caso es bastante conveniente ya que se van a lanzar numerosas terminales.

El primer paso será crear el movimiento de la marca, este proceso va a tener que repetirse cada vez que se cierre Terminator. Para ello vamos a seguir los siguientes pasos:

- Ir a la carpeta campero_ws: **$ cd campero_ws**
- Ir a la carpeta que se ha creado antes, con el contenido que se ha explicado en el Anexo A: **$ cd src/campero/campero_sim/campero_gazebo/marca**
- Crear una carpeta llamada build: **$ mkdir build**
- **$ cd build**
- **$ cmake ../**
- **$ make**
- **$ export GAZEBO_PLUGIN_PATH=‘pwd‘:$GAZEBO_PLUGIN_PATH**

Una vez creado el movimiento de la marca, en la misma terminal ejecutamos el siguiente comando: **$ roslaunch campero_navigation campero_nav.launch**

En una nueva terminal ejecutamos lo siguiente: **$ roslaunch aruco_ros single.launch**

A continuación, para poder detectar la marca ejecutamos en una nueva terminal el siguiente comando: **$ rosrun image_view image_view image:=/aruco_single/result**. Se abrirá una ventana como se la que se muestra en la figura D.1.

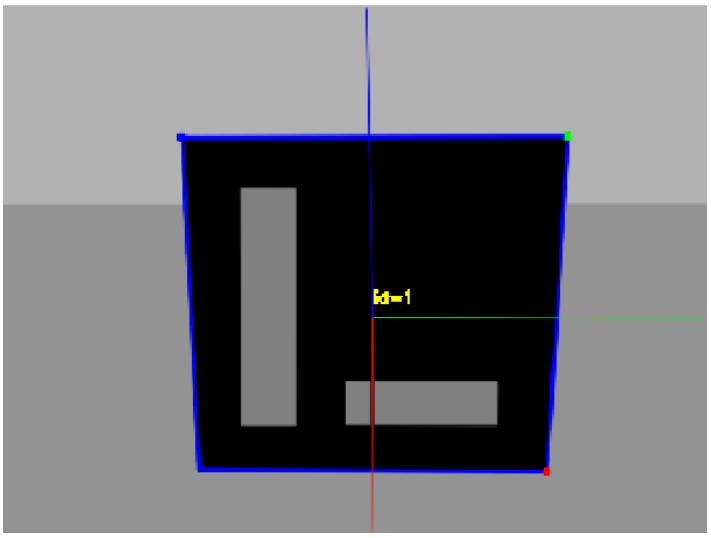

Figura D.1: Visualización de la marca

Para poder crear la transformación marker_tf, se lanza en una nueva terminal el siguiente comando **$ rosrun aruco_ros marker_tf.py**

Una vez se tiene la transformación aruco_tf, se necesita conocer la posición en la que se encuentra dicha transformación y por lo tanto se lanza el siguiente fichero **$ rosrun aruco_ros transform_listener.py**

Finalmente se lanzará el fichero que contiene el control de seguimiento, es decir, ArucoTracking.py. Para ello en una nueva terminal se escribe lo siguiente: **$ rosrun campero_navigation ArucoTracking.py**