UNIVERSIDAD NACIONAL DEL COMAHUE

FACULTAD DE INGENIERÍA

PROYECTO INTEGRADOR PROFESIONAL

INGENERÍA ELECTRÓNICA

BIBLIOTECA DE SOFTWARE PARA SIMULAR EL COMPORTAMIENTO DE UN ROBOT 4WD SSMR Y TESTEAR CONTROLADORES DE SEGUIMIENTO DE TRAYECTORIAS

AUTOR:

D'Angelo Bruno

DIRECTOR:

Sansoni Sebastian

CO-DIRECTOR:

Slawiñski Emanuel

Neuquén, 2022

Resumen

El presente trabajo surge ante la necesidad del Grupo de Investigación en Vehículos y Sistemas Inteligentes de la Facultad de Ingeniería de la Universidad Nacional del Comahue, el cual requiere un ambiente de prueba seguro para evaluar el desempeño de algoritmos de control de robots, como etapa previa al despliegue final del mismo.

A fin de minimizar el riesgo de daños materiales y costos de despliegue, se propone como solución el evaluar los algoritmos en un entorno de simulación open-source ampliamente utilizado. Es por esto que se desarrolla una biblioteca de software que integra el entorno de simulación *Gazebo*, la generación de un modelo tridimensional de un robot móvil de cuatro ruedas de dirección deslizante con tracción independiente (4WD SSMR) y algoritmos de control de bajo y alto nivel.

La comunicación entre los diferentes módulos se realiza utilizando los estándares y las herramientas que ofrece la plataforma de software *ROS*. La elección de utilizar ROS se debe a que los algoritmos pueden ser implementados en Python o en C++ y a que puede ser utilizado tanto dentro de la simulación como en el despliegue físico del robot.

Se somete la biblioteca desarrollada a distintos escenarios, implementando y haciendo uso de algoritmos de control de velocidad, seguimiento de caminos y seguimiento de trayectorias. Además, se calculan distintas métricas de desempeño del seguimiento de caminos y trayectoria ante diferentes geometrías de referencia.

La biblioteca se documenta adecuadamente para facilitar su uso, edición y ampliación. Por la misma razón, queda a total disposición del grupo de investigación, y cualquier otro interesado, en un repositorio público de GitLab.

Palabras clave: SSMR, simulación, seguimiento de camino, seguimiento de trayectoria, Gazebo, ROS.

Abstract

This work arises from the needs of the Smart Vehicles and Systems Research Group of Engineering Faculty of Universidad Nacional del Comahue, which requires a safe test environment to assess the performance of control algorithms for their robots, as a previous stage before actual deployment.

In order to minimize risk of material damage and deployment costs, this research proposal is to test the control algorithms in a popular open-source simulation environment. For this reason, a software library is developed to combine the *Gazebo* simulator, the 3D model building of a four-wheel drive skid-steer mobile robot (4WD SSMR) and both low and high level control algorithms.

Communication between different modules is achieved using the standards and tools offered by the software platform called *ROS*. ROS was chosen because it allows algorithms to be implemented either in Python or C++ and because it can be used both in simulations and in real life robot deployment.

The developed library it tested under different scenarios by implementing and using different algorithms, such as velocity control, path following and trayectory following algorithms. Different metrics are chosen to quantify the performance of path and trayectory following against different references.

The library is properly documented to promote its use, modification and enhancememnt. For the same reason, the library is available for the Research Group and everyone else in a public GitLab repository.

Keywords: SSMR, robot simulation, path following, trajectory tracking, Gazebo, ROS.

Agradecimientos

Agradezco especialmente a mi madre y a mi padre, por brindarme su apoyo total siempre, aún cuando eso significó alejarme de la carrera por tiempo indeterminado.
A mi novia Katy, que me acompaña desde hace años, que me impulsa hacia nuevas aventuras, que postergó sus planes porque yo "me recibía pronto", y porque sin ella no estaría acá.
A mi amigo Brunito De León, compañero de carrera, de docencia, de lucha, de mates y de la vida.
A mi abuela, que no llegó a verme recibido, pero nunca perdió la fe en mí.
A mi director, Ing. Sebastián Sansoni, por demostrarme cómo actúa un verdadero director de tesis y por ser directo y honesto conmigo en todo momento.
A todas las personas que defienden la educación y la salud pública, gratuita y de calidad.

Índice general

Capítulo 1

Introducción

El Grupo de Investigación en Vehículos y Sistemas Inteligentes (GIVSI) de la Facultad de Ingeniería de la Universidad Nacional del Comahue (UNCo) se encuentra desarrollando un pequeño vehículo eléctrico de cuatro ruedas de dirección deslizante con tracción independiente (4WD SSMR , del inglés four wheel drive Skid Steering Mobile Robot) en cada una de ellas [Ortiz and Wirth, 2021]. Esta actividad está enmarcada en un proyecto de investigación [PIN, 2020] que incluye, entre varios objetivos específicos, el diseño y desarrollo de un sistema de navegación autónoma para vehículos de trabajo en plantaciones frutícolas.

Generalmente, las aplicaciones que involucran robots se evalúan en un entorno controlado, lo cual implica que el ensayo no represente correctamente el entorno en dónde va a desempeñar sus tareas normalmente. Por ejemplo, cuando se desea testear un algoritmo en un robot móvil, es necesario trasladar el dispositivo al lugar específico de testeo, mantener la carga de las baterías estable durante todo el ensayo, garantizar que las condiciones ambientales sean estables y repetibles, entre otras complejidades observadas en el ciclo de desarrollo de un dispositivo que va a operar en ambientes externos.

Muchas de estas dificultades se pueden minimizar incluyendo dentro del ciclo de desarrollo e implementación el uso de simuladores que estimen la dinámica de los cuerpos rígidos en entornos 3D. El modelado, control y simulación de estos vehículos es un tema que ha mantenido la atención en el ambiente científico por muchos años.

Los 4WD SSMR son utilizados típicamente como vehículos todo terreno, debido a su robusta estructura mecánica y a que se desempeñan satisfactoriamente en superficies de difícil tránsito, (lodo, arena o nieve, por ejemplo). Sin embargo, poseen la particularidad de requerir un deslizamiento lateral entre las ruedas y el suelo para que el vehículo cambie su orientación. Esto complejiza el sistema de control, ya que se debe analizar y modelar las

propiedades dinámicas del sistema [Kozłowski and Pazderski, 2004].

Por lo tanto, se propone desarrollar una biblioteca de software que contenga los módulos necesarios para modelar e implementar un modelo tridimensional de un vehículo en configuración SSMR dentro de un entorno de simulación, implementar estrategias de seguimiento de trayectoria y evaluar el desempeño de las mismas.

1.1. Objetivos

1.1.1. Objetivo general

El objetivo general de este trabajo es desarrollar e implementar una biblioteca de software de código abierto, que permita simular el comportamiento de un robot 4WD SS. Esta biblioteca estará enfocada en el desarrollo y testeo de algoritmos de control para el seguimiento de trayectorias en estos tipos de vehículos.

1.1.2. Objetivos específicos

- Estudiar el modelado cinemático y dinámico de robtos con configuración SSMR.
- Definir e implementar una estrategia de control lineal para realizar seguimiento de trayectorias.
- Implementar un modelo tridimensional de un SSMR dentro de un entorno de simulación.
- Implementar en una biblioteca de Python los controladores y la comunicación con el entorno de simulación.
- Definir al menos dos métricas para determinar el desempeño de las estrategias de control implementadas.
- Documentar el trabajo realizado, a fin que el mismo pueda ser replicado, editado o ampliado por otras personas en el futuro.

1.2. Estructura del trabajo

El presente informe se ha organizado de la siguiente manera:

- Capítulo 1: se resume la motivación del proyecto y la estructura del mismo.
- Capítulo 2: se presentan dos tipos de desplazamiento de vehículos, se obtienen las ecuaciones que modelan su movimiento y se simulan las mismas en la plataforma MATLAB® .
- Capítulo 3: se presenta el actuador que impartirá el torque a las ruedas dentro del entorno de simulación. También se desarrolla el controlador de velocidad de las ruedas implementado en Python.
- Capítulo 4: se establecen las variables de error de posición y orientación, así como las métricas de desempeño a utilizar. En base a las mismas, se implementan dos controladores de posición del vehículo y se compara su desempeño.
- Capítulo 5: se establece la estrategia de seguimiento de trayectoria, a partir de la cual se implementan dos controladores en Python y se compara su desempeño.
- Capítulo 6: se detallan las distintas partes principales del entorno de simulación y se presenta el modelo tridimensional del robot utilizado en las simulaciones.
- Capítulo 7: se presenta la estructura de la biblioteca desarrollada y la interacción entre sus distintos módulos, a fin de facilitar su navegación y edición.
- Capítulo 8: se concluye sobre los resultados obtenidos.
- Bibliografía: se enumera el material consultado para la elaboración de este trabajo.
- Anexo: desarrollo de las ecuaciones pertinentes al controlador del Capítulo 5, consideradas demasiado exhaustivas para ser incluídas en el mismo.

Capítulo 2

Modelización del vehículo

Este capítulo presenta al lector dos tipos de vehículos caracterizados por el modo en se desplazan. Se recopila la información del material bibliográfico que permite obtener las ecuaciones diferenciales que explican el comportamiento de estos vehículos a nivel cinemático y dinámico. En lo posible, se evitará profundizar en nociones físico-matemáticas que complejicen la lectura en demasía. Una vez presentados los modelos, se explica la metodología necesaria para implementar y resolver dichas ecuaciones en la plataforma MATLAB® , permitiendo simular el comportamiento de un vehículo ante diferentes comandos de velocidad y presentar los resultados mediante gráficos.

2.1. Modos de desplazamiento

Existen diferentes formas en las que un vehículo rodado puede moverse en el espacio. Dependiendo de la cantidad de ruedas y de su configuración, tanto espacial como de transferencia de potencia, se dan tipos de movimiento muy diferentes entre sí. En particular, dentro de los distintos tipos de vehículos rodados, en esta tesis se analizarán dos modos de desplazamiento: diferencial y de maniobra por deslizamiento (de aquí en más, skid-steer). La Figura (2.1) [1,2] muestra un ejemplo de estos vehículos.

El modo diferencial consiste en un vehículo con dos ruedas y, generalmente, un tercer punto de apoyo que mantiene el robot en posición horizontal, tal como se representa en la Figura (2.2). Si ambas ruedas giran en el mismo sentido y con la misma velocidad angular, el vehículo se desplaza con velocidad lineal constante hacia adelante o atrás. Para cambiar la orientación del vehículo, se debe generar una diferencia entre la velocidad angular de ambas

[1]Imagen extraída de Benemérita Universidad Autónoma de Puebla

[2]Imagen extraída del sitio oficial de Bobcat®

(a) Diseño en CAD de robot diferencial. (b) Mini cargadora skid-steer.

Figura 2.1: Ejemplos reales de ambos tipos de vehículos: el diseño en **(a)** es similar a una aspiradora robot comercial, la maquinaria en **(b)** es utilizada en tareas de construcción.

ruedas.

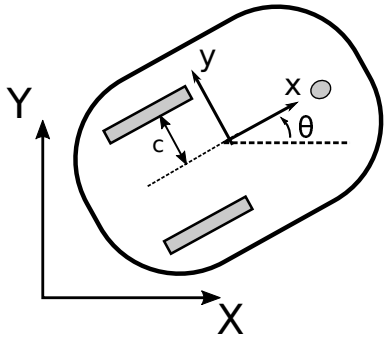


Figura 2.2: Esquema de robot diferencial típico, compuesto por dos ruedas y un tercer punto de apoyo.

Un vehículo de cuatro ruedas en modo skid-steer es similar a un clásico automóvil, con la diferencia de que sus ruedas delanteras no son maniobrables, por lo tanto, sus cuatro ruedas tienen la misma orientación que el chasis, en todo momento. Por esta razón, el vehículo sólo puede girar cuando existe una diferencia de velocidad angular entre las ruedas del lado izquierdo y las del lado derecho. Este modo de giro requiere inherentemente del deslizamiento lateral de las ruedas, he aquí la razón de su nombre.

En la siguiente sección se describen las ecuaciones de movimiento de cada modelo. En ambos casos, y en el resto del trabajo, se considerará únicamente el caso de movimiento planar, donde el vehículo sólo se desplaza horizontalmente sobre una superficie plana.

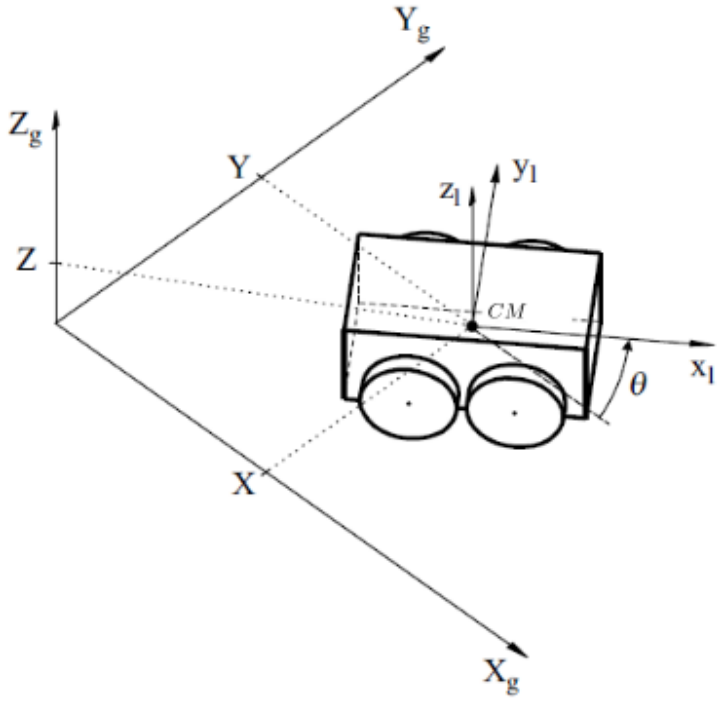


Figura 2.3: Vehículo SSMR en un marco inercial. Imagen obtenida de [Kozłowski and Pazderski, 2004]

2.2. Cinemática

Los modelos cinemáticos describen la relacion entre la velocidad de las ruedas y la velocidad del robot. Son ampliamente utilizados para obtener estimaciones de los estados y controlar el movimiento de los vehículos, se obtienen a partir de propiedades geométricas y de ecuaciones fundamentales de la cinemática.

2.2.1. Hipótesis simplificativas y ecuaciones

Dado un sistema de referencia ortonormal no inercial (X_g, Y_g, Z_g), se puede representar la posición del vehículo con las coordenadas de su centro de masa (X, Y, Z), como se observa en la Figura (2.3). Es útil asignar un marco de coordenadas locales (x_l, y_l, z_l) en dicho centro de masa. De este modo, se puede indicar la orientación del vehículo θ como el ángulo entre el marco de coordenadas locales y el de coordenadas globales. Se hará uso de letras mayúsculas para referirse a coordenadas globales y minúsculas para coordenadas locales.

Se llamará $\mathbf{q} = \begin{bmatrix} X & Y & \theta \end{bmatrix}^T$ al vector de estados del vehículo, luego $\dot{\mathbf{q}} = \begin{bmatrix} \dot{X} & \dot{Y} & \dot{\theta} \end{bmatrix}^T$ representa el vector de velocidades del vehículo en coordenadas globales. En coordenadas locales, las velocidades pueden representarse con el vector de velocidad lineal $\mathbf{v} = \begin{bmatrix} v_x & v_y \end{bmatrix}$ y

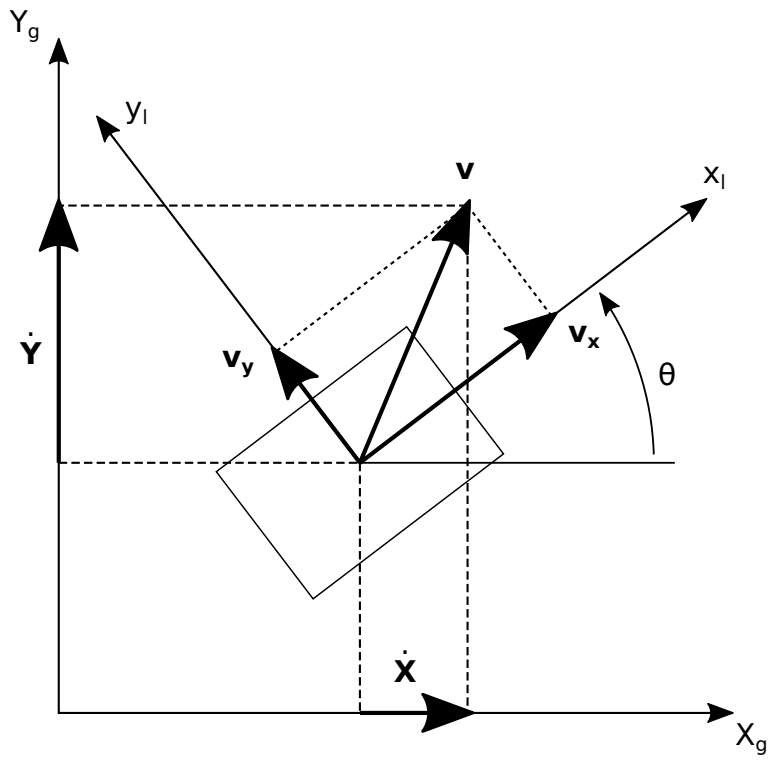


Figura 2.4: Relación geométrica entre las distintas velocidades del vehículo.

la velocidad angular respecto al eje z, denotada por ω_z. La Figura (2.4) muestra las relaciones geométricas entre las componentes de velocidad, válidas para cualquier tipo de vehículo, de dónde se obtiene:

$$\begin{bmatrix} \dot{X} \\ \dot{Y} \\ \dot{\theta} \end{bmatrix} = \begin{bmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} v_x \\ v_y \\ \omega_z \end{bmatrix} . \tag{2.1}$$

Es de interés obtener la dependencia de las velocidades globales del vehículo respecto a la velocidad angular de las ruedas. Al modelizar las interacciones rueda-suelo se suelen aplicar ciertas hipótesis simplificativas, en particular se destaca el concepto de *rueda ideal*: una rueda ideal es incompresible y entra en contacto con el suelo en un único punto, independientemente del grosor de la rueda. Con estas simplifaciones, se puede estipular que, en cada instante, cada i-ésima rueda entra en contacto con el piso en un punto P_i, gira con velocidad angular ω_i y cuenta con velocidad lineal lateral v_{iy} y longitudinal v_{ix}, como se representa en la Figura (2.5).

La distancia desde el eje de rotación hasta el punto de contacto P_i es el radio efectivo r_i de rotación de la rueda. Si se desprecia el deslizamiento longitudinal, la velocidad longitudinal de una rueda depende de su radio y velocidad angular

$$v_{ix} = r_i \omega_i \ .$$

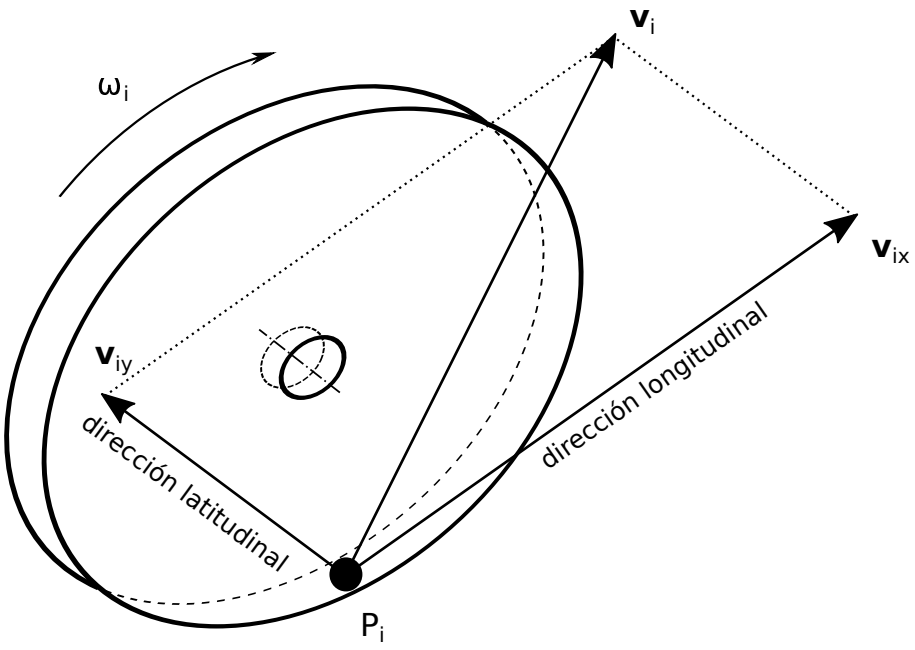


Figura 2.5: Velocidades de la i-ésima rueda.

Aún más, bajo estas condiciones puede establecerse la siguiente relación entre la velocidad de las ruedas y el vehículo, válidas tanto para un robot diferencial como para un skid-steer:

$$\omega_L = \frac{v_x - \omega_z c}{r}, \qquad \omega_R = \frac{v_x + \omega_z c}{r}. \tag{2.2}$$

donde ω_L y ω_R son las velocidades angulares de la rueda izquierda y derecha, respectivamente, y c es la mitad de la distancia entre rueda izquierda y derecha, medida desde sus puntos de contacto. Se despeja v_x y ω_z, obteniendo las siguientes expresiones equivalentes:

$$\begin{bmatrix} v_x \\ \omega_z \end{bmatrix} = r \begin{bmatrix} \dfrac{\omega_L + \omega_R}{2} \\ \dfrac{-\omega_L + \omega_R}{2c} \end{bmatrix}, \tag{2.3}$$

La relación presentada en la Ecuación (2.3) es de gran utilidad pues permite expresar el sistema en función de las velocidades del vehículo o de las ruedas, según resulte conveniente.

En el caso de un vehículo diferencial, se puede despreciar el deslizamiento lateral de las ruedas, es decir que $v_y = 0$. De esta forma, se elimina v_y de la Ecuación (2.1), obteniéndose el siguiente modelo cinemático:

$$\begin{bmatrix} \dot{X} \\ \dot{Y} \\ \dot{\theta} \end{bmatrix} = \begin{bmatrix} \cos\theta & 0 \\ \sin\theta & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} v_x \\ \omega_z \end{bmatrix}, \tag{2.4}$$

Nótese que el modelo obtenido expresa las derivadas de los estados del vehículo en función de la velocidad lineal longitudinal v_x y la velocidad angular ω_z del vehículo, que pueden considerarse como las entradas de control del modelo.

Por otro lado, para el caso de un vehículo skid-steer la velocidad lateral de las ruedas no es despreciable al girar. Por lo tanto, surge la necesidad de utilizar un modelo que sí considere el deslizamiento lateral. Tal es el caso del modelo presentado en [Caracciolo et al., 1999], que establece una relación entre la velocidad lateral del vehículo y la velocidad angular del mismo, la cual se presenta a continuación.

Cuando un vehículo se desplaza en el plano, puede considerarse como un cuerpo rígido cuyo centro de masa (CM) describe una circunferencia en torno a su centro instantáneo de rotación (ICR, por sus siglas en inglés), cuyas coordenadas son (x_{ICR}, y_{ICR}), como puede observarse en la Figura (2.6). El centro de masa se ubica a las distancias a y b del eje frontal y del eje trasero, respectivamente. En [Caracciolo et al., 1999] se observa que si x_{ICR} quedara fuera del espacio entre el eje frontal y el trasero, el vehículo deslizaría sobre el eje y perdiendo estabilidad. Para evitar esto, impone en el modelo la siguiente restricción

$$v_y + x_{ICR}\dot{\theta} = 0, \quad -b < x_{ICR} < a. \tag{2.5}$$

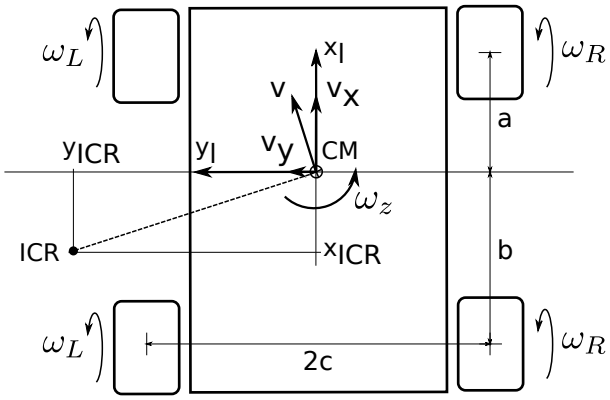


Figura 2.6: Ubicación del centro instantáneo de rotación de un vehículo skid-steer.

Esta restricción se denomina no holonómica, pues reduce el espacio de velocidades $\dot{\mathbf{q}}$ alcanzables por el vehículo, sin reducir el espacio de posiciones alcanzables. Despejando v_y de la Ecuación (2.5) y reemplazándola en (2.1), se obtiene el siguiente sistema que describe las velocidades globales admisibles del vehículo:

$$\dot{q} = \begin{bmatrix} \cos\theta & x_{ICR}\sin\theta \\ \sin\theta & -x_{ICR}\cos\theta \\ 0 & 1 \end{bmatrix} \begin{bmatrix} v_x \\ \omega_z \end{bmatrix}. \tag{2.6}$$

Habiendo presentado las ecuaciones que modelan la cinemática de ambos tipos de vehículos, se procede a explicar cómo serán implementadas y simuladas.

2.2.2. Resolución de ODEs en MATLAB®

En la sección anterior se muestra que los modelos cinemáticos se representan mediante ecuaciones diferenciales. Por lo tanto, para simular el comportamiento de un vehículo es necesario resolver dichas ecuaciones, para lo cual se propone utilizar MATLAB® , como prueba de concepto, y el simulador Gazebo, para simulaciones más complejas. En esta sección se realiza una breve introducción al uso general del *solver* de MATLAB® , finalizando con su implementación para resolver las ecuaciones de movimiento del robot skid-steer.

MATLAB® cuenta con una suite de *solvers* (del inglés, solucionador) de ODEs (por sus siglas en inglés, ecuaciones diferenciales ordinarias) de primer orden. Dependiendo del tipo de ODE y de la precisión deseada, resulta conveniente elegir un *solver* en particular, pero por lo general se recomienda utilizar el *solver ode45* y analizar su desempeño antes de optar por otro [MathWorks, 2022]. El algoritmo del *ode45* se basa en una fórmula explícita de Runge-Kutta(4,5), y puede resolver una ODE de primer orden o un sistema de ecuaciones de la forma $y' = f(t, y)$. Esto resulta conveniente, pues permite resolver una ODE de orden n (con $n \geq 2$) si primero se lo reescribe en un sistema equivalente de n ecuaciones de primer orden.

La sintaxis básica de uso de este *solver* puede observarse en el Código (2.1), donde se utilza *ode45* para resolver la ODE $y' = 2t$, en el intervalo $[0, 5]$ bajo la condición inicial $y_0 = 0$. Como se observa en la línea 3, el primer argumento de entrada es la expresión que describe la ODE, pero con la inclusión del identificador de función "@". El objetivo de la expresión $@(t, y)$ es indicarle al programa que lo que sigue es una función cuyas variables independientes son t e y. Nótese que, a pesar de que la ecuación diferencial no depende de y, el *solver* requiere que se ingrese $@(t, y)$ en todos los casos.

```
1 tspan = [0 5];
2 y0 = 0;
3 [t,y] = ode45(@(t,y) 2*t, tspan, y0);
```

Código 2.1: Ejemplo básico de uso. [MathWorks, 2022]

Si se quiere resolver una ODE de segundo orden, que dependa de más variables que t e y, es posible hacerlo realizando las sustituciones y los pasos adecuados. Por ejemplo, si se tiene la ODE

$$y'' = \frac{A}{B} t y,$$

debe reescribirse como un sistema de primer orden (implementado en el Código (2.2)):

$$\begin{matrix} {y_1}' = y_2 \\ {y_2}' = \frac{A}{B} t y_1 \end{matrix} .$$

```
1 function dydt = odefcn(t,y,A,B)
2    dydt = zeros(2,1);
3    dydt(1) = y(2);
4    dydt(2) = (A/B)*t.*y(1);
5 end
```

Código 2.2: Ejemplo de resolución con ODE de segundo orden: declaración de la función. [MathWorks, 2022]

Luego, puede resolverse el sistema con *ode45*, declarando las variables A y B y utilizando el identificador de función como $@(t, y)$ para que el *solver* interprete a $odefcn(t, y, A, B)$ como una función con sólo esas dos variables independientes (ver Código (2.3)).

```
1 A = 1;
2 B = 2;
3 tspan = [0 5];
4 y0 = [0 0.01];
5 [t,y] = ode45(@(t,y) odefcn(t,y,A,B), tspan, y0);
```

Código 2.3: Ejemplo de resolución con ODE de segundo orden: llamada al solver. [MathWorks, 2022]

Finalmente, si se desea resolver una ODE con parámetros o entradas dependientes del tiempo, dentro de la función que describe la ODE debe incorporarse una rutina que interpole el valor de dicho parámetro para un t determinado, por lo que es necesario conocer previamente los valores que toma dicho parámetro en distintos instantes. Por ejemplo, si se tiene la ODE $y' = u(t)$, la función debe ser escrita como demuestra el Código (2.4).

```
1 function dydt = odefcn(t,y,ut,u)
2    u = interp1(ut,u,t); % Interpola el conjunto (ut,u) en el instante t
3    dydt = u;
4 end
```

Código 2.4: Ejemplo de resolución de ODE con parámetros dependientes del tiempo

Si se quiere encontrar la solución ante la entrada tipo diente de sierra, representada en la Figura (2.7) y descrita por la siguiente expresión:

$$u(t) = \begin{cases} t, & \text{si } 0 \leq t < T, \\ 2T - t, & \text{si } T \leq t < 2T, \\ t - 2T, & \text{si } 2T \leq t < 3T. \end{cases} \quad , \quad \text{con } T > 0,$$

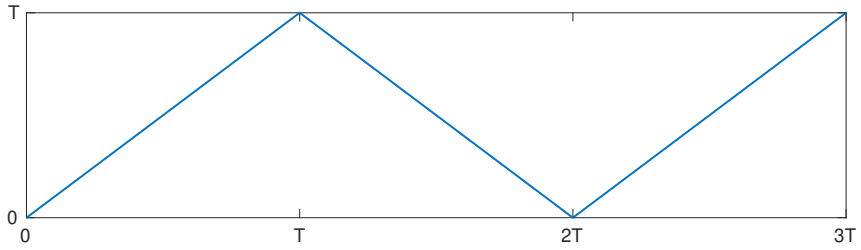


Figura 2.7: Entrada tipo diente de sierra.

debe crearse el vector de valores u asociado al vector de tiempos ut correspondiente e introducirlos como parámetros de la función al llamar al *solver*. Dado que el conjunto (ut, u) del ejemplo es lineal a tramos, basta con proveer su valor en los instantes de interés, tal como se observa en el Código (2.5), para que la interpolación lineal otorgue una aproximación precisa o, en este caso, exacta.

```
1 tspan = [0 3];
2 y0 = 1;
3 T = 1;
4 ut = [0 T 2T 3T];
5 u = [0 T 0 T];
6 [t,y] = ode45(@(t,y) odefcn(t,y,ut,u), tspan, y0);
```

Código 2.5: Ejemplo de resolución de ODE con parámetros dependientes del tiempo, llamada al solver

Hasta aquí, se han presentado distintos modos de utilizar el *solver* para casos de distintas características y complejidad, a fin de presentar la herramienta y su potencial. Sin embargo, los casos no han sido elegidos arbitrariamente: como se verá en las siguientes secciones, será necesario trabajar con ecuaciones diferenciales de segundo orden y con parámetros dependientes del tiempo.

2.2.3. Implementación y resultados en MATLAB®

Aplicando las estrategias presentadas al modelo cinemático descrito por la Ecuación(2.6), se puede implementar un algoritmo en MATLAB® que simule el comportamiento de un robot móvil en modo skid-steer. La Figura (2.8) presenta los resultados de tal simulación para el caso particular donde $x_{ICR} = 0$, $v_x = 1\ m/s$ y $\omega_z = 0.5\ rad/s$. Idealmente, se espera que el robot se mueva en una trayectoria circular de $v_x/\omega_z = 2\,m$ de radio. Para representar la pose del robot en cada instante, se optó por utilizar un triángulo a modo de flecha, cuya orientación coincide con la del robot mientras que la base indica la posición de su centro de masa. Es importante aclarar que la distancia entre cada par de triángulos consecutivos no es, necesariamente, idéntica en tiempo ni en espacio, sino que depende únicamente de los pasos temporales que adopta el *solver* para encontrar una solución dentro de los márgenes de tolerancia.

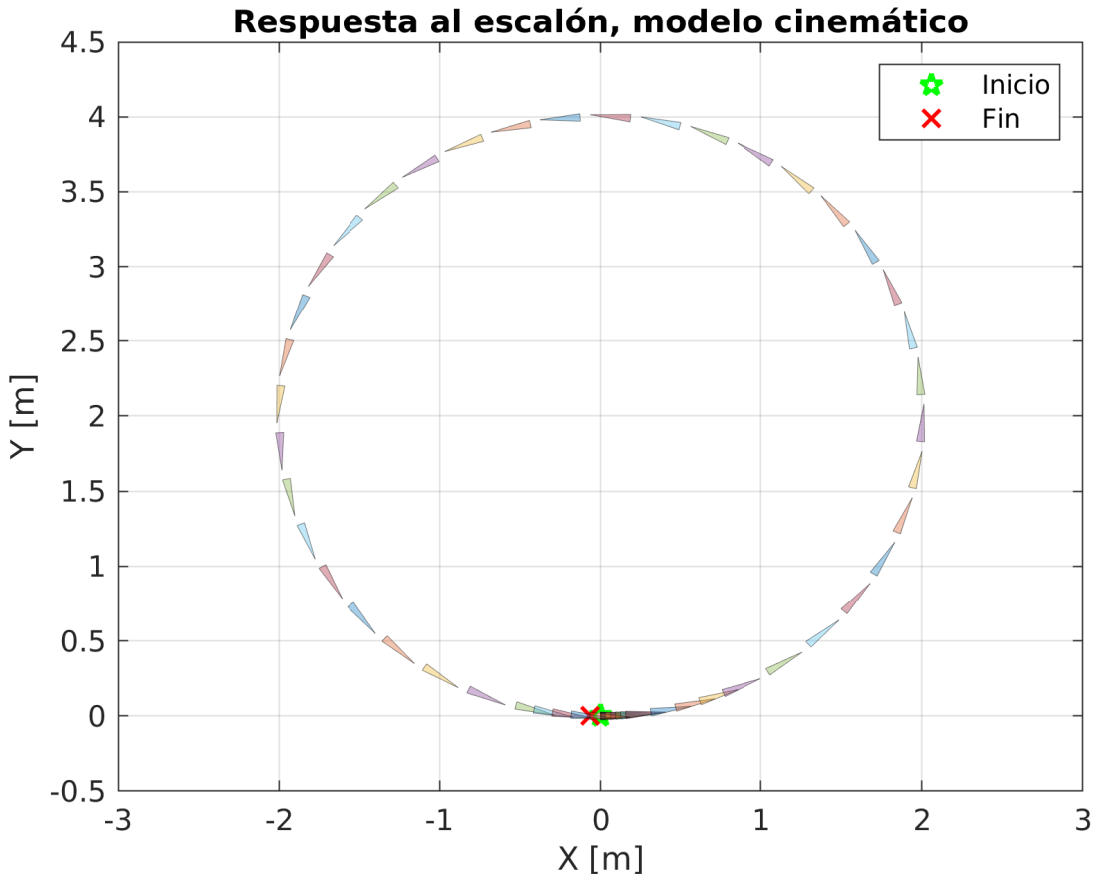


Figura 2.8: Respuesta del modelo cinemático ante entrada escalón: $v_x = 1\ m/s$, $\omega_z = 0.5\ rad/s$.

Este parámetro es específico para cada vehículo y debe ser estimado para cada escenario, pues varía según el tipo de terreno a recorrer, el estado de las ruedas y otras variables. De contar con un vehículo real, se puede estimar su x_{ICR} siguiendo el método propuesto

en [Mandow et al., 2007], que consiste en comandar, durante un intervalo finito T, que las ruedas izquierdas giren a la velocidad opuesta que las ruedas derechas (i.e. $\omega_L = -\omega_R$) y medir el giro ϕ efectivamente realizado por vehículo. Luego, aplicando la Ecuación (2.7), se obtiene un valor estimado de x_{ICR}.

$$x_{ICR} \approx \frac{\int_0^T \omega_R \, dt - \int_0^T \omega_L \, dt}{2\phi} \tag{2.7}$$

Considerando los objetivos planteados para este proyecto, en adelante se considerará $x_{ICR} = 0$, es decir, que se anula el deslizamiento lateral. Esta hipótesis no limita el alcance del trabajo ya que sólo incumbe a las simulaciones realizadas en MATLAB® y al control basado en modelo utilizado en la Sección (6.2).

2.3. Dinámica

2.3.1. Modelos de fuerzas de fricción

Los modelos dinámicos describen la relación entre las coordenadas posicionales del vehículo, sus derivadas (velocidades y aceleraciones) y las fuerzas y torques que actúan dentro y sobre el mismo. En condiciones usuales, las fuerzas que actúan sobre el vehículo son la fuerza gravitacional, el torque que ejerce el motor (o motores) sobre cada una de las ruedas, la fuerza normal que ejerce el suelo sobre las ruedas y la fricción resultante de esta interfaz. En el caso de un vehículo con cuatro ruedas ideales, la fricción existirá en cuatro puntos diferentes del mismo.

Es aquí donde surgen las mayores complicaciones al modelar con precisión la dinámica de un vehículo skid-steer, pues la fricción es un fenómeno no lineal que depende de muchas variables. Por lo general, suele simplificarse la fricción como una función de la velocidad relativa entre las superficies interactuantes. La fricción de Coulomb (también conocida como fricción estática) y la fricción viscosa, tienen la forma

$$F_f(v) = \mu_c N \,\mathrm{sgn}(v) + \mu_v v, \tag{2.8}$$

donde N es la fuerza perpendicular a la superficie, v es la velocidad relativa entre el suelo y la rueda, sgn es la función *signo*, y μ_c y μ_v son los coeficientes de fricción de Coulomb y viscosa, respectivamente. La Figura (2.9) representa ambos términos de la ecuación y su superposición.

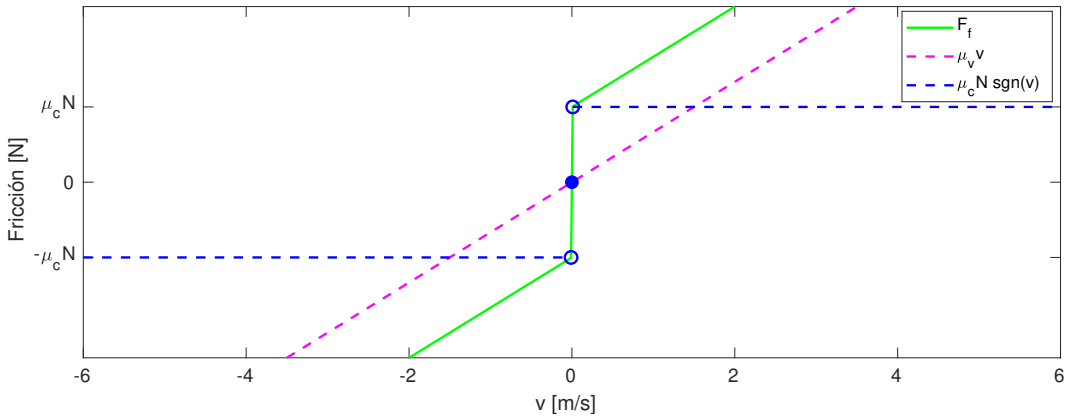


Figura 2.9: Fuerzas de fricción de Coulomb, fricción viscosa y su superposición, en función de la velocidad relativa entre superficies.

Si el vehículo se desplaza a velocidades relativamente bajas, es decir $\mu_c N \gg \mu_v v$, entonces se puede despreciar el efecto de la fricción viscosa, resultando

$$F_f(v) \approx \mu_c N \operatorname{sgn}(v). \tag{2.9}$$

Esta forma de modelar la fricción tiene la desventaja de no ser lineal, debido al uso de la función signo, lo que implica la necesidad de técnicas de control no lineal para controlar el sistema. Para superar esta desventaja, varios autores [Kozłowski and Pazderski, 2004] [Arbinolo, 2020] han optado por linealizar la función signo, aproximando la misma con la función arcotangente, es decir

$$\operatorname{sgn}(\sigma) \approx \hat{\operatorname{sgn}}(\sigma) = \frac{2}{\pi} \arctan(k_s \sigma), \tag{2.10}$$

donde $k_s \gg 1$ es una constante que determina la pendiente de la aproximación en la zona de transición, cumpliéndose

$$\lim_{k_s \to \infty} \frac{2}{\pi} \arctan(k_s \sigma) = \operatorname{sgn}(\sigma).$$

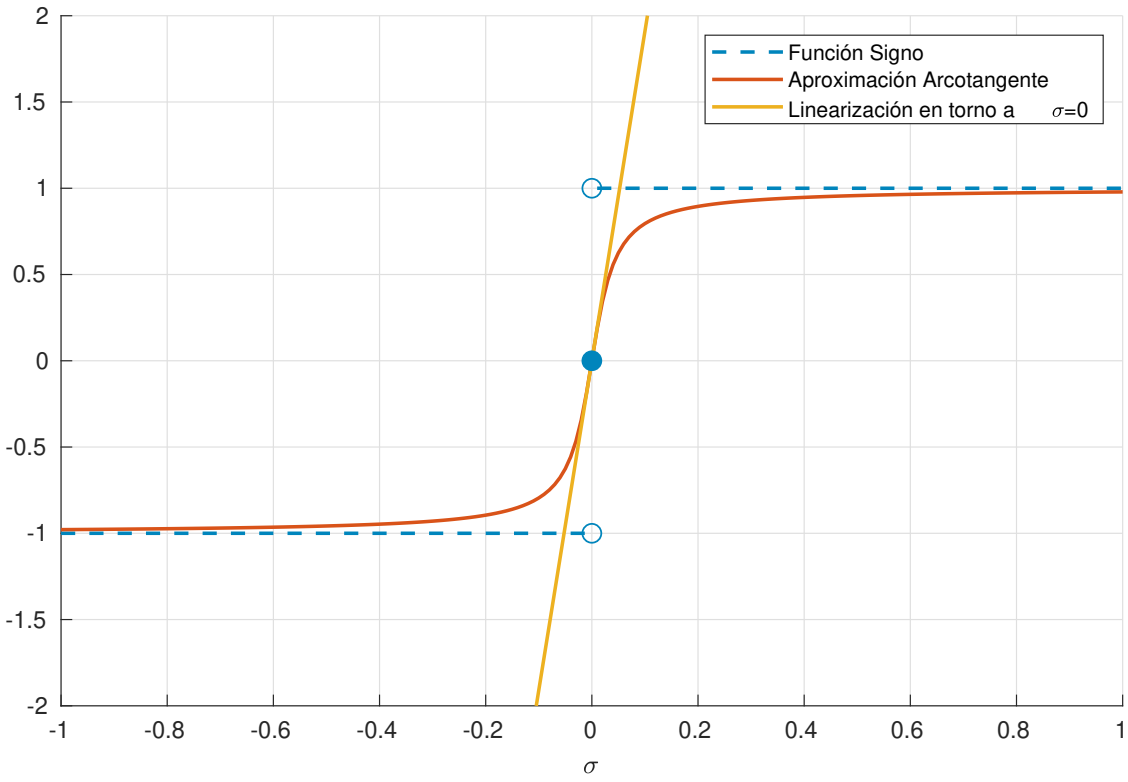


Figura 2.10: Comparación entre las distintas aproximaciones de la función signo.

Substituyendo la Ecuación (2.10) en (2.9) se obtiene

$$F_f(v) \approx \mu_c N \frac{2}{\pi} \arctan(k_s v).$$

Esta función es diferenciable en todo su dominio, luego también lo es en $v = 0$, por lo que es posible hacer una expansión de Taylor de primer orden en torno a ese punto

$$F_f(v) \approx F_f(0) + \frac{\delta F_f}{\delta v}\bigg|_{v=0} v = 0 + \mu_c N \frac{2}{\pi} \frac{k_s}{1 + (k_s v)^2}\bigg|_{v=0} v,$$

$$F_f(v) \approx NKv, \tag{2.11}$$

donde $K = \mu_c k_s \frac{2}{\pi}$. Debe tenerse en cuenta que, para llegar de la Ecuación (2.8) a la Ecuación (2.11), se despreció un término y se realizó una linealización, ambas veces bajo la presunción de que $\mu_c N \gg \mu_v v$.

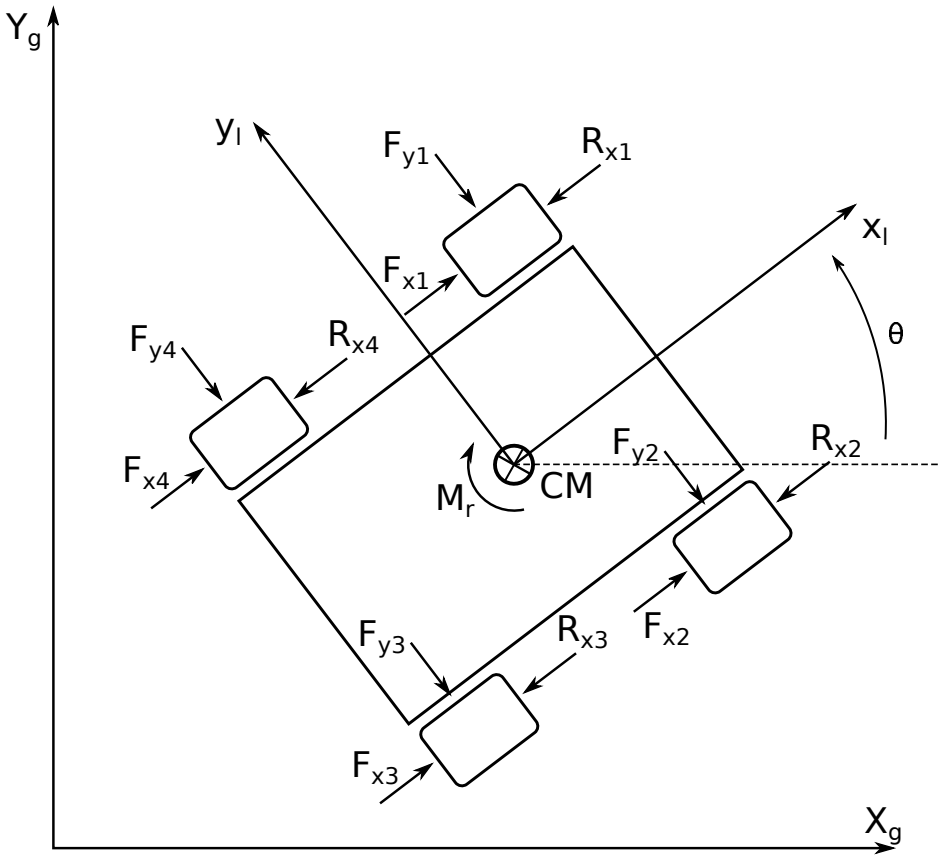


Figura 2.11: Diagrama de las fuerzas que actúan sobre el vehículo.

2.3.2. Hipótesis simplificativas y ecuaciones

Analizando la Figura (2.11), si se desprecian las fuerzas laterales F_{y1-y4} y aplicando la segunda ley de Newton, puede obtenerse las siguientes expresiones para la aceleración lineal y angular:

$$m\ddot{x} = F_{xR} + F_{xL} - R_x \tag{2.12a}$$

$$I\ddot{\theta} = (F_{xR} - F_{xL})c - M_r \tag{2.12b}$$

donde $F_{xL} = F_{x1} + F_{x4}$ y $F_{xR} = F_{x2} + F_{x3}$ y R_x y M_r son la fuerza y torque netos de fricción que actúan sobre el vehículo. Utilizando la aproximación lineal de la Ecuación (2.11) se tiene

$$R_x = NK_x v_x = mgK_x\dot{x}, \tag{2.13a}$$

$$M_r = NK_y v_y = mgK_y\dot{\theta}c. \tag{2.13b}$$

Por lo general, los robots móviles cuentan con un controlador de bajo nivel, encargado de producir la velocidad angular deseada $\omega_{i_{ref}}$ en cada rueda, aplicando un torque τ_i sobre las mismas. Para el caso de un controlador proporcional, la ley de control tendrá la forma $\tau = K_{p_\tau}(\omega_{ref} - \omega)$, donde K_{p_τ} es la ganancia del controlador, lo que deriva en la siguiente expresión [Arbinolo, 2020]:

$$\begin{aligned} \tau_L &= K_{p_\tau}[u_1 - cu_2 - (\dot{x} - c\dot{\theta})] \\ \tau_R &= K_{p_\tau}[u_1 + cu_2 - (\dot{x} + c\dot{\theta})] \end{aligned} \tag{2.14}$$

donde $u_1 = v_{x_{REF}}$ y $u_2 = \omega_{z_{REF}}$. De esta forma, las Ecuaciones (2.12a) y (2.12b) pueden reescribirse como:

$$\begin{aligned} m\ddot{x} &= \frac{1}{r}(\tau_L + \tau_R) - \dot{x}mgK_x \\ &= \frac{1}{r}K_{p_\tau}2(u_1 - \dot{x}) - \dot{x}mgK_x, \end{aligned}$$

$$\begin{aligned} I\ddot{\theta} &= \frac{1}{r}(\tau_R - \tau_L)c - mgK_y\dot{\theta}c \\ &= \frac{1}{r}K_{p_\tau}2(cu_2 - c\dot{\theta})c - mgK_y\dot{\theta}c. \end{aligned}$$

Finalmente, despejando para $\ddot{x}$ y $\ddot{\theta}$:

$$\ddot{x} = -\left(\frac{2K_{p_\tau}}{mr} + gK_x\right)\dot{x} + \frac{2K_{p_\tau}}{mr}u_1, \tag{2.15a}$$

$$\ddot{\theta} = -\left(\frac{2K_{p_\tau}c^2}{Ir} + \frac{mgK_yc}{I}\right)\dot{\theta} + \frac{2K_{p_\tau}c^2}{Ir}u_2. \tag{2.15b}$$

Las Ecuaciones (2.15) pueden llevarse a la forma matricial de espacio de estados, realizando las sustituciones

$$x_1 = x \rightarrow \dot{x_1} = x_2$$
$$x_2 = \dot{x} \rightarrow \dot{x_2} = \ddot{x}$$
$$x_3 = \theta \rightarrow \dot{x_3} = x_4$$
$$x_4 = \dot{\theta} \rightarrow \dot{x_4} = \ddot{\theta}$$

$$\begin{bmatrix} \dot{x_1} \\ \dot{x_2} \\ \dot{x_3} \\ \dot{x_4} \end{bmatrix} = \begin{bmatrix} 0 & 1 & 0 & 0 \\ 0 & -\frac{2K_{p\tau}}{mr} - gK_x & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & -\frac{2K_{p\tau}c^2}{Ir} - \frac{mgK_yc}{I} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix} + \begin{bmatrix} 0 & 0 \\ \frac{2K_{p\tau}}{mr} & 0 \\ 0 & 0 \\ 0 & \frac{2K_{p\tau}c^2}{Ir} \end{bmatrix} \begin{bmatrix} u_1 \\ u_2 \end{bmatrix}, \quad (2.16)$$

$$\begin{bmatrix} y_1 \\ y_2 \\ y_3 \\ y_4 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix} \quad (2.17)$$

Resulta interesante que, en el sistema obtenido, la entrada u_1 afecta sólo a los estados x_1 y x_2, mientras que u_2 afecta sólo a x_3 y x_4, es decir que puede analizarse la dinámica del vehículo mediante dos sistemas desacoplados.

Para obtener este sistema, los autores trabajaron bajo ciertas hipótesis simplificativas, que no deben ser ignoradas si se pretende entender dentro de qué condiciones de entorno y operación puede representar fielmente el comportamiento del robot. A continuación se enumeran las hipótesis más importantes:

1. El vehículo es rígido y su movimiento es planar.

2. El vehículo es simétrico respecto a los planos xy y xz. Por lo tanto, el centro de gravedad yace en la intersección de los mismos y su peso se distribuye equitativamente en cada rueda (es decir, $N_1 = N_2 = N_3 = N_4 = mg/4$

3. Las cuatro ruedas están en contacto constante con el suelo, y cada rueda tiene un único punto de contacto con el suelo.

4. El deslizamiento longitudinal es despreciable.

5. La fuerza que actúa lateralmente sobre las ruedas depende de la carga vertical sobre la rueda.

6. La coordenada longitudinal del centro instantáneo de rotación es constante y nula ($x_{ICR} = 0$).

7. Las ruedas de cada lado, izquierdo y derecho, se mueven a la misma velocidad y reciben el mismo torque por parte del motor (es decir, $\tau_1 = \tau_4 = \tau_L, \tau_2 = \tau_3 = \tau_R$)

8. La fricción viscosa predomina por sobre otras formas de friccion que pueda experimentar el vehículo.

2.3.3. Implementación y resultados en MATLAB®

Dado que el modelo dinámico obtenido es un sistema de ecuaciones de primer orden, puede implementarse en MATLAB® y resolverse con *ode45*, aplicando las estrategias mencionadas en la Sección (2.2.2). Cabe mencionar que la solución del modelo dinámico son los estados de desplazamiento $x_1 = x$, la velocidad lineal $x_2 = \dot{x}$, el desplazamiento angular $x_3 = \theta$ y la velocidad angular $x_4 = \dot{\theta}$, pero nada dice sobre la posición global del robot. Para encontrar la misma, debe resolverse el sistema de ecuaciones diferenciales correspondiente al modelo cinemático (Ecuación (2.6)), utilizando como entradas los estados x_2 y x_4 del modelo dinámico.

En otras palabras, se utiliza el modelo dinámico para obtener la respuesta en velocidad (tanto lineal como angular) del vehículo ante las entradas de referencia $u_1 = v_{x-REF}$ y $u_2 = \omega_{z-REF}$, y se utiliza el modelo cinemático para apreciar cómo varía la pose del vehículo ante dichas velocidades. Por lo tanto, para calcular la respuesta ante una entrada tipo escalón, se debe realizar dos llamadas al *solver ode45*, tal como se observa en el Código (2.6).

```
1 u = [1; 0.5]; % [v_x; omega_z]
2 % Condiciones iniciales nulas
3 q0_din = [0 0 0 0]';
4 q0_cin = [0 0 0]';
5 ts = [0 12.5]; % Intervalo de simulacion
6 xicr = 0;
7
8 [t1,q1] = ode45(@(t,q)ed_dinamico_ssmr(q,u),ts,q0_din);
9 u_cin = [q1(:,2)';q1(:,4)'];
```

```
10 [t,q] = ode45(@(t,q)ed_cinematico_ssmr(t,q,u_cin,t1,xicr),ts,q0_cin);
11
12 % Posicion
13 X = q(:,1);
14 Y = q(:,2);
15 % Orientacion
16 theta = q(:,3);
```

Código 2.6: Simulación del modelo dinámico ante entrada escalón $v_x = 1\ m/s$, $\omega_z = 0.5\ rad/s$.

La Figura (2.12) muestra la respuesta al escalón para el caso particular donde $x_{ICR} = 0$, $v_{x-REF} = 1\ m/s$ y $\omega_{z-REF} = 0.5\ rad/s$. Nuevamente, se espera que el robot describa una trayectoria circular de $v_{x-REF}/\omega_{z-REF} = 2\ m$, sin embargo, se observa que el radio de la trayectoria realizada es ligeramente mayor a $2\ m$. Esto se debe a que el controlador proporcional utilizado no logra anular el error en estado estacionario. En consecuencia, la salida es distinta a la entrada de referencia y, consecuentemente, nunca la alcanza.

En la Figura (2.13) se presentan las curvas de error en velocidad lineal y angular correspondientes a la misma simulación. Aquí puede apreciarse el error en estado estacionario y calcular el radio de la circunferencia que efectivamente describe el robot si ambas velocidades se mantienen constantes, obteniendo un radio de $2.04\ m$, que corresponde a un incremento del $2\,\%$ respecto al valor esperado.

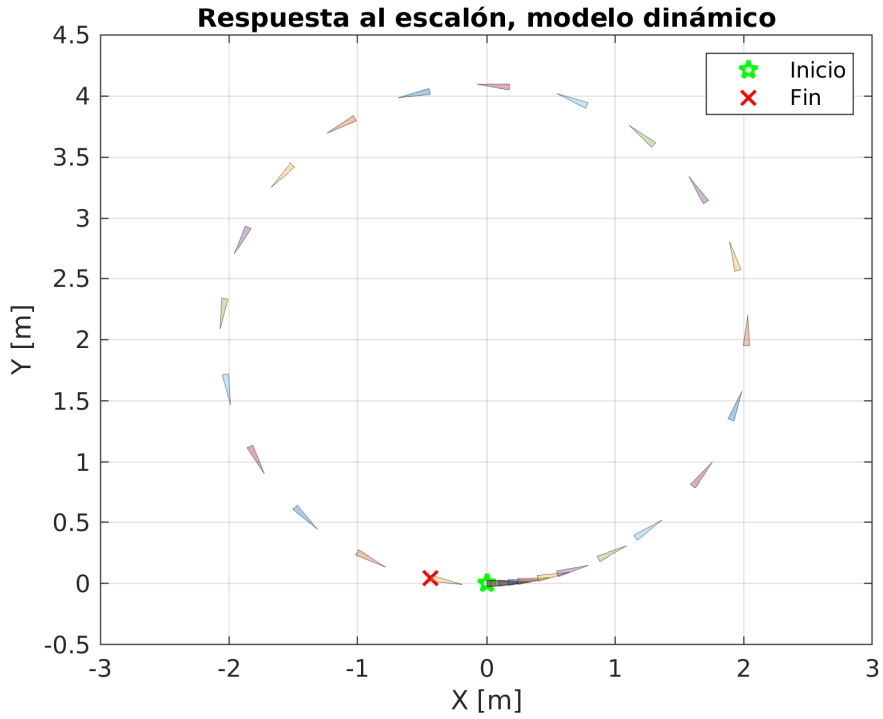


Figura 2.12: Respuesta del modelo dinámico ante entrada escalón: $v_x = 1$,$\omega_z = 0.5$.

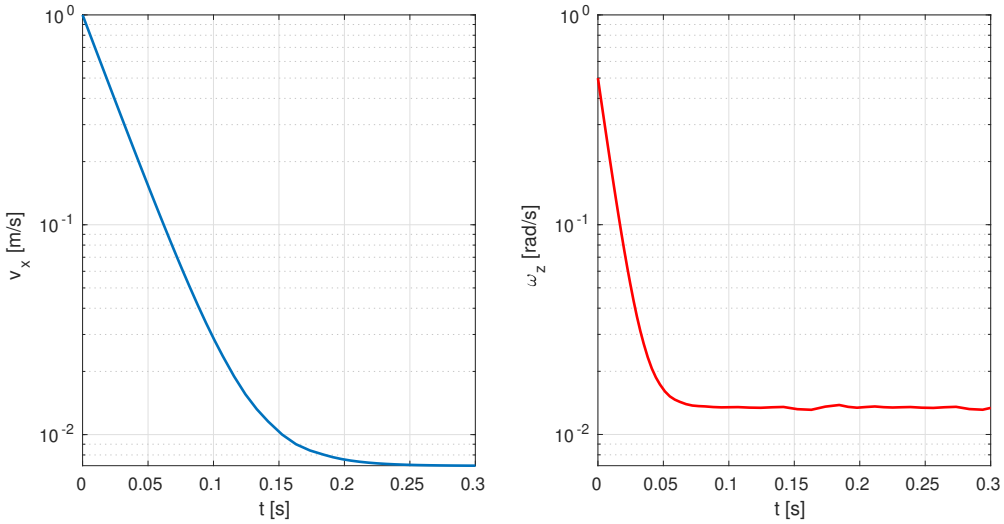


Figura 2.13: Señal de error del control proporcional de velocidad ante entrada escalón $v_{x-REF} = 1$, $\omega_{z-REF} = 0.5$. El error en estado estacionario es de 0.007 m/s y 0.013 rad/s. En azul, velocidad lineal; en rojo, velocidad angular.

Capítulo 3

Entorno de simulación

MATLAB® resulta útil para implementar los modelos matemáticos de interés, simular su desempeño bajo diferentes escenarios, presentar los resultados en forma gráfica y procesarlos para obtener estadísticas y métricas de desempeño. Pero si se busca complejizar el escenario agregando partes móviles, obstáculos o sensores, se hace cada vez más difícil encontrar un modelo mátematico implementable en esta plataforma, entonces resulta conveniente incorporar el uso de simuladores que estimen la dinámica de los cuerpos rígidos en entornos 3D.

Actualmente, existen numerosos softwares de simulación como V-REP [Rohmer et al., 2013], Gazebo [Koenig and Howard, 2004] y Webots [Michel, 2004], que incluyen motor físico, motor gráfico, interacción con sensores y otros aspectos que facilitan la simulación de robots y sus sensores en entornos tridimensionales. Estos simuladores, a través del motor físico, resuelven las ecuaciones que modelan el problema e incluyen las interacciones entre los distintos objetos y el medio, mediante el uso de coeficientes de fricción entre cada superficie. En la Sección (3.2) se explica este concepto en mayor detalle. En particular, para este proyecto, se optó por utilizar el simulador Gazebo (versión 11) [1].

Para comunicar el modelo virtual de Gazebo con los algoritmos de control, sensado y otras funcionalidades, se optó por utilizar ROS (Sistema Operativo para Robots, en inglés: Robot Operating System) [2]. Las principales razones por la que se lo eligió son su alto nivel de abstracción, el hecho de que permite la comunicación transparente entre nodos escritos en Python y C++, y por sobre todo, que la misma estructura utilizada para comunicarse con el modelo simulado puede ser utilizada para comunicarse con el robot físico. La Sección (3.1) presenta las distintas características de ROS.

[1] Sitio oficial de Gazebo: https://www.gazebosim.org/

[2] Sitio oficial de ROS: https://www.ros.org/

A fin de implementar una biblioteca de software que pueda ser utilizada por la comunidad universitaria, se consideró importante elegir Python 3 [3](versión 3.6.9) como lenguaje principal de programación, pues se espera que su popularidad continúe aumentando y no resulte obsoleto próximamente. Esto, sin embargo, presenta la desventaja de que muchas distribuciones de ROS sólo son compatibles con Python 2, mientras que ROS 2 soporta Python 3 de forma nativa. Debido a la abundante documentación sobre ROS 1 disponible actualmente en la red, se optó por utilizarlo (en particular, la versión 1.14 de la distribución ROS Melodic Morenia) en conjunto con la bibilioteca *ROSLibPy* [4], que permite a ROS interactuar con Python 3 mediante el uso de WebSockets.

3.1. ROS

ROS es un conjunto de bibliotecas y herramientas de software de código abierto, que asiste en la integración y comunicación entre distintos dispositivos. Si bien sus siglas lo definen como un sistema operativo, realmente no lo es, pero sí establece un marco definido de trabajo que pretende estandarizar la comunicación entre distintos algoritmos, sensores y dispositivos varios, dentro del mundo de la robótica. La principal característica de ROS es que la comunicación entre distintos dispositivos y/o programas está compuesta por nodos, tópicos, mensajes y servicios.

Un *mensaje* es la información que se desea transmitir, se caracteriza por el dato y el tipo de dato que transmite. Un *nodo* es un ejecutable que puede comunicarse con otros nodos, ya sea suscribiéndose o publicando a un tópico. El *tópico* es el canal por donde se transmiten los mensajes, cada tópico admite mensajes de un único tipo de datos. Así, si se quiere transmitir un mensaje del tipo *Float32* y uno del tipo *Int8MultiArray*, se debe contar con dos tópicos distintos, o crear un nuevo tipo de datos que los combine. La Figura (3.1), obtenida mediante la herramienta *rqt_graph* de ROS, muestra dos nodos (**/teleop_turtle** y **/turtlesim**) que se comunican entre sí mediante el tópico llamado **/turtle1/command_velocity**), la punta de flecha sobre el nodo **/turtlesim** indica que el mismo está suscripto a ese tópico.

La suscripción de un nodo a cierto tópico implica que el nodo recibe y procesa todos y cada uno de los mensajes que se transmiten por dicho tópico. En contraste, un *servicio* permite a un nodo enviar una solicitud y recibir una respuesta del servicio. De modo similar a un tópico, un servicio debe contar con un tipo determinado de datos, con la diferencia que

[3]Sitio oficial de Python: https://www.python.org/

[4]Documentación biblioteca ROSLibPy: https://roslibpy.readthedocs.io/

Figura 3.1: Ejemplo básico de comunicación entre dos nodos mediante tópicos. Imagen obtenida de [Robotics, 2022]

requiere un tipo de datos para la solicitud y uno para la respuesta. Por ejemplo, si un servicio no requiere argumentos de entrada, la solicitud puede ser del tipo *Empty* (del inglés, vacío) y entregar una respuesta del tipo *String*. Al nodo que solicita un mensaje a un servicio se lo llama *cliente* y el nodo que responde la solicitud es el *servidor*.

Gracias a este principio de comunicación estandarizada entre nodos, que pueden estar escritos en distintos lenguajes o correr en diferentes dispositivos, una aplicación desarrollada en un nodo para cierto proyecto puede ser trasladada a un proyecto de características diferentes, o ser complementada con nuevos nodos que amplien su funcionalidad.

3.2. Gazebo

Gazebo se destaca entre otros simuladores por ser software libre. Incluye cuatro motores físicos distintos, ofrece bibliotecas para implementar sensores de todo tipo, plugins para controlar robots, poblar el entorno con distintos elementos y manipularlos. Además, ofrece extensa documentación y tutoriales de distinta complejidad y cuenta con una gran comunidad de usuarios activos, con su propio foro de preguntas y respuestas.

Para correr una simulación, Gazebo requiere la existencia de diferentes archivos y la declaración de ciertas variables de entorno. A destacar, se tienen los archivos de mundo (*world files*, extensión *.world*) y los archivos de modelo (*model files*, extensión *.sdf*), ambos escritos bajo el formato SDF (*Simulation Description Format*), que se basa en el formato XML. Un archivo de mundo contiene todos los elementos de una simulación, incluyendo robots, luces, sensores, objetos estáticos y otros. El Código (3.1) muestra una versión sencilla de mundo, que contiene una fuente de luz (sol) y un suelo.

```
<?xml version="1.0" ?>
<sdf version="1.5">
  <world name="default">
    <!-- A global light source -->
    <include>
      <uri>model://sun</uri>
    </include>
    <!-- A ground plane -->
    <include>
      <uri>model://ground_plane</uri>
    </include>
  </world>
</sdf>
```

Código 3.1: Código SDF que describe un mundo minimalista, con un sol y un suelo. Obtenido de [Robotics, 2013]

Por otro lado, un archivo de modelo consiste en una colección de *links*, *joints*, *collision objects*, *visuals* y *plugins*:

- Un link contiene las propiedades físicas de un cuerpo del modelo. Cada link puede contener elementos de tipo collision, visual, inertial y sensor. Un elemento collision consiste en un cuerpo geométrico y se utiliza para detectar colisiones entre objetos, puede ser una figura geométrica sencilla o una malla 3D. Un elemento visual es utilizado por el motor gráfico para visualizar el modelo. Un elemento inercial describe las propiedades dinámicas del link, como su masa y tensor de inercia. Un elemento sensor recolecta información del mundo simulado, la cual es utilizada por los plugins.

- Un joint conecta dos links entre sí. Se establece una relación *parent-child* en conjunto con otros parámetros, como tipo de unión, ejes de rotación, límites de esfuerzo y coeficientes de fricción de la unión.

- Un plugin es una biblioteca creada para controlar el comportamiento de un modelo. Hay muchos plugins creados por terceros, disponibles para su uso libre, que permiten recibir y procesar la información de sensores, automatizar el movimiento de obstáculos y muchas otras tareas.

Cabe destacar que la carga computacional durante una simulación incrementa a medida que las mallas 3D tienen más nodos, es decir, tienen un mayor detalle. Por otro lado, si se reduce la cantidad de nodos de la malla, las superficies pueden dejar de ser suaves, generando puntos de contacto indeseados y comportamiento anormal. Por esta razón, es buena práctica utilizar los cuerpos geométricas sencillos soportados por el formato SDF (cilindro, esfera y prisma rectangular) para describir los elementos del tipo collision, mientras que para los elementos del tipo visual se pueden utilizar mallas de mayor complejidad sin generar un impacto significativo en el costo computacional.

Al describir los elementos tipo collision, se debe incluir la información correspondiente al coeficiente de fricción de Coulomb para cada material. Si se utiliza el motor físico ODE (*Open Dynamics Engine*), opción por defecto en Gazebo, esta información se explicita en los parámetros *mu* y *mu2*, donde *mu* es el coficiente de fricción de Coulomb para la dirección de fricción principal, y *mu2* es el coeficiente correspondiente a la dirección de fricción secundaria (ortogonal a la primaria). La dirección primaria de fricción (representada por el parámetro *fdir1*) es calculada automáticamente por ODE en cada iteración o, de ser necesario, puede ser especificada manualmente.

Los parámetros *mu* y *mu2* pueden tomar cualquier valor no negativo, con 0 representando una superficie ideal sin fricción, y un valor muy grande representando una superficie con fricción infinita. Cabe destacar que cada objeto de la simulación debe contar con estos parámetros. Cuando dos objetos colisionan, Gazebo elige el *mu* y *mu2* más pequeños entre ambos objetos, y computa las fuerzas resultantes en base a ellos.

Finalmente, los elementos del tipo inertial son esenciales para que Gazebo simule los modelos apropiadamente. Este elemento contiene la información correspondiente a la masa del objeto (en kilogramos) y su matriz de inercia I, reperesentado por los seis elementos del tensor $I_{xx}, I_{xy}, I_{xz}, I_{yy}, I_{yz}, I_{zz}$, donde los subíndices x, y y z hacen referencia a los ejes cartesianos del mismo nombre. El tensor de inercia depende de la masa y de la distribución de la masa del objeto. Para cuerpos geométricos simples, la matriz de inercia puede obtenerse resolviendo las ecuaciones integrales pertinentes o utilizando los valores de referencia disponibles en la bibliografía[5]. Para el caso sólidos irregulares con densidad de masa uniforme, es recomendable obtener un modelo 3D digital (por ejemplo, en un archivo DAE o STL) y calcular su matriz de inercia mediante un software[6].

Los valores asignados a los elementos del tensor de inercia deben ser realistas, de lo

[5]https://en.wikipedia.org/wiki/List_of_moments_of_inertia

[6]https://classic.gazebosim.org/tutorials?tut=inertia&cat=build_robot

contrario, la simulación puede comportarse de manera errática y, en casos extremos, el modelo puede colapsar sin advertencia alguna y los elementos del mismo aparecerán superpuestos en las coordenadas $(0, 0, 0)$, es decir en el origen del mundo. A modo de comparación, vale considerar que los valores $I_{xx} = I_{yy} = I_{zz} = 0.001$ corresponde a un cubo de $0.1\,m$ de lado y $0.6\,kg$ de masa. Por otro lado, un tensor de inercia igual a la matriz unidad puede ser considerado como un cubo de $0.1\,m$ de lado y $600\,kg$ de masa.

3.3. Modelo 3D del vehículo

Mientras que Gazebo utiliza el formato SDF para describir un modelo, ROS utiliza un formato diferente, el URDF (*Unified Robotic Description Format*), también basado en el lenguaje XML. Este formato tiene la desventaja de no contar con cierta información indispensables para la simulación de robots y su entorno en Gazebo, tal como los coeficientes de fricción descriptos en la sección anterior. Para solventar esto, ROS permite añadir esta información en el URDF mediante el uso de la etiqueta <*gazebo*>, que luego unifica con el resto de la descripción y convierte el modelo al formato SDF para que sea cargado por Gazebo.

El formato *xacro* (XML Macros) es un lenguaje de macros XML que permite construir archivos XML más compactos y legibles mediante el uso de macros que, al ser procesadas, se expanden a expresiones XML más extensas. Es ampliamente utilizada en robótica para generar los archivos de descripción de robot, tal como el mencionado URDF, pues además incorpora el uso de constantes, funciones matemáticas, lógica condicional, llamadas a comandos de ROS y soporta el formato YAML (útil para cargar datos desde archivos). Por estas razones, se optó por describir al robot 4WD SS utilizando el formato *xacro*. Por considerar su uso intuitivo y su explicación demasiado extensa como para ser incluída aquí, no se entrará en mayores detalles sobre el mismo[7].

La Figura (3.2) muestra el modelo 3D escrito en formato *xacro*, convertido a URDF para ser cargado por ROS, y enviado a Gazebo con el resto de la información necesaria para completar las propiedades mínimas requeridas en el formato SDF. Se optó por un modelo minimalista, que representa un robot 4WD SS con cuerpo de prisma rectangular y cuatro ruedas idénticas. Las dimensiones, masas y tensores de inercia son similares a las

[7]Para mayor información, consultar https://wiki.ros.org/urdf/Tutorials/Using Xacro to Clean Up a URDF File

del robot Husky[8] de la compañía Clearpath Robotics. En la Figura (3.3) se puede observar, representados en rosa, los elementos inerciales asociados a cada objeto del robot.

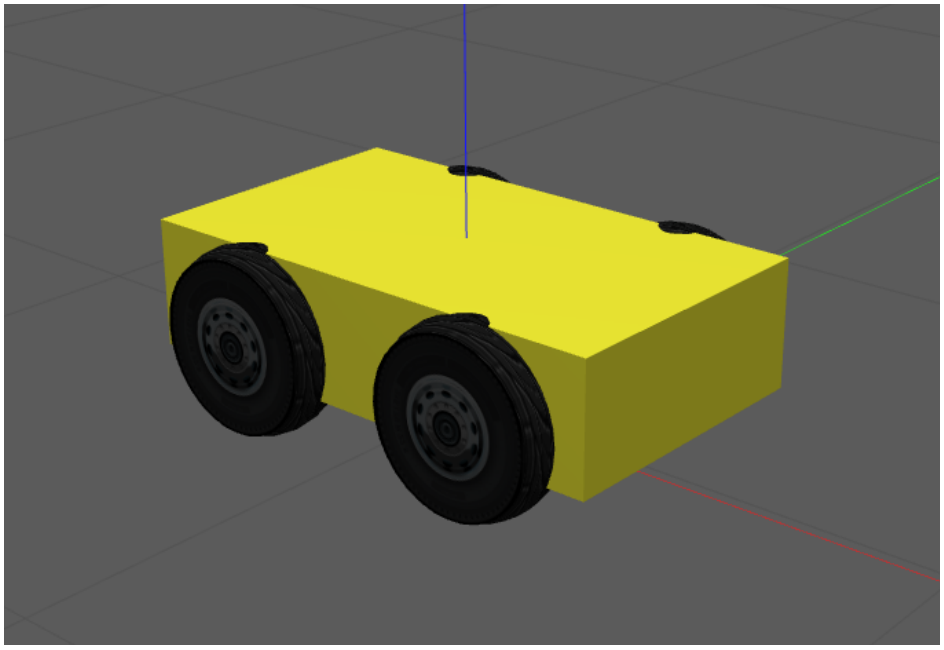


Figura 3.2: Modelo 3D del robot generado con *xacro* y simulado en Gazebo.

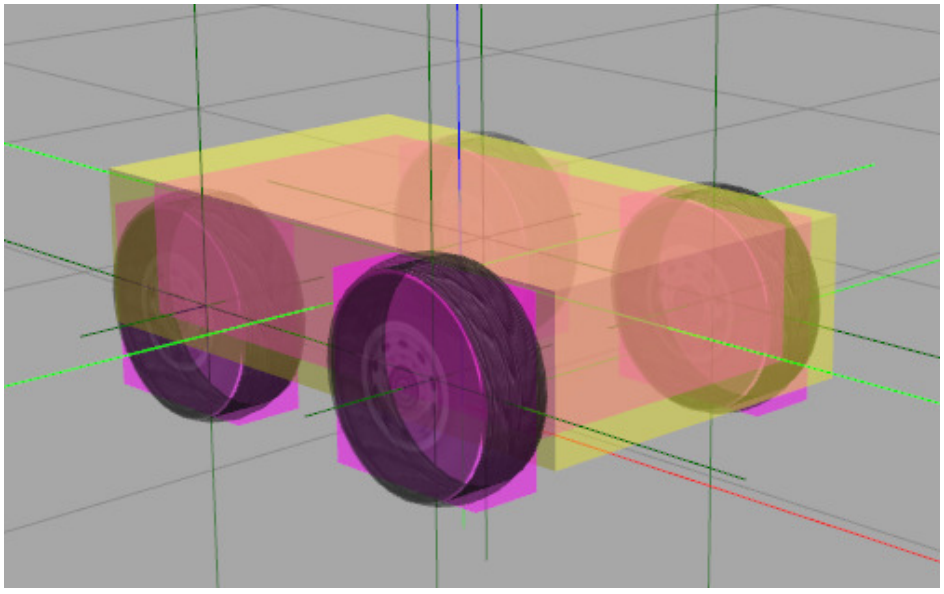


Figura 3.3: Visualización de los elementos de inercia (en rosa) del robot.

Cada rueda cuenta con su malla para la representación visual, presente en la Figura (3.4a). Como elemento collision, se evaluó el comportamiento de la simulación utilizando tanto la misma malla como un simple cilindro. Dado que la malla representa una rueda con perfil semiesférico, sólo presenta un punto de contacto con el suelo en cada instante, como

[8]https://clearpathrobotics.com/husky-unmanned-ground-vehicle-robot/

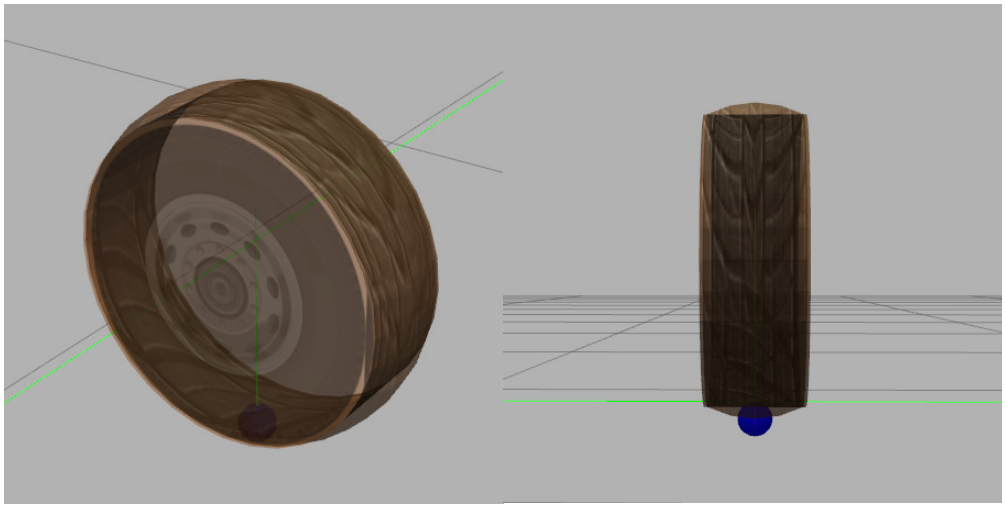
puede observarse en la Figura (3.4b). De esta manera, se asemeja a las características de una rueda ideal, mencionadas previamente.

Por el contrario, cuando la superficie curva de un cilindro entra en contacto con una superficie plana, Gazebo reconoce dos puntos de contacto entre ellos, coincidentes con los extremos de la línea de contacto efectiva. Esto puede observarse en la Figura (3.4c), donde los puntos de contacto son representados por esferas azules, mientras que las líneas verticales verdes representan las fuerzas normales que experimenta el objeto en cada punto.

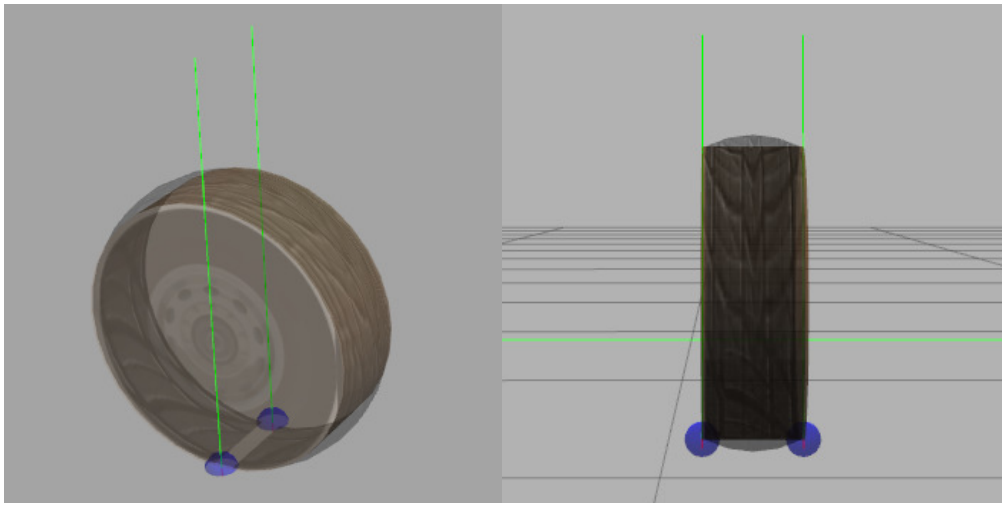
Luego de realizar múltiples simulaciones con ambos tipos de elementos de collision, se puede concluir que el vehículo se comporta de modo similar en ambos casos, pero la velocidad de simulación al utilizar mallas es 70 % más lenta[9] que al utilizar cilindros. Esto es resultado directo de la carga computacional extra que aporta la presencia de las mallas. Por esta razón, se optó por utilizar cilindros para todas las simulaciones plasmadas en este trabajo.

[9]La velocidad de simulación puede variar según el dispositivo en cual corre. El dispositivo utilizado para obtener estos resultados en particular es una *notebook* con procesador Intel Core i7-8750H, GPU NVIDIA GeForce GTX 1060 y 16 Gb de RAM.

(a) fig 1



(b) fig 2



(c) fig 3

Figura 3.4: Modelo 3D de una rueda del robot.

Capítulo 4

Control de velocidad

Este capítulo presenta el modelo de actuador que se utilizará para generar torques en Gazebo y la caja de reducción que acopla el actuador con las ruedas. También se detalla el controlador tipo PID que genera la señal de excitación del actuador, la integración de estos elementos dentro del entorno de simulación y, finalmente, los resultados obtenidos mediante la simulación. Se hizo uso de un complemento informático (*plugin*) que simula un motor eléctrico, pero sus fórmulas fueron modificadas para lograr el comportamiento deseado, por lo que se parte de las ecuaciones clásicas que modelan este tipo de motores y se las desarrolla hasta obtener la fórmula a implementar. Finalmente, el controlador PID está basado en la bibliografía clásica, mas su implementación en lenguaje Python, su ajuste y representación gráfica en un lenguaje estándar es producto de esta tesis.

4.1. Introducción al plugin

Para generar fuerzas y torques en sus modelos, Gazebo cuenta con ciertos actuadores y controladores de distinto tipo. Durante las pruebas iniciales de un modelo resulta conveniente utilizar dichos controladores, pues son lo suficientemente robustos para descartarlos como causas de los errores que puedan surgir durante la simulación, pero, al mismo tiempo, pueden enmascarar otros errores de la misma. En este proyecto se optó por implementar el plugin *dc_motor* [Marton Juhasz, 2020] para Gazebo, que simula un motor eléctrico de corriente continua (CC) a partir de su ecuación diferencial electro-mecánica, cuyos parámetros eléctricos y mecánicos han sido elegidos en semejanza a aquellos de un motor de CC utilizado en robótica (en particular, una silla de ruedas con un motor de $24\,V/450\,W$).

Una vez diseñado el modelo 3D del robot skid-steer, se realizaron las pruebas iniciales para evaluar su comportamiento con los controladores de Gazebo, obteniendo resultados

satisfactorios: el vehículo avanzaba, retrocedía y giraba con velocidad y radio de giro similares a los valores esperados. Luego, se añadió el modelo de los cuatro motores eléctricos, cada uno encargado de impartir su torque a las ruedas del robot.

La implementación inicial de los motores requirió ciertos ajustes en los parámetros físicos del robot, a fin de lograr el comportamiento deseado. En particular, se destaca la necesidad de disminuir los coeficientes de fricción mu y $mu2$ de las ruedas, cuyo valor por defecto $mu = mu2 = 1$ resulta demasiado grande e impide que el robot gire. Esto no ocurría con los controladores de Gazebo ya que su esfuerzo máximo, valor configurable por el usuario, era considerablemente mayor al que puede generar el motor eléctrico. Esto da cuenta de lo valioso que es contar con un modelo de simulación de un motor real. Realizando simulaciones con distintos valores se determinó que la elección $mu = mu2 = 0.2$ permite girar correctamente, sin que se aprecien efectos secundarios indeseados. Sin embargo, lo ideal es ajustar estos valores para que sean idénticos a los del robot físico que se desee simular.

El plugin del motor de CC hace uso de tres topics de ROS:

- Voltaje de excitación del motor, normalizado respecto al voltaje nominal (por defecto, 24 V) *(entrada)*. Rango: $[-1; 1]$.
- Velocidad angular *(salida)*. Unidades: rad/s.
- Corriente del motor *(salida)*. Unidades: A.

4.1.1. Modificaciones al plugin

Para comprobar el funcionamiento del plugin *dc_motor* se utilizó un escenario de prueba que consiste en una única rueda, cuyo eje de rotación se encuentra lo suficientemente elevado para garantizar que la rueda no entre en contacto con el suelo, como puede observarse en la Figura (4.1). Por requisitos del software de simulación, el eje debe estar asociado y anclado a un objeto, por lo que se instanció una base rectangular rígidamente fijada al suelo que garantiza que el eje no se desplace ni experimente vibraciones. Este escenario puede utilizarse tanto para comprobar el correcto funcionamiento del plugin como para obtener la respuesta a lazo abierto del motor con carga ante diferentes señales de comando o variaciones en los parámetros físicos de la rueda o del motor, por ejemplo.

Luego de varias pruebas, se determinó que el plugin no otorgaba los mismos valores de corriente, torque y velocidad que aquellos obtenidos al resolver las ecuaciones diferenciales que, acorde a la documentación, implementa. Por esta razón, se optó por partir de un modelo

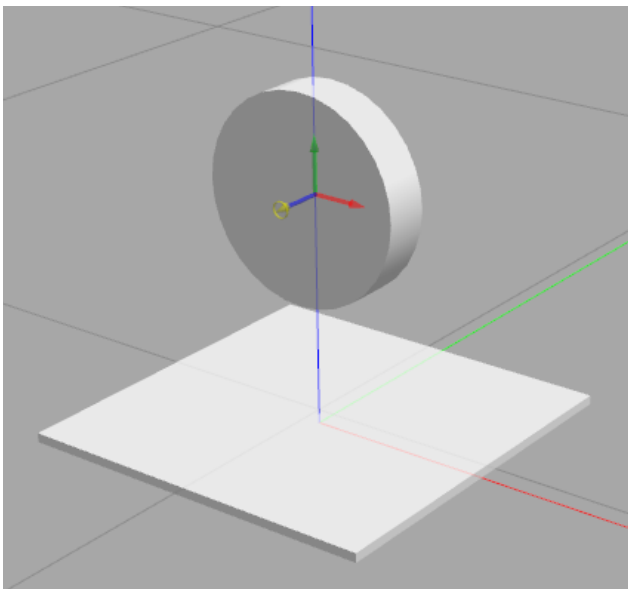


Figura 4.1: Escenario de prueba del plugin *dc_motor*. Consisten en una base fijada al suelo y una rueda suspendida en el aire. El círculo amarillo señala la flecha azul como su eje de rotación.

clásico de motores de corriente continua, discretizar sus ecuaciones y reemplazar con la versión discreta las ecuaciones originales del plugin. El modelo clásico de segundo orden de un motor CC controlado por inducido [Dorf and Bishop, 2008] puede representarse con el diagrama de bloques de la Figura (4.2) o las funciones de transferencia de la Ecuación (4.1).

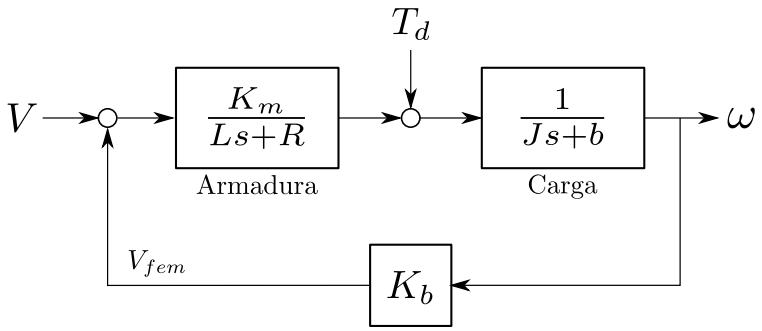


Figura 4.2: Diagrama de bloques de un motor de CC controlado por inducido.

$$\begin{aligned} \omega(s) &= G_1(s)V(s) + G_2(s)T_d(s), \\ G_1(s) &= \frac{K_m}{(Ls+R)(Js+b)+K_bK_m}, \\ G_2(s) &= -\frac{Ls+R}{(Ls+R)(Js+b)+K_bK_m}, \end{aligned} \tag{4.1}$$

donde V es el voltaje de excitación, T_d representa el torque externo sobre la carga (en este caso, la rueda), K_b es la constante de fuerza electromotriz, K_m es la constante del motor, R y L son la resistencia eléctrica e inductancia del bobinado, J es el momento de inercia del rotor y b es es la fricción del rotor. Dada la frecuencia de muestreo del simulador ($1\,kHz$),

los efectos de polo eléctrico ($s_1 = -R/L$) pueden ser despreciados, obteniendo el siguiente sistema de primer orden:

$$G_1(s) = \frac{K_m}{R(Js+b) + K_b K_m},$$
$$G_2(s) = -\frac{R}{R(Js+b) + K_b K_m}.$$

Aplicando un muestreo de matenedor de orden cero (zoh: *zero order hold*, en inglés) de paso h [Åström and Wittenmark, 2013], se obtiene el sistema discretizado descrito por los siguientes operadores de transferencia:

$$\omega(k) = g_1(q)V(k) + g_2(q)T_d(k), \tag{4.2a}$$

$$g_1(q) = k_1\tau\frac{1 - e^{-h/\tau}}{q - e^{-h/\tau}}, \tag{4.2b}$$

$$g_2(q) = -\frac{\tau}{J}\frac{1 - e^{-h/\tau}}{q - e^{-h/\tau}}, \tag{4.2c}$$

donde q es el operador de desplazamiento hacia adelante, tal que $q\,\omega(k) = \omega(k+1)$, $\tau = RJ/(Rb + K_b K_m)$ y $k_1 = K_m/RJ$. Finalmente, introduciendo la constante $E = e^{-h/\tau}$, reemplazando (4.2b) y (4.2c) en (4.2a), aplicando los desplazamientos temporales correspondientes y despejando para $\omega(k+1)$, se obtiene el modelo discretizado:

$$\omega(k+1) = E\,\omega(k) + \tau\,(1-E)\,\left(k_1 V(k) - \frac{T_d(k)}{J}\right). \tag{4.3}$$

Sin embargo, luego de implementar la ecuación (4.3) en el plugin y evaluar su desempeño en el escenario de prueba, se encontró que los valores obtenidos tampoco se correspondían con aquellos obtenidos al resolver las ecuaciones diferenciales. Aún más, no solo los valores diferían, sino que el comportamiento ante variaciones en los parámetros contradecía las ecuaciones. Por ejemplo, al aumentar el momento de inercia en dos órdenes de magnitud, se espera que la respuesta del sistema sea, aproximadamente, 100 veces más lento. Por el contrario, en la simulación se obtenía una variación casi imperceptible en la velocidad del sistema.

Este fenómeno da pauta que el inconveniente no se debía a la implementación de las ecuaciones discretas del motor, por lo que se continuó analizando las posibles causas. Finalmente, se descubrió que se trataba de un error en la interpretación del rol que cumple Gazebo dentro de la simulación. Como puede verse en la Figura (4.2), el modelo del motor CC controlado por inducido cuenta con dos bloques principales, la parte eléctrica (o armadura) y la parte

mecánica (o carga). Si bien las ecuaciones desarrolladas previamente son totálmente válidas, intentan describir la parte mecánica mediante los parámetros físicos del rotor y su carga, así como su dinámica entre torque y velocidad angular. Sin embargo, esta tarea es precisamente la que realiza Gazebo para cada objeto, en cada iteración de la simulación.

Por lo tanto, el plugin sólo debe calcular el torque que se aplicará a la carga en función del voltaje de excitación y la velocidad angular del rotor, mientras que Gazebo aplicará dicho torque y devolverá la velocidad angular correspondiente, en función de todos los torques actuantes sobre la carga y su matriz de inercia. Aún más, dado que la constante de tiempo asociada al polo eléctrico es mucho más pequeña que el periodo de ejecución del plugin, su dinámica puede ser despreciada, por lo que el torque se calcula a partir de la ganancia en estado estacionario de la planta eléctrica. Dado que el polo eléctrico está en el semiplano izquierdo de s, esta ganancia puede obtenerse utilizando el teorema del valor final (TVF) como sigue:

$$K_{ss} = \lim_{s \to 0} s \frac{K_m}{Ls + R} \frac{1}{s},$$

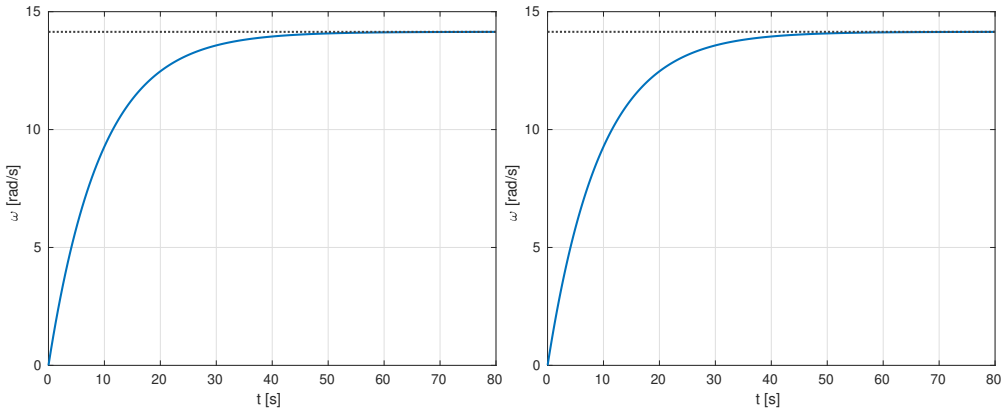
$$K_{ss} = \frac{K_m}{R}.$$

Así, la operación del plugin se reduce a la siguiente cuenta algebraica:

$$T(k) = \frac{K_m}{R}(V(k) - K_b \omega(k)). \tag{4.5}$$

Una vez implementada la Ecuación (4.5) en el plugin, se simuló la respuesta a una entrada escalón en el escenario de prueba, asignando la matriz de inercia J_{rueda} a la rueda y la constante de fricción b al *joint* del motor. Estos mismos parámetros fueron cargados a MATLAB® para resolver la Ecuación (4.1) ante una entrada escalón de la misma magnitud. Los resultados obtenidos para cada caso se muestran en la Figura (4.3), mientras que la Figura (4.4) muestra el error relativo entre ambas simulaciones. Nótese que, para el cálculo del error relativo, se han excluído los puntos donde la señal de referencia era nula.

Resulta evidente que, con esta modificación del plugin, la simulación se corresponde fielmente con el modelo clásico de un motor CC. Por lo tanto, esta nueva versión será la que se utilice para todas las simulaciones presentadas en este proyecto.



(a) (b)

Figura 4.3: Respuesta del motor a entrada escalón simulada en: (a) MATLAB® , y (b) Gazebo.

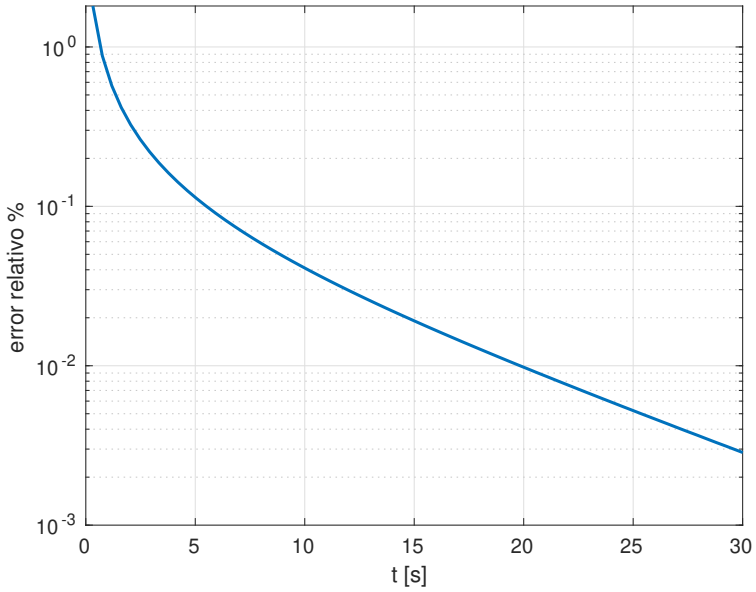


Figura 4.4: Error relativo porcentual entre las respuesta al escalón simuladas en MATLAB® y Gazebo.

4.1.2. Caja de reducción

Los motores CC convencionales suelen tener una velocidad nominal de giro demasiado alta para ser útil en aplicaciones de robótica. Para solventar esto suele utilizarse una caja de reducción, que acopla el rotor a la carga de modo que la velocidad de salida (carga) es n veces menor a la de entrada (rotor), mientras que lo opuesto ocurre con el torque. De este modo se cumple

$$\omega_{carga} = \omega_{rotor}/n,$$

$$\tau_{carga} = n\,\tau_{rotor}.$$

Al mismo tiempo, reduce la inercia de carga J_{carga} que se refleja sobre el motor por un factor de n^2, es decir $J_{carga/rotor} = J_{carga}/n^2$.

El plugin emula una caja de reducción ideal, cuyo factor de reducción puede ser configurado junto al resto de los parámetros físicos y eléctricos del motor. Para este proyecto se escogió un factor $n = 100$, que reduce 10 mil veces los efectos de la carga sobre el motor. Esto permite realizar el ajuste de los controladores de las rueda sin requerir que esten montadas sobre el robot. Luego, al montarlas sobre el mismo, sólo habrá que hacer leves ajustes. En contraste, si se siguiera este procedimiento sin la caja de reducción, el ajuste inicial de los controladores resultaría superfluo, pues las interacciones entre las otras ruedas y la estructura del robot pueden modificar sustancialmente la planta que se busca controlar, siendo necesario realizar un nuevo ajuste sobre este sistema más complejo.

Para el caso de una rueda en suspensión, la respuesta al escalón al implementar la caja de reducción puede observarse en la Figura (4.5), nótese que la velocidad angular graficada corresponde a la salida de la caja de reducción. Así, para la misma carga, la caja permite disminuir el tiempo de asentamiento en más de 10 veces y la velocidad angular máxima en 50 veces.

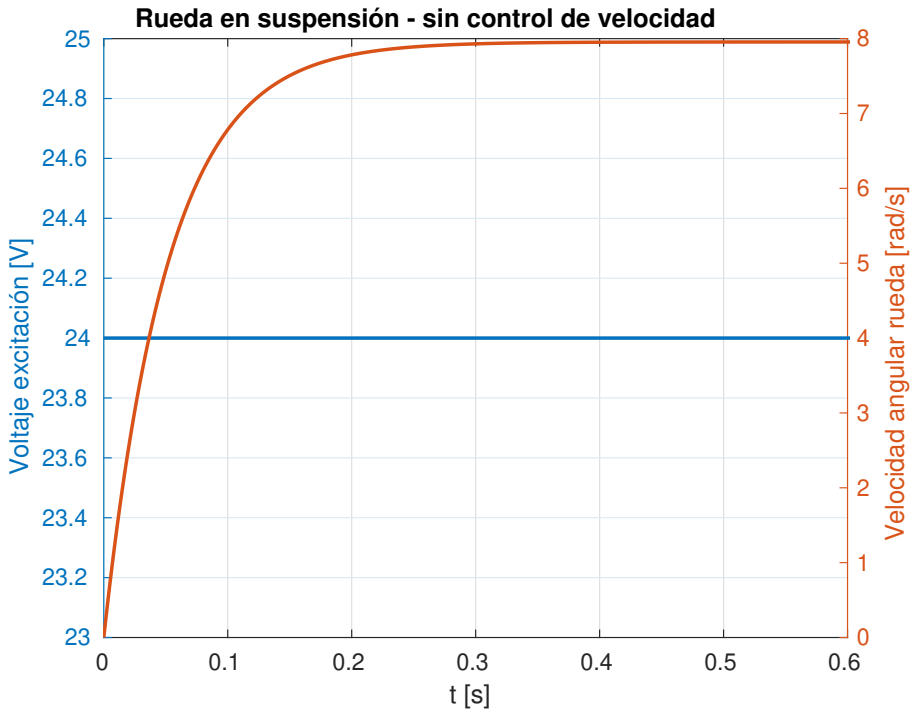


Figura 4.5: Velocidad de salida de la caja de reducción, ante entrada escalón al motor.

Una vez aplicadas las modificaciones del plugin y comprobado su correcto funcionamiento, se realizó un *fork* del repositorio del plugin original y se aplicaron los cambios aquí descritos. Esencialmente, hacer un *fork* es generar una copia exacta de un repositorio A y almacemarla

en nuevo repositorio B, al cual se tendrá acceso total. Uno de sus usos es el de permitir contribuir a un proyecto ajeno de forma segura, es decir que se pueden realizar modificaciones en el repositorio B y ofrecer al autor de A el replicar dichas modificaciones en su propio repositorio.

4.2. Implementación del controlador en Python - ROS

Habiendo presentado el plugin que simula el motor eléctrico, surge la necesidad de implementar un controlador de velocidad que reciba una velocidad de referencia y envíe la señal de voltaje apropiada al motor, asegurando que su velocidad sea, idealmente, igual a la de referencia. En este trabajo se optó por implementar un controlador PID discreto: la parte integral fue discretizada por *diferencias finitas hacia atras* y la parte derivadora por *aproximación de Tustin* [Åström and Wittenmark, 2013], obteniendo la forma:

$$\begin{aligned} U_k &= P_k + I_k + D_k, \\ P_k &= K_p(r_k - y_k), \\ I_k &= I_{k-1} + K_p K_i T_s e_k, \\ D_k &= \frac{K_d}{K_d + NT_s} D_{k-1} - \frac{K_p K_d N}{K_d + NT_s}(y_k - y_{k-1}), \end{aligned} \tag{4.6}$$

donde r_k, y_k son el valor de la señal de referencia y de salida, respectivamente, para el instante k; T_s es el tiempo de muestreo del controlador; y N es una constante que limita la ganancia de la parte derivadora a altas frecuencias, generalmente toma un valor entre 3 y 20. También se implementa una lógica de *clamping*, que detiene la integración del error cuando la salida del controlador se satura para evitar el fenómeno conocido como *wind-up*. En particular, la estrategia utilizada se denomina *dynamic clamping* y consiste en calcular en cada paso los limites de saturación de la parte integradora, de forma tal que ésta cambie de valor sólo cuando, al hacerlo, la salida permanezca dentro del rango de operación del actuador [Wilson, 2013].

La estructura del PID presentada en la Ecuación (**??**) es genérica y puede ser utilizada para controlar distintos procesos. Dado que Python soporta la programación orientada a objetos (POO), es posible crear la clase *PID* con los atributos y métodos necesarios para implementar las ecuaciones mencionadas. Así, puede instanciarse un objeto de la clase *PID* para controlar la velocidad de cada rueda y otro objeto para controlar la posición del robot, por ejemplo. En este trabajo, a fin de presentar las distintas clases que componen la biblioteca desarrollada, se utilizará Universal Modeling Language (UML).

UML es un lenguaje estándar de modelado de sistemas orientados a objetos que cuenta con distintos tipos de diagramas y simbología para representar tanto la estructura como la interacción entre objetos. En UML, una clase se representa como una *caja* con tres compartimentos: el superior indica el nombre de la clase, en letra negrilla y centrada; en el medio se presentan los atributos de la clase y, si corresponde, sus valores por defecto precedidos por dos puntos (:); finalmente, en el inferior se ubican los métodos asociados a la clase, sus parámetros de entrada entre paréntesis, seguidos por dos puntos y el tipo de valor que devuelven.

La Figura (4.6) muestra la representación en este lenguaje para la clase *PID*. El método *clear()* reinicia las variables del PID, mientras que *update(feedback_value)* actualiza el valor de la salida *output* del controlador en base al último valor de *feedback_value*.

PID
kp : Float32
ki : Float32
kd : Float32
ts : Float32
limit_output : Float32
output : Float32
clear() : void
update(feedback_value) : void

Figura 4.6: Diagrama de la clase *PID* implementada.

Por otro lado, la clase *Rueda* cuenta con el método *load_pid_controller(kp,ki,kd,ts)*, encargado de instanciar un objeto *PID* asociado al objeto *Rueda*, y *run_pid()* que actualiza la salida del controlador y envía el comando al topic pertinente de ROS. La Figura (4.7) muestra el diagrama de la clase.

Finalmente, al ser instanciada, la clase *Vel_controller()* crea cuatro objetos *Rueda*. Su método *run_pid()* inicia los controladores PID de cada rueda y actualiza sus valores periódicamente, hasta que el método *stop_pid()* sea llamado y se anulen los controladores por completo. Las velocidades de referencia que el controlador asigna para cada rueda son enviadas al topic *setVelocity_topic* en un único vector de dimensión 4, cuyos elementos deben ser del tipo *Float32*. La Figura (4.8) muestra el diagrama de la clase. Este paradigma de progra-

Rueda
motor_name : String velocity_topic : ROSTopic command_topic : ROSTopic
load_pid_controller(kp,ki,kd,ts) : void run_pid() : void

Figura 4.7: Diagrama de la clase *Rueda* implementada.

mación apunta a la modularización del código, permitiendo añadir nuevos controladores tan solo creando una nueva clase y agregando lós métodos correspondientes para instanciarlo en una *Rueda* y para actualizar su valor de salida.

Vel_controller
front_left_wheel : Rueda front_right_wheel : Rueda rear_left_wheel : Rueda rear_right_wheel : Rueda setVelocity_topic : ROSTopic
run_pid() : void stop_pid() : void

Figura 4.8: Diagrama de la clase *Vel_controller* implementada.

Una vez implementado el PID, se debe realizar el ajuste de sus ganancias a fin de obtener el comportamiento deseado. En particular, para esta configuración se decidió ajustar el controlador de forma que la respuesta al escalón sea sobreamortiguada, es decir que no presente sobreimpulso ni oscilaciones, que el tiempo de asentamiento sea al menos un 70 % más pequeño que el del sistema no controlado y que la señal de control no sature para una velocidad de referencia de 1 rad/s.

4.3. Resultados

Las siguientes figuras muestran la respuesta en velocidad angular de las ruedas ante una señal de referencia $\omega_{rueda_{ref}} = 1\, rad/s$ para cada rueda. En particular, la Figura (4.9) se obtuvo al utilizar motores únicamente en las ruedas delanteras mientras que la Figura (4.10) se obtiene utilizar un motor por cada rueda. Sin pérdida de generalidad, para facilitar la interpretación de las gráficas, en ambas figuras sólo se presenta la información correspondiente a la rueda delantera izquierda.

Comparando ambas figuras, puede notarse que la señal de excitación alcanza aproximadamente el mismo valor máximo mientras que, al usar cuatro actuadores, el valor en estado estacionario disminuye alrededor de un 45 % y el tiempo de asentamiento disminuye a la mitad. Esto es esperable, pues el esfuerzo de mover la carga se distribuye entre los distintos actuadores.

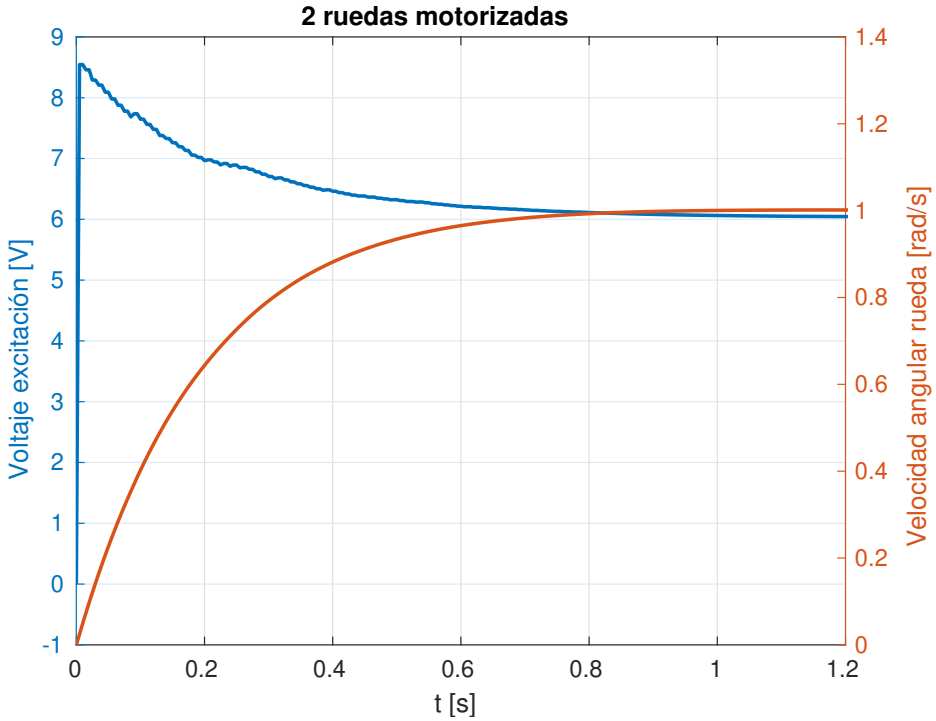


Figura 4.9: Respuesta de la rueda delantera izquierda del vehículo, ante señal de referencia $\omega_{ref} = 1\, rad/s$, actuando sólo sobre las ruedas delanteras.

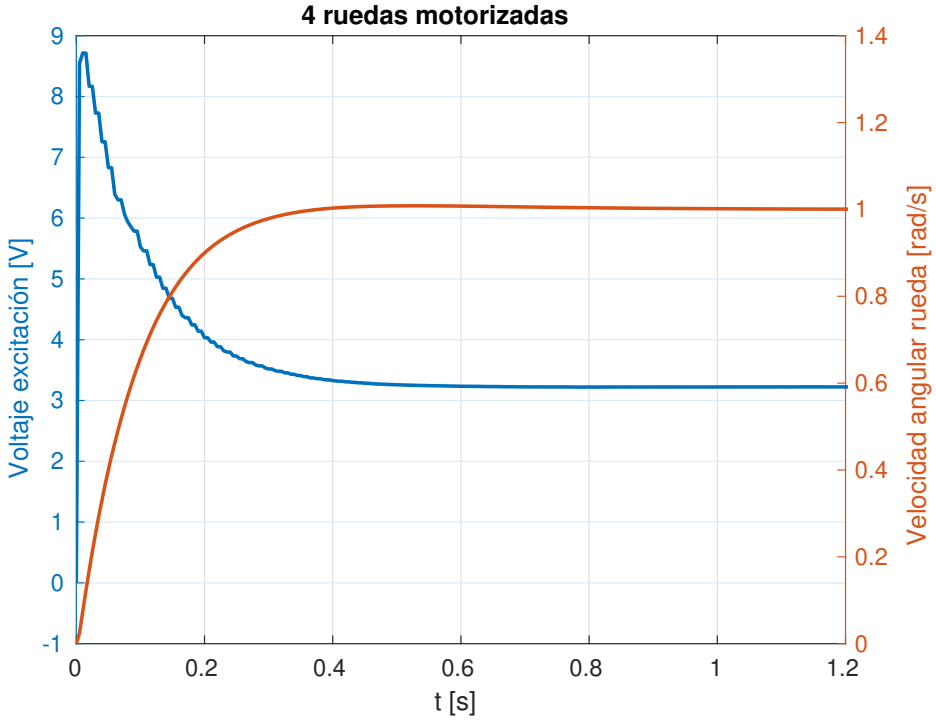


Figura 4.10: Respuesta de la rueda delantera izquierda del vehículo, ante señal de referencia $\omega_{ref} = 1\, rad/s$, actuando sobre todas las ruedas.

Capítulo 5

Control de posición

En este capítulo se presentan las variables geométricas que se usarán para calcular el error de pose del vehículo. Luego se describen las estructuras de los controladores implementados, tanto en las simulaciones de MATLAB® como en aquellas de Gazebo, y se establecen las métricas de desempeño según la cual se compararán los resultados obtenidos.

Las variables geométricas y la estructura general de los controladores PID son las propuestas en [Arbinolo, 2020], con incorporaciones propias de esta tesis, como la utilización de *dynamic clamping*, las restricciones aplicadas al movimiento del vehículo para mejorar su comportamiento y la implementación de código en el entorno de MATLAB® y Python. Finalmente, las métricas de desempeño fueron seleccionadas en base al trabajo realizado en [Ermacora et al., 2020].

5.1. Evaluación del error de posición y orientación

Para controlar la posición del vehículo, primero resulta necesario establecer qué variables son las que se quiere realimentar y cómo actuarán estas sobre cada entrada. Existen varias estrategias posibles de control, pero aquí se trabajará introduciendo tres variables, cada una de ellas relacionada con un tipo de error de pose. La Figura (5.1) representa graficamente las tres variables, que se definen de la siguiente forma:

- ρ es la distancia euclidiana entre el centro de masa del vehículo y el siguiente punto de ruta, WP_n. Es decir, $\rho = \sqrt{\Delta X^2 + \Delta Y^2}$. Indica cuánto debe desplazarse el vehículo, asumiendo el camino más corto, para llegar al punto objetivo.

- Δy es la distancia latitudinal entre el vehículo y el siguiente punto de ruta WP_n. No debe ser confundida con ΔY, asociada al eje global Y. Indica cuánto debe girar el

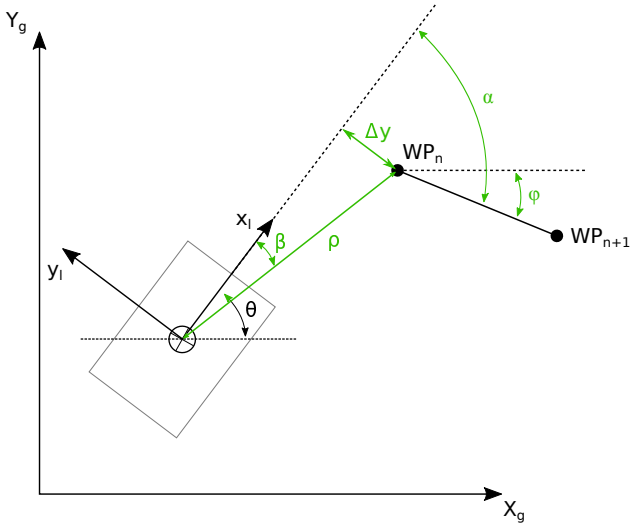


Figura 5.1: Variables de error obtenidas para cada punto de ruta WP_n, en relación al siguiente WP_{n+1} y la posición actual.

vehículo para alinearse en dirección al punto objetivo.

- α es el ángulo formado entre el eje local x y el segmento que une el siguiente punto de ruta WP_n con WP_{n+1}. Indica cuánto deberá girar el vehículo para alinearse en dirección al futuro punto objetivo.

Puede obtenerse α y Δy a partir de las siguientes ecuaciones:

$$\begin{aligned} \beta &= \arctan\left(\frac{\Delta Y}{\Delta X}\right) - \theta, \\ \Delta y &= \rho \ \sin(\beta), \\ \alpha &= \theta - \phi. \end{aligned} \tag{5.1}$$

Es importante notar que el cálculo de β se realizará utilizando la función arcotangente de dos parámetros, usualmente representada como $atan2(y, x)$, que calcula el ángulo formado por el vector (x, y) y el eje x positivo, dentro del intervalo $[-\pi, \pi]$. No debe ser confundida con la funcion $atan(y/x)$, cuya imagen es $[-\pi/2, \pi/2]$. Por otro lado, al calcular α se le aplica también un control condicional, tal que su valor se anule cuando la distancia al punto objetivo sea mayor a un determinado umbral ρ_α, de modo que el aporte de α sólo influya en la cercanía del punto objetivo actual. El controlador considera que ha alcanzado el punto objetivo WP_n cuando se encuentre dentro de cierta distancia ρ_{min} al mismo, momento en el cual asignará el siguiente punto WP_{n+1} como nuevo objetivo. Los valores de ρ_{min} y de ρ_α deben ser ajustados manualmente para obtener el desempeño adecuado, en función de los márgenes de error del sistema de localización y de la precisión deseada.

Tomando la velocidad lineal v_x y la velocidad angular ω_z como variables de control de posición del robot, se puede encontrar una ley de control que genere un valor de referencia en función al error de pose. Dado que ρ indica la distancia que el robot debe recorrer hasta el siguiente punto, mientras que Δy y α dan cuenta del desplazamiento angular requerido, luego la ley de control puede tener la siguiente forma

$$v_{x_{REF}} = f_1(\rho),$$
$$\omega_{z_{REF}} = f_2(\Delta y, \alpha).$$

En particular, en este trabajo se optó por utilizar tres controladores del tipo PID para controlar cada una de estas variables, respectivamente nombrados $PID-\rho$, $PID-\Delta y$ y $PID-\alpha$, con la misma estructura que aquellos presentados en la Ecuación (**??**). Luego, $v_{x_{REF}}$ consiste en la salida de $PID-\rho$ y $\omega_{z_{REF}}$ consiste en la suma de las salidas de $PID-\Delta y$ y $PID-\alpha$. Si bien cada PID cuenta con su limitador y con lógica anti-windup, también es necesario limitar la suma de estos dos PID, a fin de que $\omega_{z_{REF}}$ se encuentre siempre dentro del rango deseado.

A su vez, estas señales de referencia son convertidos a las velocidades angulares que debe tomar cada rueda, $\omega_{L_{REF}}$ para las ruedas de la izquierda y $\omega_{R_{REF}}$ para las de la derecha, que pueden despejarse de la Ecuación (2.3), obteniendo la siguiente relación

$$\omega_{L_{REF}} = (v_{x_{REF}} - c\,\omega_{z_{REF}})/r,$$
$$\omega_{R_{REF}} = (v_{x_{REF}} + c\,\omega_{z_{REF}})/r.$$

La velocidad angular máxima alcanzable por las ruedas dependerá, entre otros factores, de las prestaciones del motor. Resulta interesante encontrar el rango de valores que $v_{x_{REF}}$ y $\omega_{z_{REF}}$ pueden tomar simultáneamente, de modo tal que las velocidades de las ruedas se mantengan en el rango $[-\omega_{L,R_{max}}, \omega_{L,R_{max}}]$, para garantizar que los valores de referencia sean alcanzables. No existe un modo único de establecer dicho rango, por lo que se decidió establecer una restricción extra: se priorizará siempre la orden de giro del vehículo por sobre su desplazamiento lineal. En otras palabras, se busca minimizar el radio de giro del vehículo $v_{x_{REF}}/\omega_{z_{REF}}$, para permitirle realizar curvas lo más cerradas posible, en sacrificio de la velocidad con que las realiza.

De esta forma se obtienen las siguientes relaciones, que fueron incorporadas dentro de la lógica de control de posición:

$$max(|\omega_{z_{REF}}|) = \frac{r}{c}\omega_{L,R_{max}}$$
$$max(|v_{x_{REF}}|) = r\,\omega_{L,R_{max}} - c\,|\omega_{z_{REF}}|$$

El controlador implementado acepta cualquier conjunto de datos de coordenadas $WP_n = (X_n, Y_n)$ que indiquen los puntos de ruta a seguir, sin analizar si el trayecto es realizable o no. Para las simulaciones de prueba, se han utilizado las siguientes geometrías de camino:

- Cuadrado
- Círculo
- Figura de ocho

Es de esperar que las geometrías con curvas suaves sean más realizables que aquellas no suaves, como el cuadrado con sus esquinas. Sin embargo, es de interés observar cómo se comporta el robot ante estas situaciones. Con el mismo espíritu, al utilizar las geometrías circulares el robot parte siempre desde el centro del círculo, por lo que primero deberá acercarse al borde para luego continuar por el mismo.

5.2. Métrica de desempeño

A fin de cuantificar el desempeño de un controlador de posición en seguimiento de caminos, se decidió implementar una rutina en MATLAB® que calcula la distancia euclidiana hasta el camino de referencia para cada punto del camino realizado por el robot. Como métricas de desempeño se utilizan las siguientes medidas de tendencia y de dispersión de las distancias euclidianas: media d_{mean}, valor máximo d_{max} y desvío estándar d_{std}. La importancia que tenga cada métrica para el usuario dependerá de la aplicación particular: en algunos casos se buscará minimizar la media del error, mientras que en otros casos se buscará acotar el valor máximo sin importar tanto el error promedio.

Los parámetros de entrada de la función implementada son las coordenadas $(X_k, Y_k), k = 1, ..., K$, correspondientes a las posiciones que tomó el robot durante la simulación, y las coordenadas objetivo $WP_n, n = 1, ..., N$. Considerando que N es un valor fijo para cada camino objetivo, mientras que K depende de la cantidad de mediciones obtenidas durante la simulación, suele darse que $K \gg N$. Dado que la función calcula la distancia euclidiana de un punto a otro, resulta conveniente sobre-muestrear el camino objetivo de modo que

$K \ll N$. La Figura (5.2) utiliza las distancias calculadas para dibujar un círculo en torno a cada punto, con radio igual a la distancia correspondiente al mismo.

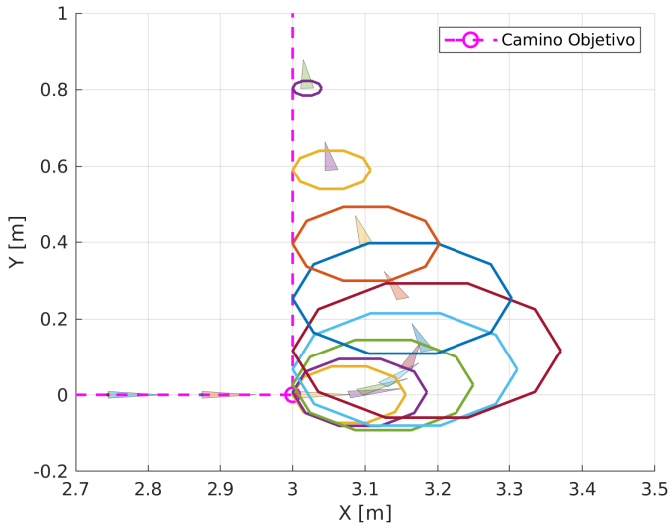


Figura 5.2: Representación gráfica de la distancia euclidiana entre la posición del robot y el punto más cercano del camino objetivo.

También se implementó el cálculo de otra métrica d_{ratio}, que se obtiene como la razón entre la longitud del camino recorrido por el robot y el camino objetivo:

$$d_{ratio} = \frac{d_{real}}{d_{ref}}.$$

Así, en el caso ideal que el robot siguiera el camino sin error alguno se tendría $d_{ratio} = 1$. En general, para otros casos, el valor de la métrica se alejará de la unidad.

5.3. Implementación en MATLAB®

Para no entorpecer la continuidad de la lectura, no se detallará la implementación de la lógica del control de posicion en MATLAB® . A destacar, resulta necesario realizar ciertos cambios respecto a lo realizado en la Sección (2.3.3) en el modo de utilización del *solver ode45*, a fin de incorporar la dinámica del controlador. El cambio radica en que ahora se utilizará el *solver* para encontrar la solución a las EDOs, tanto del modelo dinámico como del cinemático, sólo para el siguiente paso temporal T_S, que coincide con el tiempo de muestreo del controlador. Así, en forma iterativa se calculan los errores de pose del vehículo en el instante t_k, se actualizan las variables del controlador y se calcula la nueva pose que tendrá el vehículo en el instante $t_{k+1} = t_k + T_S$ en respuesta a dichas variables de control. El Código (5.1) presenta el pseudocódigo correspondiente.

```
1  Inicializar variables
2  Importar puntos de ruta WP_n, n : 1, 2, ..., N
3  while i < N:
4    Calcular error posicion
5    if (distancia < ρ_0):
6      i = i + 1  % Avanzar al siguiente punto
7    else:
8      Actualizar variables de control
9      Simular modelo dinamico desde q(t = 0) a q(t = T_S)
10     Simular modelo cinematico desde q(t = 0) a q(t = T_S)
11     Sobreescribir condiciones iniciales % (i.e. q(t = 0) = q(t = T_S))
```

Código 5.1: Pseudocódigo de simulación con controlador de posición.

5.3.1. Resultados

A modo demostrativo, la Figura (5.3) muestra el recorrido del vehículo simulado en MATLAB® para un control tipo PI y camino circular, mientras que la Figura (5.4) presenta las curvas de distancia del vehículo al camino y al punto objetivo. Debe tenerse en cuenta que, para esta geometría, el vehículo parte desde el centro del círculo de radio $1.5\,m$, por lo que la distancia máxima registrada entre el vehículo y el camino de referencia es igual al radio para todas las simulaciones.

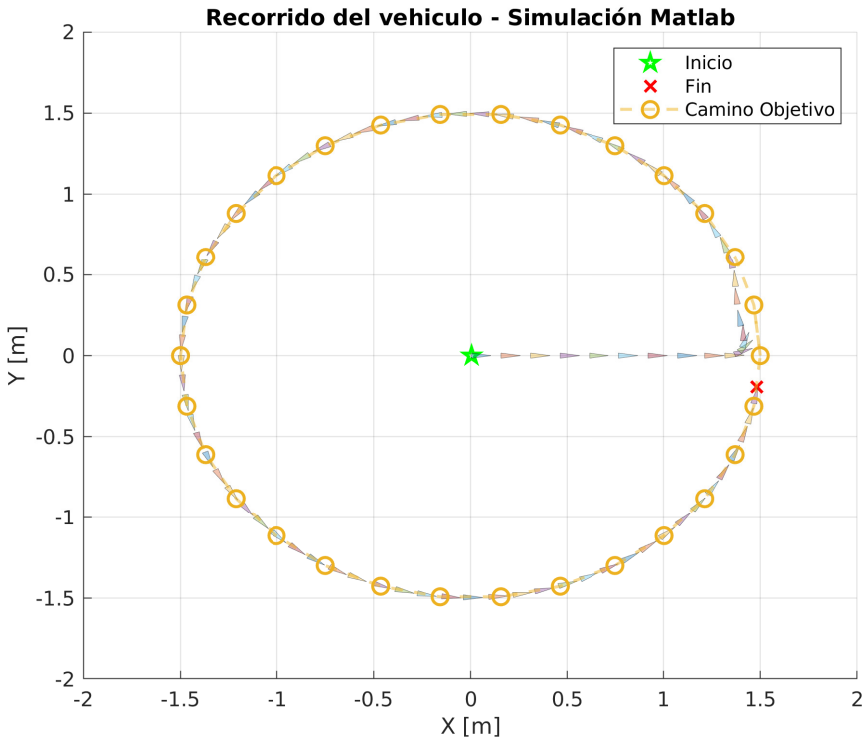


Figura 5.3: Seguimiento de caminos, recorrido circular, control proporcional + integral.

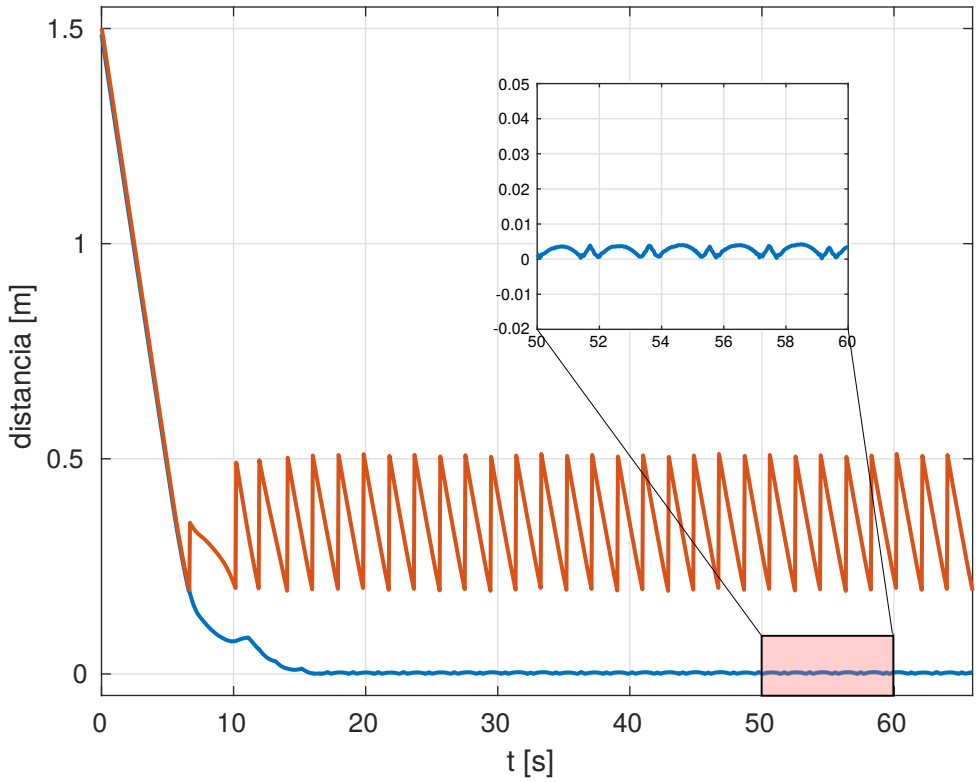


Figura 5.4: Distancias euclidianas medidas desde la posición del robot hasta el punto objetivo (en rojo) y hasta el punto más cercano del camino de referencia (en azul).

La Tabla (5.1) muestra las métricas de desempeño obtenidas con cada tipo de controlador, señalando en negrillas el mejor valor de cada métrica para cada una de las geometrías de camino previamente mencionadas.

	P				**PI**			
	d_{mean}	d_{std}	d_{max}	d_{ratio}	d_{mean}	d_{std}	d_{max}	d_{ratio}
Círculo	0.117	**0.272**	**1.500**	**1.084**	**0.094**	0.274	**1.500**	1.085
Cuadrado	0.026	0.032	0.117	0.842	**0.017**	**0.022**	**0.075**	**0.855**
Figura de 8	0.031	0.024	0.108	0.909	**0.029**	**0.023**	**0.101**	**0.912**

Tabla 5.1: Desempeño de controladores en distintas geometrías, simulados en MATLAB® .

Finalmente, con el objetivo de simplificar la comparación de desempeño entre los distintos controladores, la Figura (5.5) presenta los resultados de forma gráfica. El diagrama es del tipo *telaraña*, donde todos los ejes se dibujan en la misma escala, las distintas lineas de color corresponden a los distintos controladores y cada eje se corresponde con el valor de cada métrica, normalizada por el peor de los valores obtenidos. Por ejemplo: un punto rojo posicionado en el valor 1.0 de la métrica d_{mean} implica que el *controlador P* fue el que peor se desempeñó en esta métrica, para esta geometría. Por otro lado, un punto amarillo en el 0.5 implica que el *controlador PI* obtuvo una métrica 2 veces más pequeña que la del *controlador P*.

Dado que para el caso de seguimiento ideal d_{ratio} tiende a 1, mientras que las otras métricas tienden a 0, en las gráficas se utiliza la medida $|d_{ratio} - 1|$. De este modo, si un controlador presenta los valores más bajos para cada eje, significa que su desempeño fue superior al de los otros controladores, para las métricas analizadas. Tal es el caso del *controlador PI* para un camino cuadrado.

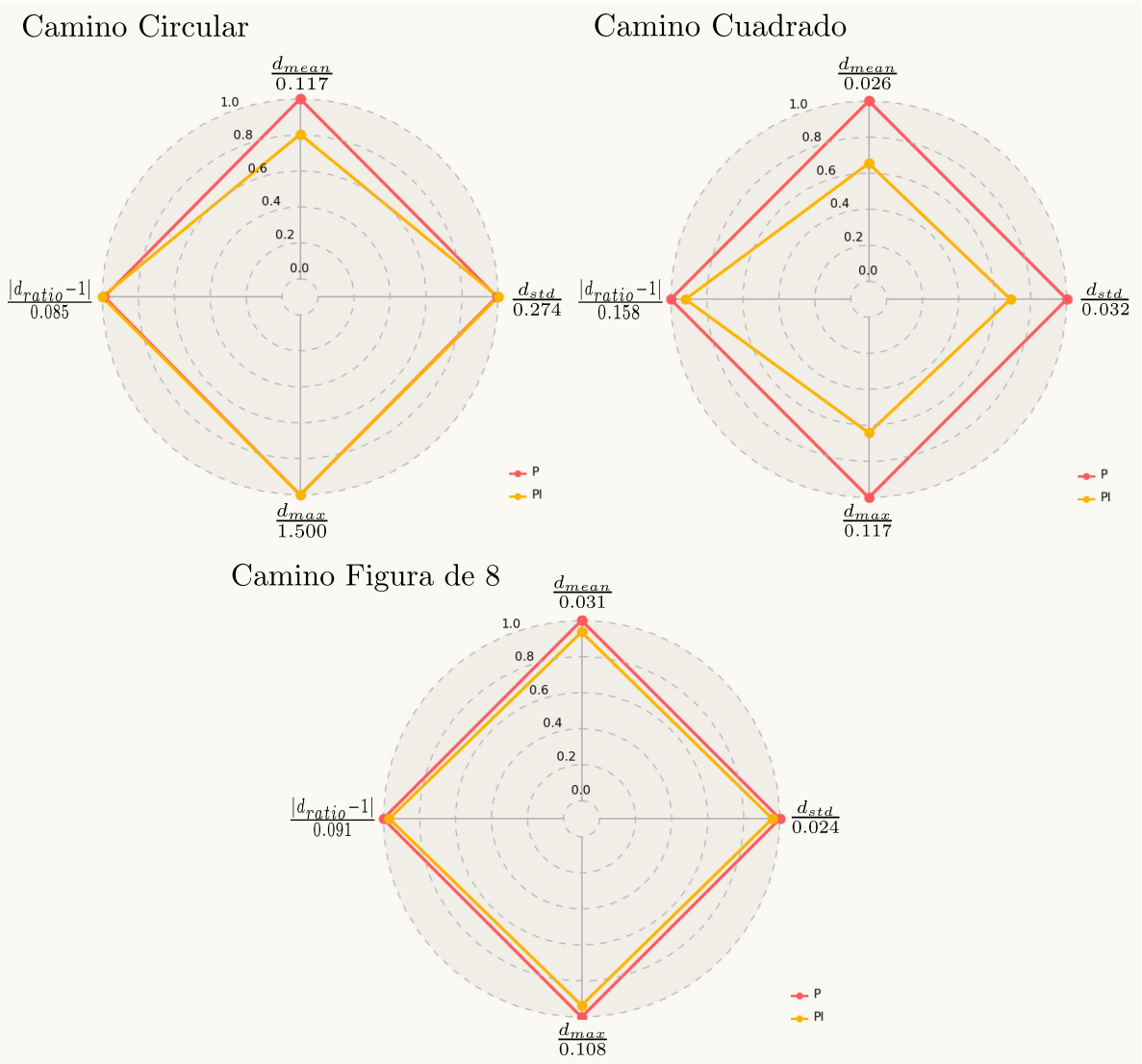


Figura 5.5: Comparación desempeño de los controladores simulados. Cada métrica ha sido normalizada por el valor indicado en los ejes correspondientes.

5.4. Implementación en Python - Gazebo

Como se mencionó en la Sección (4.2), la clase *PID* creada para controlar la velocidad del vehículo puede ser utilizada para controlar otros procesos. Por ejemplo, para el control de posición del vehículo se crearán tres instancias de *PID* que controlen las tres señales de error ρ, Δy y α.

En este trabajo, no se simulará el uso de sensores que permitan estimar la pose del vehículo, en cambio, se obtendrá la información de pose a partir de Gazebo. Para esto se implementó la clase *RobotBody*, que recibe como entrada el nombre del modelo que debe monitorear y se suscribe a los topics correspondientes. Además, cuenta con el método *wait_for_data()* que entra en un bucle ocioso hasta recibir y almacenar los últimos datos publicados por Gazebo.

La Figura (5.6) muestra el diagrama correspondiente a esta clase.

RobotBody
model_name : String
body_pos_x : Float32
body_pos_y : Float32
body_yaw : Float32
body_vel : Float32
body_twist_z : Float32
wait_for_data() : void

Figura 5.6: Diagrama de la clase *RobotBody*.

Por otro lado, se creó la clase *Trajectory* para realizar la carga de los puntos de ruta y entregar los mismos de manera secuencial. La carga requiere un archivo en formato *CSV* (valores separados por coma) que contenga hasta tres filas de datos, correspondiendo las primeras dos a las coordenadas x e y y la tercera, de contar con ella, a la coordenada temporal t, utilizada en seguimiento de trayectoria. Al alcanzar el último punto, el método *next_waypoint()* devuelve el valor booleano *Falso*, que es utilizado por el script principal para detener el bucle de control.

Trajectory
x : Float32
y : Float32
t : Float32
load_waypoints() : list, list, list
next_waypoint() : list, list, bool
select_waypoint() : list, list, bool

Figura 5.7: Diagrama de la clase *Trajectory*.

Así, el controlador de posición del tipo PID implementado en Python comienza por instanciar los tres *PID*, instanciar un objeto *Trajectory* e inicializar los puntos de ruta y crear un objeto *RobotBody* que reciba la pose del vehículo a controlar. Luego, se inicia un bucle que consiste en obtener la pose del vehículo, calcular el error de pose, generar las

señales de control correspondiente para anularlo y enviarlas al controlador de velocidad. El Código (5.2) presenta el pseudocódigo correspondiente a esta rutina.

```
1  Instanciar controladores PID
2  Instanciar Trajectory, cargar los mismos WP_n, n : 1, 2, ..., N
3  Instanciar RobotBody
4  while seguir:
5    Obtener pose
6    Calcular error posicion
7    if (distancia < ρ_0):
8      % Punto objetivo alcanzado
9      Trajectory -> siguiente punto
10     Resetear controladores
11   else:
12     Actualizar variables de control
13     Enviar comandos al controlador de velocidad
```

Código 5.2: Pseudocódigo del control de posición implementado.

Para el ajuste de los controladores PID de posición, se realizaron simulaciones con distintas geometrías que generaran entradas tipo escalón en cada variable controlada de manera independiente, a fin de observar con claridad la respuesta a cada una. Al igual que en el capítulo anterior, el criterio de ajuste consiste en obtener una respuesta sobreamortiguada y minimizar el tiempo de asentamiento. En estas simulaciones se encontró que el controlador proporcional generaba resultados similares a los controladores PI, PD y PID, con la ventaja de contar con menos parámetros a ajustar. Por lo tanto, se optó por configurar los tres controladores PID de posición en modo proporcional. Sin embargo, cabe destacar que si la pose del vehículo fuera estimada mediante sensores, las mediciones presentarán ruido. Ante estas condiciones, el desempeño controlador proporcional podría verse damnificado, en cuyo caso se recomienda utilizar un controlador PI.

5.4.1. Resultados

Las Figuras (5.8) - (5.10) muestran el recorrido del vehículo utilizando el control de posición proporcional, para una geometría circular, cuadrada y figura de ocho, respectivamente.

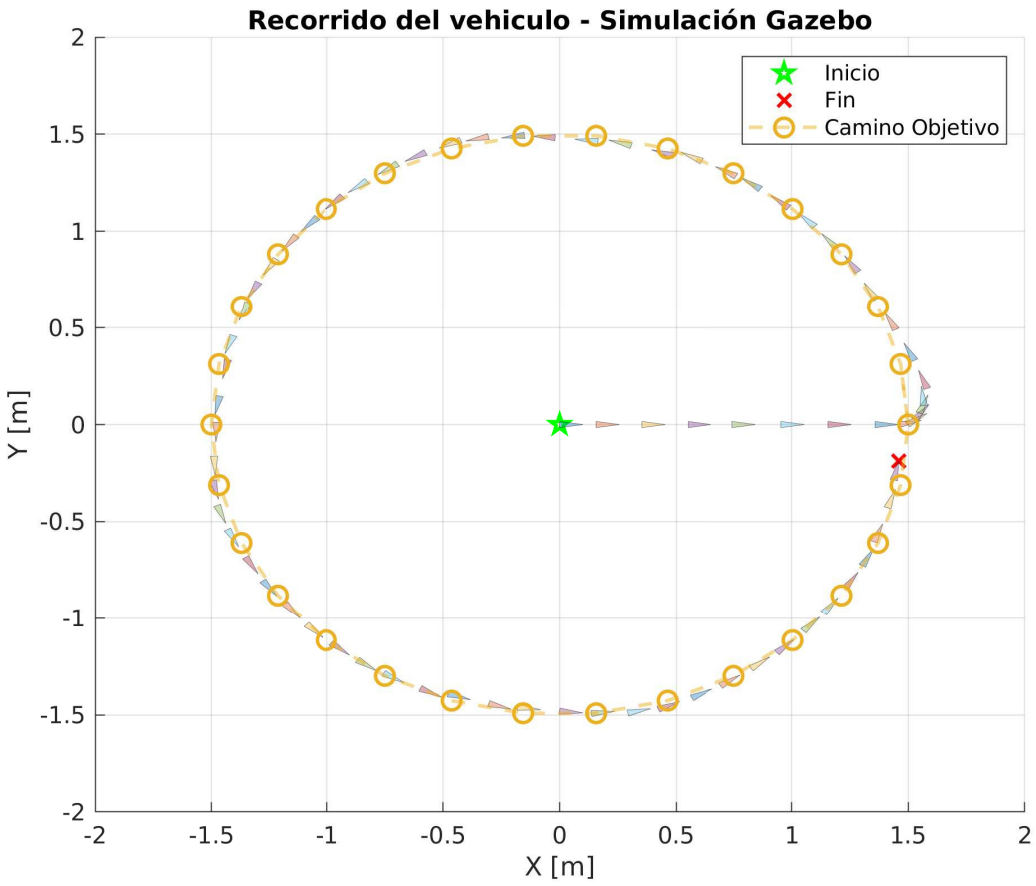


Figura 5.8: Seguimiento de caminos, recorrido circular, control P.

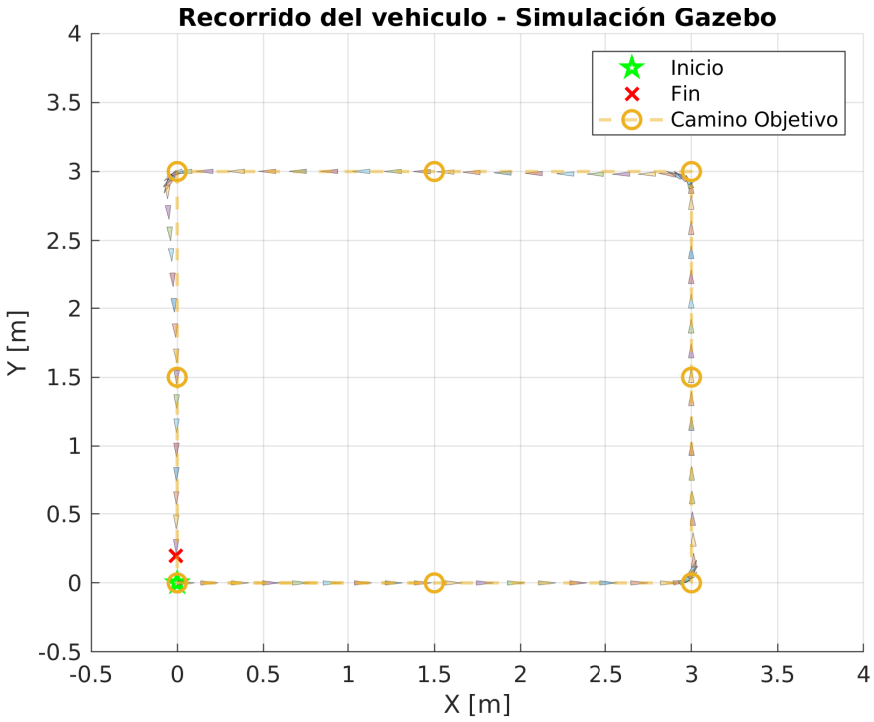


Figura 5.9: Seguimiento de caminos, recorrido cuadrado, control P.

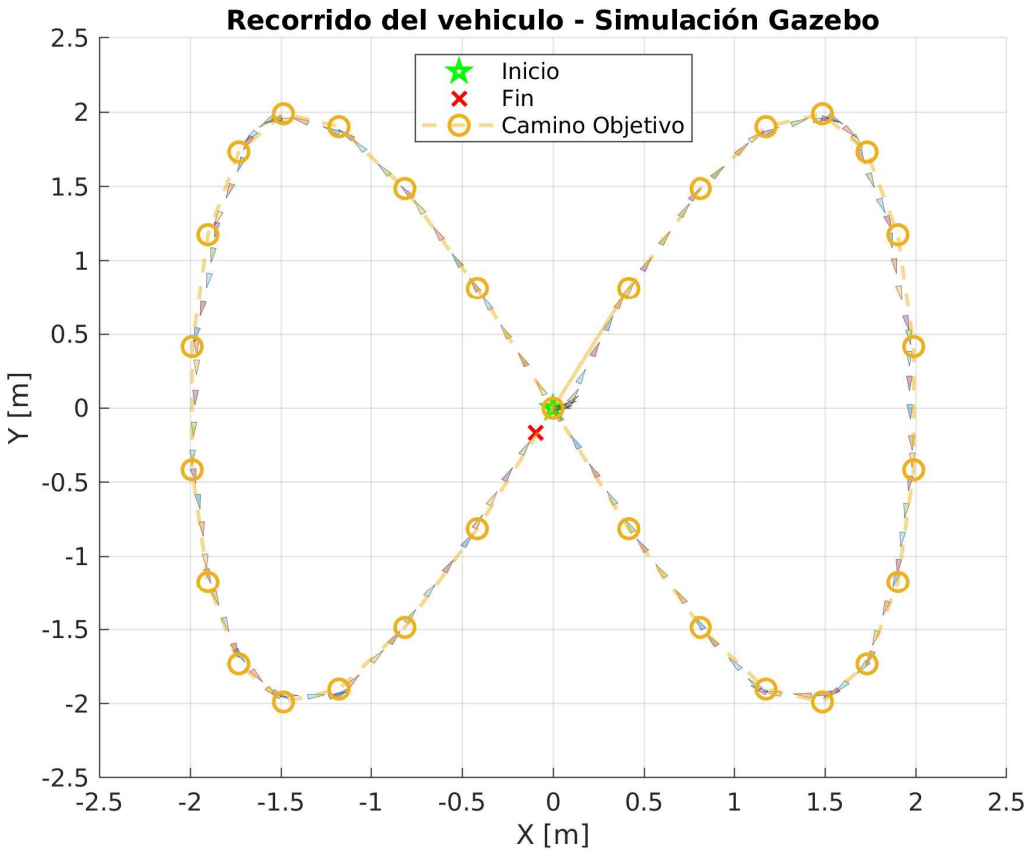


Figura 5.10: Seguimiento de caminos, recorrido figura de 8, control P.

Por otro lado, las Figuras (5.11) - (5.13) presentan las curvas de distancia del vehículo al camino de referencia (en rojo) y al punto objetivo (en azul), para cada tipo de recorrido. Puede observarse que, para los tres casos, la distancia al punto objetivo es siempre mayor o igual que la distancia al camino de referencia y que nunca es menor a $0.2\,m$.

Esto tiene sentido, pues $0.2\,m$ es el umbral establecido para continuar al siguiente punto objetivo y, precisamente, es la distancia al punto objetivo la que genera la acción del controlador: si esta fuera nula, el robot no se movería. Por el contrario, la distancia al camino de referencia presenta variaciones más suaves y su valor medio se aproxima a cero.

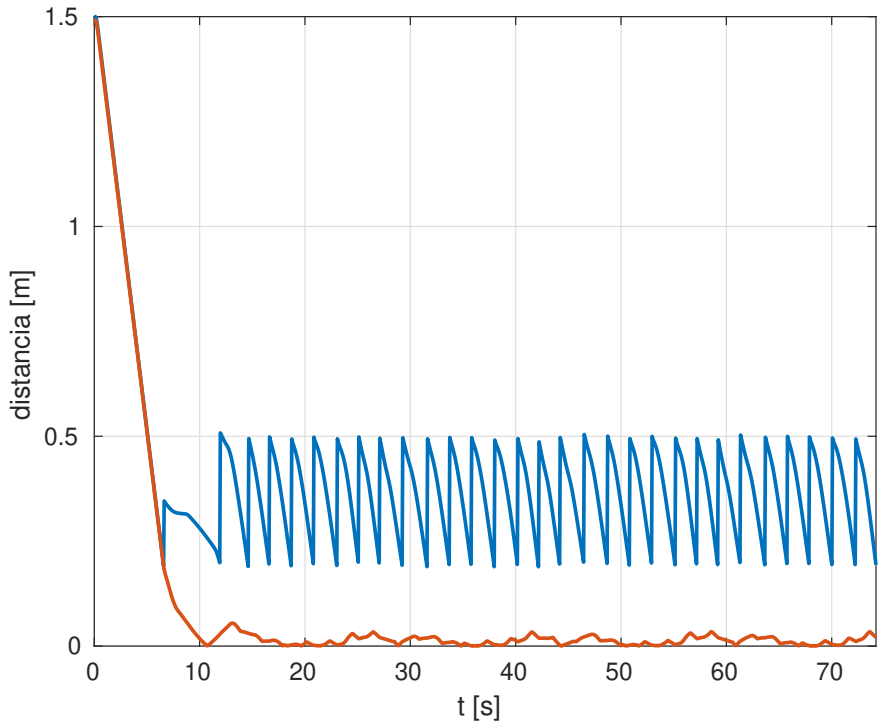


Figura 5.11: Distancias euclidianas medidas desde la posición del robot hasta el punto objetivo (en rojo) y hasta el punto más cercano del camino de referencia (en azul), para un recorrido circular.

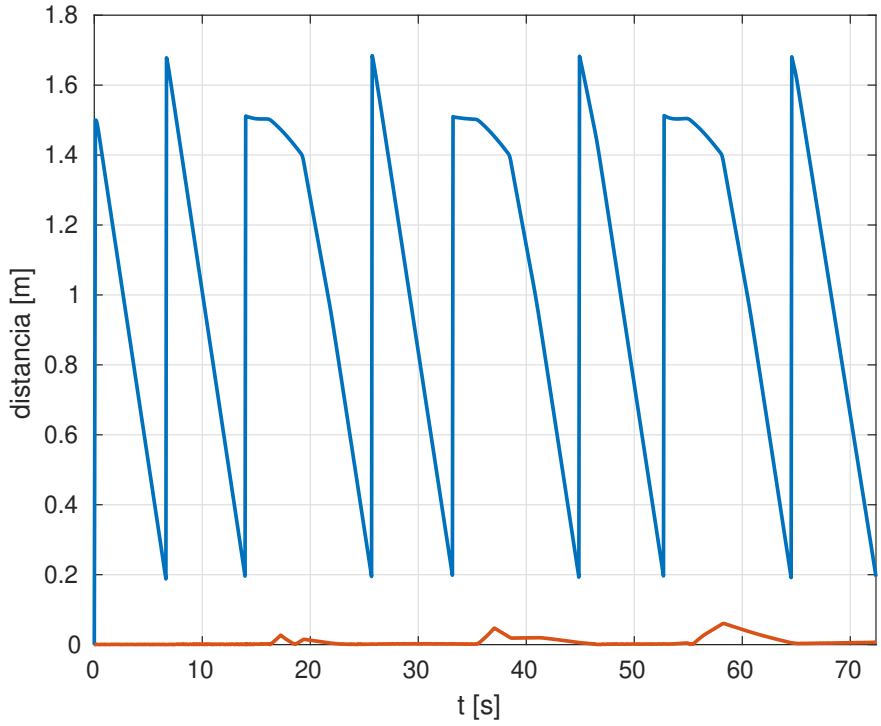


Figura 5.12: Distancias euclidianas medidas desde la posición del robot hasta el punto objetivo (en rojo) y hasta el punto más cercano del camino de referencia (en azul), para un recorrido cuadrado.

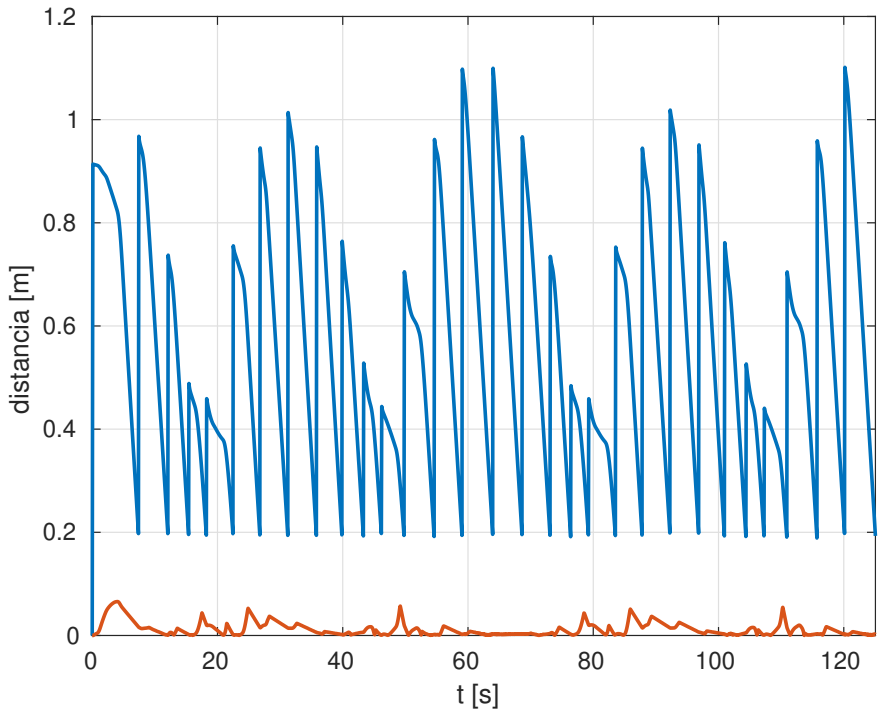


Figura 5.13: Distancias euclidianas medidas desde la posición del robot hasta el punto objetivo (en rojo) y hasta el punto más cercano del camino de referencia (en azul), para un recorrido figura de 8.

Finalmente, la Tabla (5.2) muestra las métricas de desempeño del controlador para cada simulación.

	d_{mean}	d_{std}	d_{max}	d_{ratio}
Círculo	0.094	0.269	1.500	1.101
Cuadrado	0.009	0.013	0.061	0.977
Figura de 8	0.016	0.012	0.063	0.987

Tabla 5.2: Desempeño del controlador de posición en distintas geometrías, simulados en MATLAB®

.

Capítulo 6

Seguimiento de trayectoria

En este capítulo se presenta la estrategia utilizada en [Arbinolo, 2020] para que el robot siga una trayectoria determinada haciendo uso de los controladores de posición descritos previamente. También se introducen tres controladores basados en la linealización del sistema [Scaglia et al., 2009], mostrando sus ecuaciones generales y desarollando uno de ellos para el modelo cinemático de un robot diferencial. Finalmente, se presenta el desempeño de cada controlador para las distintas geometrías propuestas. El aporte original de esta tesis consiste en la implementación de dichas estrategias y controladores en Python y la simulación de cada uno de ellos en Gazebo, en forma modular y configurable por el usuario.

6.1. Estrategia de seguimiento

Dependiendo de la tarea que desarrolla un robot, puede resultar de interés que el mismo realice el seguimiento de un camino o el seguimiento de una trayectoria. Como ya se ha mencionado, un *camino* consiste de dos o más coordenadas espaciales que indican los puntos por donde debe pasar el robot, dentro de cierto margen de tolerancia. Por otro lado, una *trayectoria* incluye una coordenada temporal asociada a cada uno de los puntos, que indica el instante t_n en que se espera que el robot alcance el punto (X_n, Y_n), en el caso de movimiento planar, dentro de cierto margen de tolerancia. Por lo tanto, en seguimiento de trayectoria, se selecciona el siguiente punto objetivo WP_{n+1} cuando el *tiempo de misión* t_M sea mayor al tiempo del objetivo actual t_n, sin importar la distancia al punto.

Para disminuir las oscilaciones en la orientación y suavizar el comportamiento general del robot [Arbinolo, 2020], se optó por interpolar linealmente las coordenadas X, Y y t en cada operación del controlador, obteniendo así un objetivo diferente para cada ciclo del mismo. La Figura (6.1) muestra el resultado de la interpolación para tres tiempos de misión distintos.

Nótese que la interpolación se realiza siempre entre dos puntos consecutivos de la trayectoria original, tal que $t_n \leq t_M < t_{n+1}$, por lo que la nueva coordenada estará sobre la recta

que une a ambos. Por lo tanto, no se modifica la geometría de la trayectoria original. Otro resultado de implementar esta estrategia es que se prioriza el acercamiento del robot a la trayectoria global en lugar de a los puntos que la componen, lo cual puede ser considerado como ventaja o desventaja, según la aplicación.

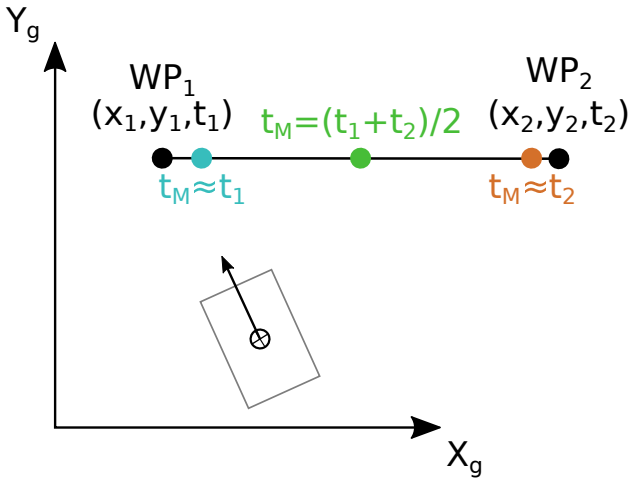


Figura 6.1: Demostración del efecto de interpolar el punto objetivo en cada ciclo.

De esta forma, el seguimiento de trayectoria puede implementarse mediante dos módulos diferentes: por un lado, la lógica de interpolación del punto objetivo; por el otro, el control de posición, tal como el controlador PID desarrollado en el capítulo anterior. La rutina de interpolación del punto objetivo se implementó en el método *select_waypoint()* de la clase *Trajectory*, como puede observarse en el diagrama de clase de la Figura (5.7).

Como alternativa al controlador PID, se implementó un tipo de controlador basado en linealizar el sistema mediante distintos métodos. Este controlador [Scaglia et al., 2009] será referenciado como *controlador de Scaglia*, por el nombre del autor que lo ha propuesto.

6.2. Seguimiento de trayectoria: controlador de Scaglia

La estrategia planteada por Scaglia consiste en linealizar el sistema en torno al instante de muestreo n y, a partir de las ecuaciones lineales, despejar el valor que deben tomar las entradas en el instante n para que los estados alcancen el valor deseado en el siguiente paso $n+1$. Para la tarea de linealización, propone los siguientes tres métodos:

- Desarrollo en series de Taylor
- Interpolación lineal
- Interpolación por métodos numéricos (Euler y Runge-Kutta)

A continuación se presentan las ecuaciones generales y el planteamiento de cada método para la ecuación diferencial de primer orden:

$$\dot{y} = h(y, u, t), \qquad y(0) = y_0. \tag{6.1}$$

Los valores que $y(t)$ toma en los intervalos de muestreo $t = nT_0$, con $n = 0, 1, 2, ...$ se representarán como $y_n = y(nT_0)$.

Para el método de **aproximación por series de Taylor**, si y es $m+1$ veces derivable en torno a $t = nT_0$, entonces puede ser aproximada por el polinomio de Taylor de orden m:

$$y(t) = y(nT_0) + \left.\frac{dy}{dt}\right|_{t=nT_0} (t - nT_0) + \left.\frac{d^2y}{dt^2}\right|_{t=nT_0} \frac{(t-nT_0)^2}{2!} + ... + \left.\frac{d^my}{dt^m}\right|_{t=nT_0} \frac{(t-nT_0)^m}{m!} \tag{6.2}$$

A modo de ejemplo, si se utiliza el polinomio de Taylor de orden uno, para estimar y en $t = (n+1)T_0$ se tiene:

$$y_{n+1} \approx y_n + \dot{y}_n T_0 \tag{6.3}$$

Reemplazando la Ecuación (6.1) en (6.3):

$$y_{n+1} = y_n + h(y_n, u_n, t_n) T_0 \tag{6.4}$$

Si se reemplaza y_{n+1} por el valor de referencia yd_{n+1} en la Ecuación (6.4), puede observarse que la única incógnita es la señal de control u_n. Luego, puede obtenerse el valor de u_n que lleve el sistema al valor estado deseado en el siguiente instante de muestreo.

Por otro lado, el método de **interpolación lineal** busca aproximar a $y(t)$ en el intervalo $[nT_0;\ (n+1)T_0]$ por la recta P que pasa entre los puntos $y(nT_0)$ y $y(nT_0 + T_0)$, es decir:

$$y(t) \approx P(t) = y_n + \frac{y_{n+1} - y_n}{T_0}(t - nT_0) \tag{6.5}$$

Integrando la Ecuación (6.1), se cumple lo siguiente:

$$y_{n+1} = y_n + \int_{nT_0}^{(n+1)T_0} h(y, u, t)dt \tag{6.6}$$

Reemplazando la Ecuación (6.5) en (6.6) y haciendo $y_{n+1} = yd_{n+1}$ se obtiene:

$$yd_{n+1} = y_n + \int_{nT_0}^{(n+1)T_0} h(P(t), u, t)dt \tag{6.7}$$

Al igual que en el método anterior, resolviendo la integral de la Ecuación (6.7) se puede despejar el valor que debe tomar la señal de control en el instante n.

Finalmente, el método de **interpolación por métodos numéricos** propone resolver la ecuación diferencial en (6.1) aproximando a $y(t)$ con el esquema Runge-Kutta de dos etapas, cuyas ecuaciones se presentan a continuación. Cabe destacar que, para aplicar este método, es necesario contar con la información de velocidad del vehículo, además de su posición, lo que no ocurre con los otros métodos aquí presentados.

$$y_{n+1} = y_n + \frac{T_0}{2}\left[h(y_n, u_n, t_n) + h(y_{n+1}, u_{n+1}, t_{n+1})\right] \tag{6.8a}$$

$$h(y_{n+1}, u_{n+1}, t_{n+1}) = h(y_n + T_0 h(y_n, t_n), u_{n+1}, t_{n+1}) \tag{6.8b}$$

Habiendo presentado las ecuaciones generales de cada método, se procede a detallar en la siguiente sección el procedimiento realizado para aplicar el método por series de Taylor al modelo cinemático de un robot diferencial ideal, expresado en la Ecuación(2.4). No se utilizó el modelo de skid-steer de la Ecuación (2.6) ya que el mismo requiere la identificación del parámetro x_{ICR}, tarea que, como se mencionó previamente, no formará parte de este trabajo.

6.2.1. Seguimiento de trayectoria basado en series de Taylor

En esta sección se presentan las ecuaciones que explican el controlador propuesto, procurando no entorpecer la lectura con detalles excesivos de los pasos realizados para obtenerlas. Si el lector desea ahondar en dichos detalles, puede encontrar un desarrollo más puntilloso en el Capítulo (9), al final de este trabajo.

Por comodidad, en la Ecuación (6.9) se presentan nuevamente las ecuaciones del modelo cinemático del robot diferencial, que serán linealizadas aplicando el primer método descrito en la sección anterior. Para mayor legibilidad, se reemplazaron las variables de control v_x y ω_z por V y W, respectivamente. Se busca calcular las señales de control V_n y W_n que generen que el vehículo pase de la posición (x_n, y_n) a la posición deseada $(x_{d_{n+1}}, y_{d_{n+1}})$ en el siguiente instante de muestreo, donde (x_{d_n}, y_{d_n}) son las coordenadas espaciales de la trayectoria de referencia para el instante n.

$$\begin{aligned} \dot{x} &= V\cos\theta \\ \dot{y} &= V\sin\theta \\ \dot{\theta} &= W \end{aligned} \tag{6.9}$$

Si se aproxima x, y y θ utilizando un polinomio de Taylor de orden 1, se obtiene:

$$\begin{aligned} x_{n+1} &= x_n + \dot{x}_n T_0 \\ y_{n+1} &= y_n + \dot{y}_n T_0 \\ \theta_{n+1} &= \theta_n + \dot{\theta}_n T_0 \end{aligned} \tag{6.10}$$

Se reemplazan los valores futuros de los estados por los valores de referencia, es decir x_{n+1} por $x_{d_{n+1}}$ e y_{n+1} por $y_{d_{n+1}}$. Al mismo tiempo, se introduce la variable θ_{d_n}, que representa la orientación necesaria en el instante n para que el robot alcance la posición deseada en $n+1$, y se calcula como sigue:

$$\theta_{d_n} = \arctan \frac{y_{d_{n+1}} - y_n}{x_{d_{n+1}} - x_n}. \tag{6.11}$$

El autor propone una función de coste J, que pondera la energía del error de seguimiento y la energía de las derivadas de las variables de estado.

$$J_n = k_1^2[(x_{d_{n+1}} - x_{n+1})^2 + (y_{d_{n+1}} - y_{n+1})^2] + k_2^2(\dot{x}_n^2 + \dot{y}_n^2) + k_3^2(\theta_{d_n} - \theta_{n+1})^2 + k_4^2\dot{\theta}_n^2 \tag{6.12}$$

Luego, la estrategia de control está dada por los V_n y W_n que minimizan el coste en el instante n, expresados en las Ecuaciones (6.13a) y (6.13b).

$$V_n = k_v^2 \left\{ \frac{x_{d_{n+1}} - x_n}{T_0} \cos\theta_{d_n} + \frac{y_{d_{n+1}} - y_n}{T_0} \sin\theta_{d_n} \right\} \tag{6.13a}$$

$$W_n = k_w^2 \frac{\theta_{d_n} - \theta_n}{T_0} \tag{6.13b}$$

donde

$$k_v^2 = \frac{k_1^2}{k_1^2 + k_2^2/T_0^2} \tag{6.14a}$$

$$k_w^2 = \frac{k_3^2}{k_3^2 + k_4^2/T_0^2} \tag{6.14b}$$

Puede observarse que k_v^2 y k_w^2 pertenecen al intervalo real [0; 1]. Aún más, vale remarcar que no reviste importancia el valor exacto que tomen las constantes $k_{1,2,3,4}$, sino la relación entre ellas dada por las Ecuaciones (6.14a) y (6.14b). En este trabajo, se optó por utilizar $k_v^2 = k_w^2 = 0.4$, que se corresponde con asignar un peso aproximadamente 10 veces mayor a la energía del error de posición y orientación respecto de la energía de las variables de control, es decir $k_1^2 \approx 10k_2^2$ y $k_3^2 \approx 10k_4^2$.

6.3. Resultados

Las métricas para evaluar el seguimiento de trayectoria son las mismas utilizadas para seguimiento de camino. Si bien los cálculos no involucran la componente temporal en forma explícita, es de esperar que el trayecto realizado por el robot se aleje o sea más corto que la trayectoria de referencia, especialmente al aumentar la velocidad de referencia.

La Figura (6.2) muestra la trayectoria realizada por el vehículo ante una trayectoria de referencia circular, con velocidad lineal de referencia de $0.1\,m/s$. Recordando lo observado en seguimiento de camino, el vehículo alcanzaba la cercanía del primer punto objetivo $(1.5, 0)$ y luego continuaba con el resto de los puntos. Por otro lado, en seguimiento de trayectoria el robot intenta alcanzar el punto correspondiente al tiempo de la misión, por lo que se puede observar que inicia su avance en dirección al primer punto, pero luego rota gradualmente a medida que varía el punto objetivo.

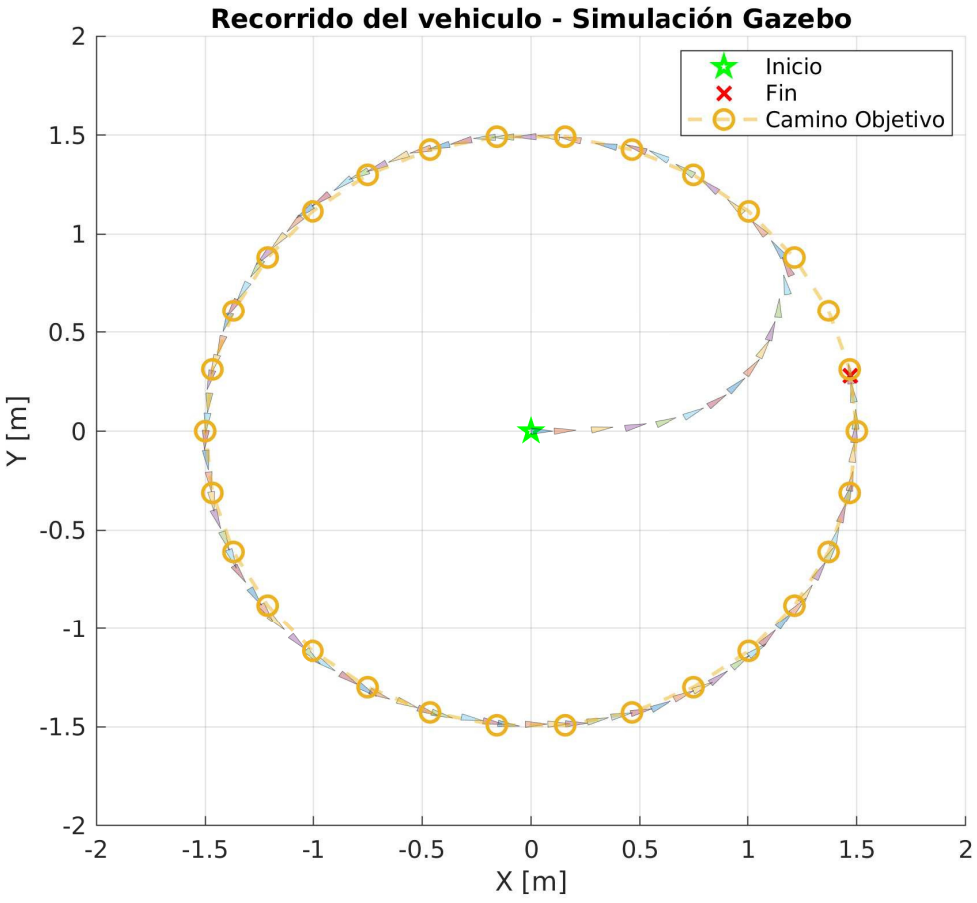


Figura 6.2: Seguimiento de trayectoria, recorrido circular, control PID.

La Figura (6.3) presenta la comparación de desempeño entre los distintos controladores implementados, para las distintas trayectorias. Por practicidad, las referencias del gráfico presentan los métodos de seguimiento de trayectora de Scaglia bajo nombres alternativos, cumpliéndose las siguientes equivalencias:

- *Scaglia_v1* = desarrollo en series de Taylor.
- *Scaglia_v2* = interpolación lineal.
- *Scaglia_v3* = interpolación por métodos numéricos.

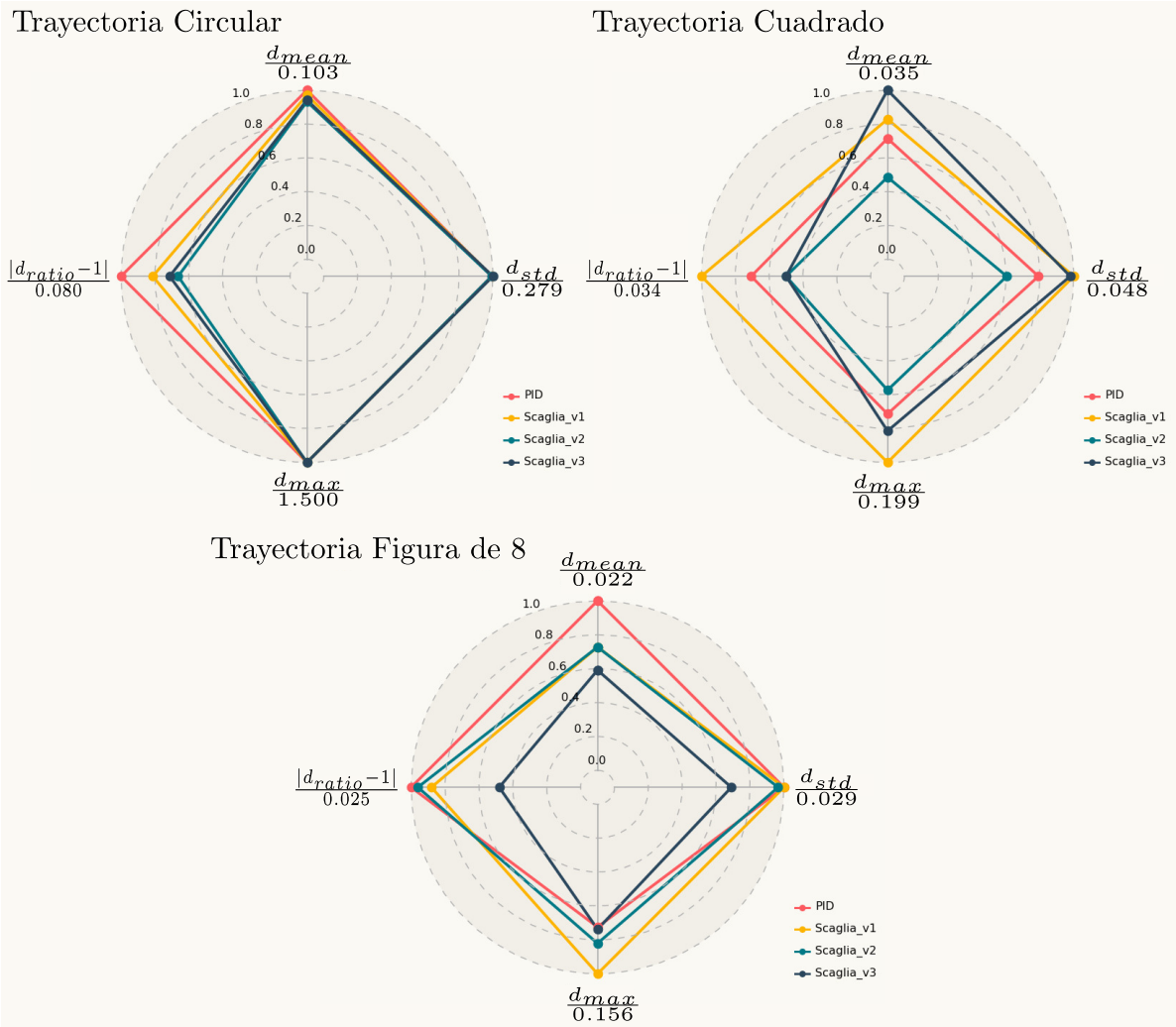


Figura 6.3: Comparación desempeño de los controladores evaluados. Cada métrica ha sido normalizada por el valor indicado en los ejes correspondientes.

Capítulo 7

Uso de la biblioteca de simulación

En este capítulo se presenta una serie de pasos que puede seguirse para implementar un controlador de posición del tipo Scaglia, a partir de la biblioteca desarrollada en este trabajo. Con esto, se busca ilustrar en forma simultánea el modo de uso y la practicidad de la biblioteca.

Se presenta la clase *MasterController*, que sirve para construir subclases que tengan ciertos métodos y/o atributos en común, evitando así la duplicación de código. Esta clase no debe ser instanciada directamente pues declara uno o más métodos que no realizan acción alguna, los cuales deben ser sobreescritos por la subclase. En particular, *MasterController* define una interfaz común para los controladores PID y del tipo Scaglia, presentados en el Capítulo (6).

7.1. Ejemplo: implementar controlador de Scaglia

La Figura (7.1) muestra un diagrama en bloques con las partes principales de la biblioteca, donde se puede visualizar la interacción entre los distintos archivos y funciones que la componen. La Figura (7.2) hace foco sobre la clase MasterController y su interacción con sus subclases y otros módulos. A continuación, se procede a detallar los pasos sugeridos para implementar un controlador del tipo Scaglia y realizar seguimiento de trayectoria. Se asume que la biblioteca ya ha sido descargada del repositorio y ubicada en el directorio de trabajo *~/catkin_ws/src*. Esto puede realizarse mediante los comandos:

```
      ~$  mkdir -p ~/catkin_ws/src
      ~$  cd ~/catkin_ws/src
.../src$  git clone https://gitlab.com/BDA92/givsi-unco-ssmr.git
```

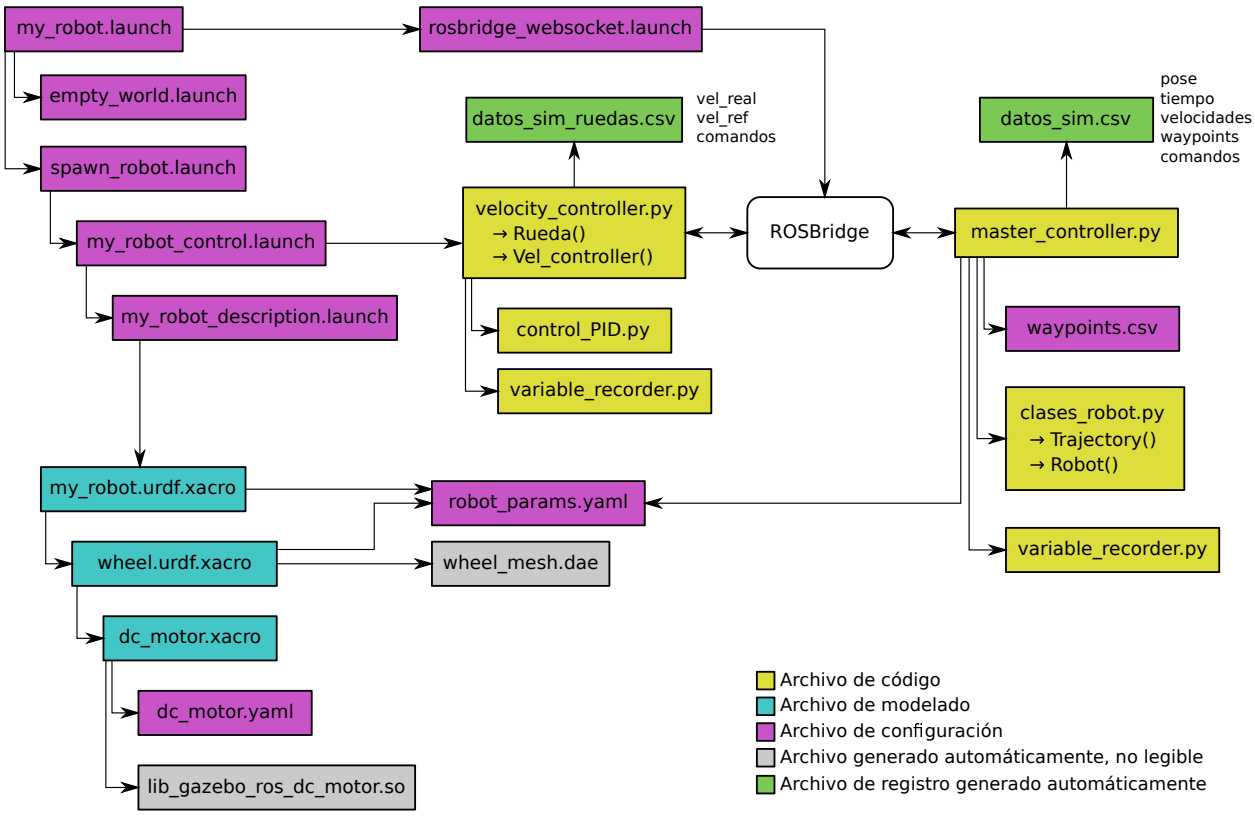


Figura 7.1: Diagrama bloques para *my_robot_control*.

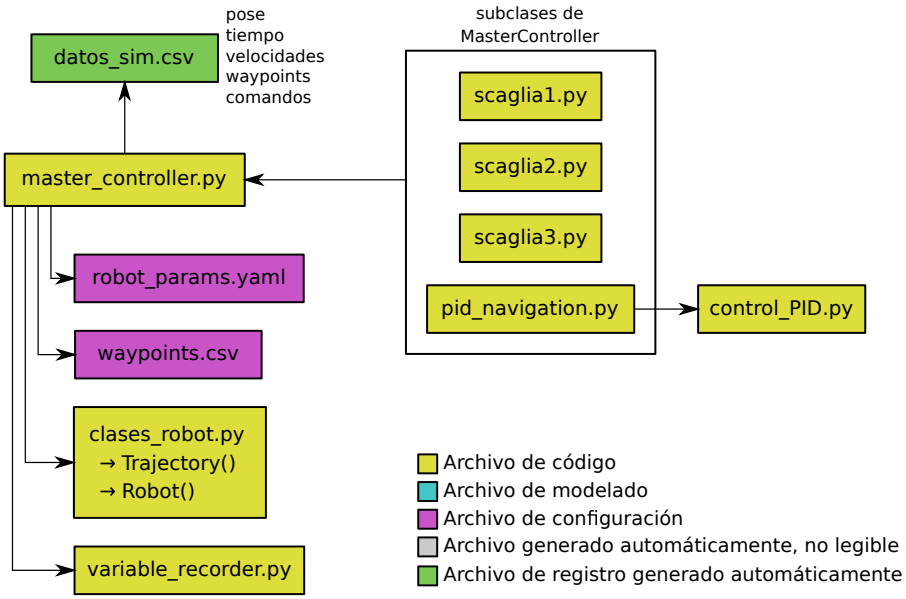


Figura 7.2: Diagrama clase *MasterController* y subclases.

7.1.1. Escribir y probar el algoritmo

Los códigos de Python asociados a la lógica de control de posición del vehículo se encuentran en la carpeta *scripts*. Allí se creará un nuevo archivo con nombre *scaglia_runge_kutta.py*, en referencia al método de linealización utilizado en el controlador.

```
~/catkin_ws/src$  cd scripts
.../scripts$  code scaglia_runge_kutta.py
```

Como ya se ha mencionado, se utilizará la clase *MasterController* como clase padre de nuestro controlador, por lo tanto, en las primeras líneas de código se importará esta clase, así como las bibliotecas *numpy* y *math* y la función *wrap_to_pi*, que serán de utilidad más adelante.

```
from master_controller import MasterController
import numpy as np
import math
from clases_robot import wrap_to_pi
```

Se procede a crear la clase que implemente el controlador de Scaglia por Runge Kutta de dos etapas, la cual se llamará *ScagliaRK*. La clase *MasterController* ha sido documentada, lo cual permite visualizar información sobre el uso de la misma, la interfaz necesaria para implementar un controlador correctamente y los métodos y atributos que posee. En particular, en VSC se puede visualizar esta información al posar el mouse sobre el nombre de la clase, como se muestra en la Figura (7.3). Lo mismo es válido para cada uno de los métodos y clases de la biblioteca.

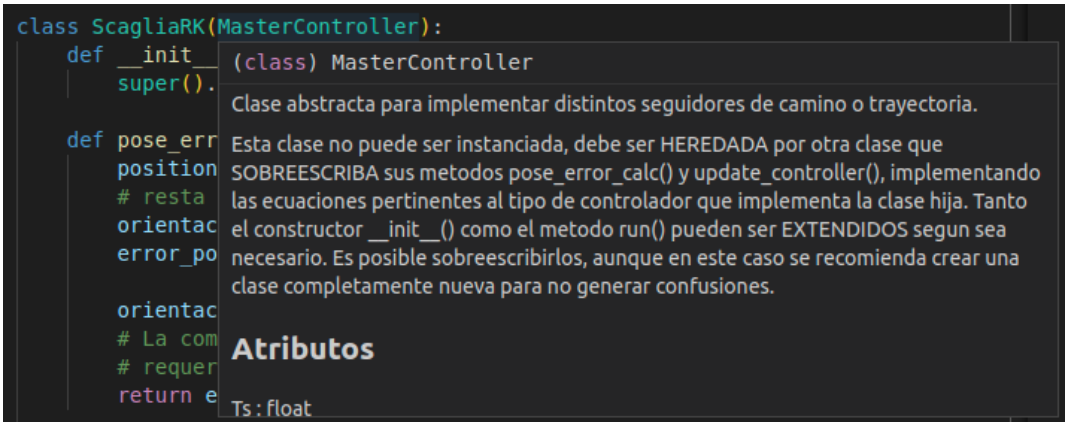


Figura 7.3: Visualización de la documentacion de la biblioteca en VSC.

Como puede observarse en la documentación, los métodos *pose_error_calc* y *update_controller* deben ser sobreescritos por la subclase, el primero se encarga de realizar los cálculos necesarios para obtener las variables de error de posición utilizadas por el controlador, mientras que el segundo se encarga de calcular la señal de control en base al error de posición y es el que implementa las ecuaciones características del controlador.

En cuanto al constructor de la clase, es posible utilizar el constructor de *MasterController* mediante la función *super()*, el cual toma dos argumentos de entrada: **seguimiento**, indica si se realizará seguimiento de trayectoria o de camino; y **waypoints**, es la ruta al archivo CSV que contiene los puntos de ruta a seguir. Dado que Scaglia es utilizado únicamente para seguimiento de trayectorias, no es necesario indicarlo como argumento de entrada en la nueva clase, por lo que el constructor puede escribirse como sigue

```
class ScagliaRK(MasterController):
    def __init__(self, waypoints='waypoints_circle.csv'):
        super().__init__(seguimiento='Trayectoria', waypoints=waypoints)
```

A continuación se presentan las ecuaciones propias del controlador de Scaglia por Runge Kutta, para un modelo de robot diferencial ideal:

$$V_{n+1} = k_v \frac{2}{T_0} \left(\Delta x \cos\theta_d + \Delta y \sin\theta_d\right) - V_n \cos\left(\theta_d - \theta_n\right) \tag{7.1a}$$

$$W_{n+1} = k_w \left(\frac{2}{T_0}\Delta\theta - W_n\right) \tag{7.1b}$$

$$\theta_d = \arctan \frac{k_v \frac{2}{T_0}(\Delta y) - V_n \sin\theta_n}{k_v \frac{2}{T_0}(\Delta x) - V_n \cos\theta_n} \tag{7.1c}$$

donde $\Delta x = wp_{x_n} - x_n$ y $\Delta y = wp_{y_n} - y_n$.

Observando estas ecuaciones, se puede apreciar que las únicas variables asociadas al error de posición son Δx y Δy, por lo que se puede sobreescribir *pose_error_calc* como sigue

```
def pose_error_calc(self, pose, goal_present, goal_next):
    error_x = goal_present[0] - pose[0]
    error_y = goal_present[1] - pose[1]
    return error_x, error_y
```

Por otro lado, el método *update_controller* recibe, como argumento de entrada, la salida de *pose_error_calc*. De las Ecuaciones (7.1) se deduce que es necesario utilizar la información de orientación (*body_yaw*), velocidad lineal (*body_vel*) y velocidad angular (*body_twist_z*) del

robot, así como la frecuencia de muestreo del controlador (Ts). Todas estas variables son atributos de *MasterController*, por lo cual son también atributos de la clase *ScagliaRK* y pueden ser utilizados dentro del método. Por lo tanto, es posible implementar *update_controller* como sigue

```
def update_controller(self,pose_error_tuple):
    error_position = pose_error_tuple[0]
    error_x = error_position[0]
    error_y = error_position[1]
    k_v = 0.4
    k_w = 0.5
    # ecuacion 5.4.1.17 Tesis Scaglia
    theta = math.atan2(
           k_v*2/self.Ts*error_y - self.body_vel*math.sin(self.body_yaw),
           k_v*2/self.Ts*error_x - self.body_vel*math.cos(self.body_yaw))
    cmd_vel_lineal = k_v*2/self.Ts \
           *(error_x*math.cos(theta) + error_y*math.sin(theta)) \
           - self.body_vel*math.cos(theta - self.body_yaw)
    cmd_vel_angular = k_w*(2/self.Ts*wrap_to_pi(theta - self.body_yaw)
           - self.body_twist_z)
    return cmd_vel_lineal, cmd_vel_angular
```

Nótese que, por razones de legibilidad del código, se han modificado los nombres de algunas variables, respecto de las Ecuaciones (7.1), se confía en que estos cambios sean transparentes al lector Cabe destacar que las constantes k_v y k_w son los parámetros de ajuste del controlador, por lo que encontrar su valor ideal queda en manos del programador. En este caso han sido fijados arbitrariamente a 0.4 y 0.5. Finalmente, la función *wrap_to_pi* recibe un ángulo α indicado en radianes, tal que $\alpha \in \mathbb{R}$, y devuelve el ángulo equivalente β, tal que $\beta \in [-\pi; \pi]$. Aquí se la utiliza para garantizar que la señal de referencia de velocidad angular esté siempre acotada.

Eso concluye la implementación del controlador en la clase *ScagliaRK*. Para probar su funcionamiento, puede crearse una instancia de la misma y llamar al método *run*, que iniciará el ciclo de control de posición. Una forma de hacer esto es dentro del mismo archivo, con un bloque condicional que inicie un controlador cuando el archivo se ejecute directamente:

```
if __name__ == '__main__':
    controller = ScagliaRK('waypoints_square.csv')
    controller.run()
```

De esta forma, es posible iniciar una simulación, ejecutar el archivo *scaglia_runge_kutta.py* y observar cómo el vehículo inicia el seguimiento de la trayectoria indicada, todo mediante los siguientes comandos, que se recomienda ejecutar en diferentes terminales:

```
.../scripts$  roslaunch my_robot_gazebo my_robot.launch
.../scripts$  gzclient
.../scripts$  python scaglia_runge_kutta.py
```

El primer comando inicia la simulación de un mundo que sólo contiene una instancia del robot; el segundo es opcional, pues inicia la interfaz gráfica de Gazebo, en caso de desear visualizar el desarrollo de la simulación; finalmente, el tercer comando inicia el control de posición. Una vez finalizado el recorrido, los datos recolectados durante la simulación pueden ser encontrados en el directorio *~/catkin_ws/src/sim_results* en formato CSV.

Para más detalles relacionados a este ejemplo, se recomienda revisar la documentación propia de la clase *MasterController* y la biblioteca en general. Aún más, puede encontrarse la versión en video de este tutorial en la plataforma YouTube [1].

7.1.2. Adquisición de estado del vehículo

Como se mencionó previamente, en este trabajo se utiliza la información de posición, orientación y velocidades reales del robot, provistas por Gazebo. Para que la biblioteca pueda utilizarse tanto en simulación como con un robot real, resulta necesario capturar dicha información a través de un topic diferente.

Supóngase que se cuenta con un robot real que publica su localización en el topic */robot/estados*. Luego, resulta necesario indicarle a la biblioteca que debe suscribirse a dicho topic. Esto puede realizarse mediante el archivo de configuración *robot_params.yaml* del paquete *my_robot_description*, escribiendo el nombre del topic pertinente en el campo *robot_state_topic*. Cabe mencionar que, según el caso, también puede ser necesario modificar el tipo de topic a utilizar, para lo cual se debe editar el campo *robot_state_topic_type*, según corresponda. Si se modifica el tipo de topic, luego deberá sobreescribirse el método *robot_state_callback* de la clase *Robot*, que se encarga de capturar el mensaje recibido y almacenar cada tipo de dato en la variable correspondiente.

[1] Video tutorial: https://youtu.be/EDjNqGO57EU

Por ejemplo, si el topic */robot/estados* transmite posición y orientación bajo el tipo de mensaje estándar *geometry_msgs/Pose* se debe analizar cómo está compuesto el mensaje. Según la documentación[2], *geometry_msgs/Pose* se compone de la siguiente forma:

```
pose = {
        'position' : {'x':float64, 'y':float64, 'z':float64},
        'orientation' : {'x':float64, 'y':float64, 'z':float64, 'w':float64}
        }
```

Para este ejemplo, los cambios a realizar en *robot_params.yaml* se muestran Figura (7.4), mientras que el Código (7.1) muestra la correcta implementación del método *robot_state_callback* para el nuevo tipo de mensaje.

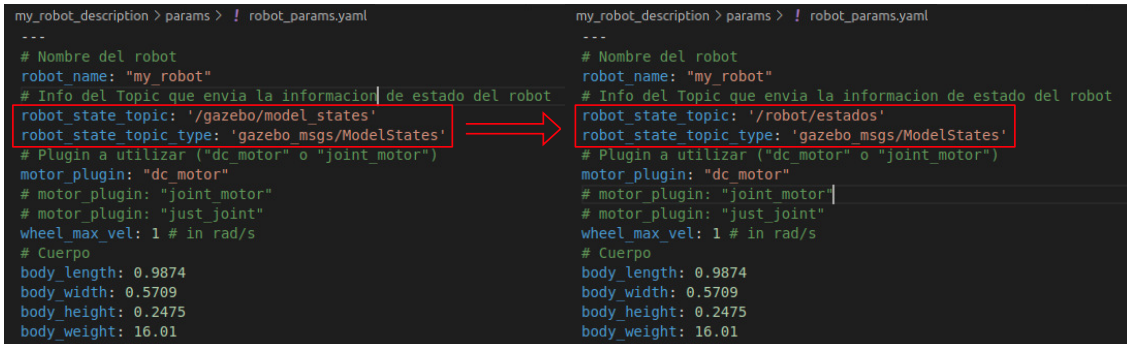


Figura 7.4: Modificación del nombre del topic dentro de robot_params.yaml.

```
class Robot():
    def robot_state_callback(self, topic_msg):
        # Metodo encargado de manejar data recibida por el topic link_state
        # Almaceno la data recibida en los respectivos atributos de la clase
        self.body_pos_x = topic_msg['position']['x']
        self.body_pos_y = topic_msg['position']['y']
        self.body_yaw = euler_from_quaternion(
            topic_msg['orientation']['x'],
            topic_msg['orientation']['y'],
            topic_msg['orientation']['z'],
            topic_msg['orientation']['w'])
        # "Flag" de nuevo dato disponible
        self.new_data = True
```

Código 7.1: Implementación de *robot_state_callback* para recibir datos del tipo *geometry_msgs/Pose*

[2]https://docs.ros.org/en/api/geometry_msgs/html/msg/Pose.html

Capítulo 8

Conclusiones

En primer lugar, es posible afirmar que se han cumplido los objetivos estipulados para este Proyecto Integrador Profesional, enumerados al comienzo del escrito y reiterados a continuación:

- Estudiar el modelado cinemático de robtos con configuración SSMR; se estudiaron diferentes modelos cinemáticos y dinámicos de robots en configuración skid-steer, escogiendo un módelo de cada tipo e implementándolos en MATLAB® .
- Definir e implementar una estrategia de control lineal para realizar seguimiento de trayectorias; se implementaron cuatro seguidores de trayectorias basados en dos principios de control lineal: control PID y linealización de la planta mediante distintos métodos.
- Implementar un modelo tridimensional de un SSMR dentro de un entorno de simulación; se utilizó el entorno de simulación Gazebo en conjunto con la herramienta ROS para gestionar la comunicación entre las partes. Se implementó un modelo simple de SSMR, consistente de cuatro ruedas unidas a un cuerpo de prisma rectangular, cuyas dimensiones pueden ser modificadas mediante la edición de un único archivo de configuración YAML.
- Implementar en una biblioteca de Python los controladores y la comunicación con el entorno de simulación; toda la lógica de control fue implementada en Python bajo el paradigma de Programación Orientada a Objetos. La comunicación con el entorno de simulación se realiza mediante ROS y la biblioteca ROSLibPy, que permite a Python 3.x interactuar con ROS.
- Definir al menos dos métricas para determinar el desempeño de las estrategias de

control implementadas; se escogieron cuatro métricas de desempeño, utilizables tanto para seguimiento de caminos como seguimiento de trayectorias.

- Documentar el trabajo realizado, a fin que el mismo pueda ser replicado, editado o ampliado por otras personas en el futuro; los códigos de Python han sido escritos y documentados utilizando como referencia los estándares propuestos por esa organización[1,2]. Aún más, se espera que este escrito sea un recurso útil para la utilización de la biblioteca, la cual se encuentra disponible en un repositiorio de GitLab.

Además, como la biblioteca *ROSLibPy* hace uso de WebSockets, es posible correr la simulación en un servidor y el algoritmo de control en un equipo remoto. Esto resulta práctico para su uso en educación, pues un estudiante no requiere una computadora potente ni instalar y configurar múltiples bibliotecas, tan solo necesita Python, *RosLibPy* y una conexión estable con el servidor.

[1] https://peps.python.org/pep-0008/
[2] https://peps.python.org/pep-0257/

Bibliografía

[Arbinolo, 2020] Arbinolo, F. (2020). *Modelling and Control of a Skid-Steering Mobile Robot for Indoor Trajectory Tracking Applications*. PhD thesis, Politecnico di Torino.

[Åström and Wittenmark, 2013] Åström, K. J. and Wittenmark, B. (2013). *Computer-controlled systems: theory and design*. Courier Corporation.

[Caracciolo et al., 1999] Caracciolo, L., De Luca, A., and Iannitti, S. (1999). Trajectory tracking control of a four-wheel differentially driven mobile robot. In *Proceedings 1999 IEEE international conference on robotics and automation (Cat. No. 99CH36288C)*, volume 4, pages 2632–2638. IEEE.

[Dorf and Bishop, 2008] Dorf, R. C. and Bishop, R. H. (2008). *Modern control systems*. Pearson Prentice Hall.

[Ermacora et al., 2020] Ermacora, G., Sartori, D., Rovasenda, M., Pei, L., and Yu, W. (2020). An evaluation framework to assess autonomous navigation linked to environment complexity. In *2020 IEEE International Conference on Mechatronics and Automation (ICMA)*, pages 1803–1810. IEEE.

[Koenig and Howard, 2004] Koenig, N. and Howard, A. (2004). Design and use paradigms for gazebo, an open-source multi-robot simulator. In *2004 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)(IEEE Cat. No. 04CH37566)*, volume 3, pages 2149–2154. IEEE.

[Kozłowski and Pazderski, 2004] Kozłowski, K. and Pazderski, D. (2004). Modeling and control of a 4-wheel skid-steering mobile robot. *International journal of applied mathematics and computer science*, 14(4):477–496.

[Mandow et al., 2007] Mandow, A., Martinez, J. L., Morales, J., Blanco, J. L., Garcia-Cerezo, A., and Gonzalez, J. (2007). Experimental kinematics for wheeled skid-steer

mobile robots. In *2007 IEEE/RSJ international conference on intelligent robots and systems*, pages 1222–1227. IEEE.

[Marton Juhasz, 2020] Marton Juhasz, G. G. (2020). Motor simulation plugins for gazebo - ros. `https://github.com/nilseuropa/gazebo_ros_motors`.

[MathWorks, 2022] MathWorks (2022). Choose an ODE Solver. `https://www.mathworks.com/help/matlab/math/choose-an-ode-solver.html`.

[Michel, 2004] Michel, O. (2004). Cyberbotics ltd. webots™: professional mobile robot simulation. *International Journal of Advanced Robotic Systems*, 1(1):5.

[Ortiz and Wirth, 2021] Ortiz, F. and Wirth, L. (2021). Desarrollo e implementación del sistema de control de bajo nivel de un robot 4wd para ambientes frutícolas. *Proyecto Integrador Profesional en curso, Ingeniería Electrónica, Universidad Nacional del Comahue.*

[PIN, 2020] PIN (2020). Año PIN 04/I257. Proyecto de Investigación: Sistemas de visión, robótica y navegación autónoma de vehículos para la fruticultura digital, 2020-2023. Universidad Nacional del Comahue. Director / codirector: Ing. Darío Mendieta / Dr. Marcelo Moreyra.

[Robotics, 2013] Robotics, O. (2013). Gazebo repository. `https://github.com/osrf/gazebo/tree/master/worlds`.

[Robotics, 2022] Robotics, O. (2022). Ros tutorials. `http://wiki.ros.org/ROS/Tutorials/`.

[Rohmer et al., 2013] Rohmer, E., Singh, S., and Freese, M. C. (2013). A versatile and scalable robot simulation framework. In *Proceedings of the International Conference on Intelligent Robots and Systems (IROS).*

[Scaglia et al., 2009] Scaglia, G., Montoya, L. Q., Mut, V., and di Sciascio, F. (2009). Numerical methods based controller design for mobile robots. *Robotica*, 27(2):269–279.

[Wilson, 2013] Wilson, D. (2013). Texasinstruments - teaching your pi controller to behave. `https://e2e.ti.com/blogs_/b/industrial_strength/posts/teaching-your-pi-controller-to-behave-part-vii`.

Capítulo 9

Anexo

9.1. Seguimiento de trayectoria basado en series de Taylor

En este anexo se presentan los cálculos, simplificaciones y reemplazos necesarios para obtener las ecuaciónes de control de Scaglia por desarrollo en series de Taylor, para un robot diferencial ideal.

Por comodidad, en la Ecuación (9.1) se presentan nuevamente las ecuaciones del modelo cinemático del robot diferencial, que serán linealizadas aplicando el primer método descrito en la sección anterior. Para mayor legibilidad, se reemplazaron las variables de control v_x y ω_z por V y W, respectivamente. Se busca calcular las señales de control V_n y W_n que generen que el vehículo pase de la posición (x_n, y_n) a la posición deseada $(x_{d_{n+1}}, y_{d_{n+1}})$ en el siguiente instante de muestreo, donde (x_{d_n}, y_{d_n}) son las coordenadas espaciales de la trayectoria de referencia para el instante n.

$$\begin{aligned} \dot{x} &= V\cos\theta \\ \dot{y} &= V\sin\theta \\ \dot{\theta} &= W \end{aligned} \tag{9.1}$$

Si se aproxima x, y y θ utilizando un polinomio de Taylor de orden 1, se obtiene:

$$x_{n+1} = x_n + \dot{x}_n T_0 \tag{9.2}$$

$$y_{n+1} = y_n + \dot{y}_n T_0 \tag{9.3}$$

$$\theta_{n+1} = \theta_n + \dot{\theta}_n T_0 \tag{9.4}$$

despejando $\dot{x_n}$, $\dot{y_n}$ y $\dot{\theta_n}$, e igualando con la Ecuación (9.1) resulta:

$$\dot{x}_n = \frac{x_{n+1} - x_n}{T_0} = V_n \cos\theta_n \tag{9.5}$$

$$\dot{y}_n = \frac{y_{n+1} - y_n}{T_0} = V_n \sin\theta_n \tag{9.6}$$

$$\dot{\theta}_n = \frac{\theta_{n+1} - \theta_n}{T_0} = W_n \tag{9.7}$$

El siguiente paso del método implica reemplazar los valores futuros de los estados por los valores de referencia, es decir x_{n+1} por $x_{d_{n+1}}$ e y_{n+1} por $y_{d_{n+1}}$. Al mismo tiempo, en la Ecuación (9.5) se reemplaza θ_n por θ_{d_n}, que representa la orientación necesaria en el instante n para que el robot alcance la posición deseada en $n+1$. De esta forma, se obtiene:

$$\frac{x_{d_{n+1}} - x_n}{T_0} = V_n \cos\theta_{d_n} \tag{9.8}$$

$$\frac{y_{d_{n+1}} - y_n}{T_0} = V_n \sin\theta_{d_n} \tag{9.9}$$

Combinando ambas ecuaciones en (9.8), se despeja θ_{d_n}:

$$\tan\theta_{d_n} = \frac{y_{d_{n+1}} - y_n}{x_{d_{n+1}} - x_n} \Rightarrow \theta_{d_n} = \arctan\frac{y_{d_{n+1}} - y_n}{x_{d_{n+1}} - x_n}. \tag{9.10}$$

De la Ecuación (9.8) también se desprende:

$$V_n(\cos^2\theta_{d_n} + \sin^2\theta_{d_n}) = \frac{x_{d_{n+1}} - x_n}{T_0}\cos\theta_{d_n} + \frac{y_{d_{n+1}} - y_n}{T_0}\sin\theta_{d_n}$$

$$V_n = \frac{x_{d_{n+1}} - x_n}{T_0}\cos\theta_{d_n} + \frac{y_{d_{n+1}} - y_n}{T_0}\sin\theta_{d_n}$$

Por otro lado, para obtener W_n, en la Ecuación (9.5) se reemplaza θ_{n+1} por θ_{d_n}, dado por la Ecuación (9.10). De esta forma, las señales de control pueden escribirse como sige:

$$V_n = \frac{x_{d_{n+1}} - x_n}{T_0}\cos\theta_{d_n} + \frac{y_{d_{n+1}} - y_n}{T_0}\sin\theta_{d_n} \tag{9.11a}$$

$$W_n = \frac{\theta_{d_n} - \theta_n}{T_0} \tag{9.11b}$$

Una característica de la estrategia de control determinada por la Ecuación (9.11) es que carece de parámetros que permitan realizar un ajuste fino del comportamiento del robot. Para solventar esto, Scaglia propone una función de coste J que pondera la energía del error de seguimiento y la energía de las derivadas de las variables de estado. Luego, se calcula V_n y W_n que minimizan el coste en el instante n, igualando a cero las derivadas parciales de J, es decir $\frac{\delta J}{\delta V} = 0$ y $\frac{\delta J}{\delta W} = 0$. A continuación se presentan los cálculos correspondientes.

$$J_n = k_1^2[(x_{d_{n+1}} - x_{n+1})^2 + (y_{d_{n+1}} - y_{n+1})^2] + k_2^2(\dot{x}_n^2 + \dot{y}_n^2) + k_3^2(\theta_{d_n} - \theta_{n+1})^2 + k_4^2\dot{\theta}_n^2 \tag{9.12}$$

Reemplazando las ecuaciones (9.2) y (9.1) en (9.12), aplicando derivadas parciales e igualando a cero:

$$\begin{aligned} J_n =& k_1^2[(x_{d_{n+1}} - x_n - V_n \cos\theta_n T_0)^2 + (y_{d_{n+1}} - y_n - V_n \sin\theta_n T_0)^2] + \\ & k_2^2[(V_n \cos\theta_n)^2 + (V_n \sin\theta_n)^2) + k_3^2(\theta_{d_n} - \theta_n - W_n T_0)^2 + k_4^2 W_n^2 \\ \frac{\delta J_n}{\delta V_n} =& k_1^2[2(x_{d_{n+1}} - x_n - V_n \cos\theta_n T_0)\cos\theta_n T_0 + \\ & 2(y_{d_{n+1}} - y_n - V_n \sin\theta_n T_0)\sin\theta_n T_0] + k_2^2 2V_n = 0 \\ \frac{\delta J_n}{\delta W_n} =& k_3^2 2(\theta_{d_n} - \theta_n - W_n T_0)T_0 + k_4^2 2W_n = 0 \end{aligned} \tag{9.13}$$

Finalmente, se despeja V_n y W_n obteniendo la siguiente estrategia de control:

$$V_n = k_v^2 \left\{ \frac{x_{d_{n+1}} - x_n}{T_0} \cos\theta_{d_n} + \frac{y_{d_{n+1}} - y_n}{T_0} \sin\theta_{d_n} \right\} \tag{9.14a}$$

$$W_n = k_w^2 \frac{\theta_{d_n} - \theta_n}{T_0} \tag{9.14b}$$

donde

$$k_v^2 = \frac{k_1^2}{k_1^2 + k_2^2/T_0^2} \tag{9.15a}$$

$$k_w^2 = \frac{k_3^2}{k_3^2 + k_4^2/T_0^2} \tag{9.15b}$$

Puede observarse que k_v^2 y k_w^2 pertenecen al intervalo real $[0;\ 1]$. Si se toma $k_i^2 = 0$ para $i = 1, 2, 3, 4$ se anula la influencia de los términos correspondientes de la función de coste. En particular, si k_1^2 y/o k_3^2 se anulan, también lo hacen las señales de control V y/o W, por lo cual el sistema diverge. Por otro lado, si se anula k_2^2 y/o k_4^2 se deja de penalizar la energía de las señales de control V y/o W, por lo que el sistema puede presentar oscilaciones indeseables. Aún más, vale remarcar que no reviste importancia el valor exacto que tomen las constantes $k_{1,2,3,4}$, sino la relación entre ellas dada por las Ecuaciones (9.15a) y (9.15b).

En este trabajo, se optó por utilizar $k_v^2 = k_w^2 = 0.4$, que se corresponde con dar un peso aproximadamente 10 veces mayor a la energía del error de posición y orientación respecto de la energía de las variables de control, es decir $k_1^2 \approx 10k_2^2$ y $k_3^2 \approx 10k_4^2$.