Instituto Tecnológico y de Estudios Superiores de Monterrey

Implementación de robótica inteligente (Gpo 501)

Análisis de controladores

Daniel Kaled Bernal Ayala A01750047

24 de Mayo del 2025

En el presente documento se abordarán los temas de controladores para el puzzlebot. Se explicará su funcionamiento y cómo estos funcionan en simulación tanto en simulink como en ROS. Se tomarán los temas relacionados a los controladores, haciendo una investigación de uno de estos. A continuación se presenta el desarrollo en clase para implementar el movimiento del robot hacia un punto deseado.

Para la simulación en simulink se elaboraron los diferentes bloques que conforman al puzzlebot como lo es, la hackerboard, Jetson y la física del robot. Además se utiliza la odometría para poder conocer la posición y orientación del robot en todo momento.

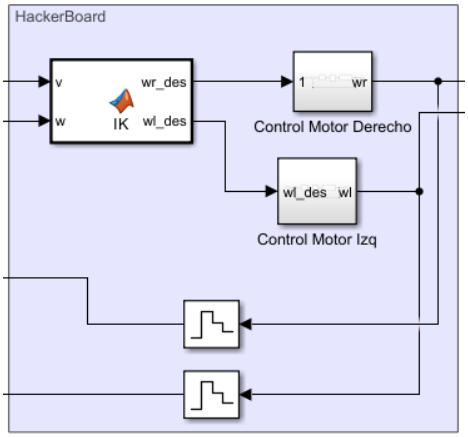


Imagen 1. Bloque representativo de la Hackerboard para su simulación.

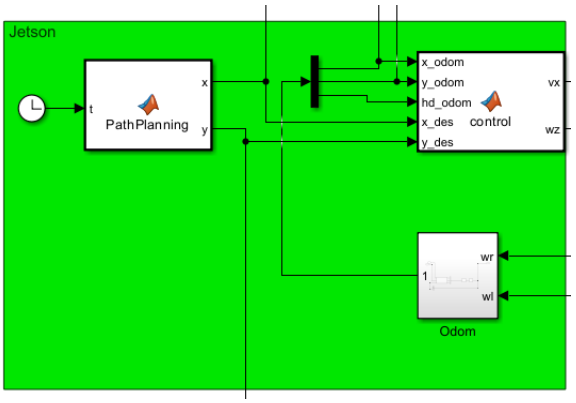


Imagen 2. Bloque representativo de la Jetson para su simulación.

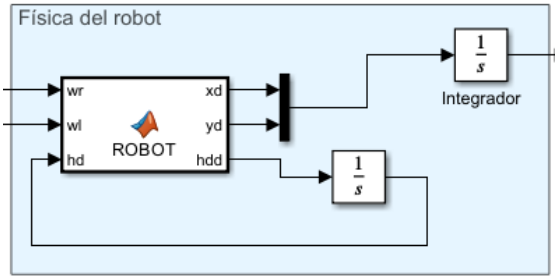


Imagen 3. Bloque representativo de la física del robot para su simulación.

Empezaremos con el bloque de la Jetson (Imagen 2), en la cual se encuentra el bloque de la odometría. Se puede decir que la odometría es un sensor matemático que nos permite estimar la pose de un robot, en este caso el puzzlebot. Esta utiliza la misma información de sus sensores como lo son los encoders de los motores. En el bloque de **odometría** tenemos la cinemática del robot donde obtenemos las posiciones en **x, y** y el **heading** dada por la letra griega ϕ que indican la orientación. El proceso de obtención es encontrar la cinemática que describe al puzzlebot, para ello se consideran las siguientes expresiones.

1. $xd = w_r + w_l \cdot \frac{R}{2} \cdot Cos\phi$
2. $yd = w_r + w_l \cdot \frac{R}{2} \cdot Sin\phi$
3. $\phi d = w_r - w_l \cdot \frac{R}{b}$

Con esto definido pasamos al bloque de control, que será el encargado de comparar la odometría con el punto deseado para alcanzar la posición dada en el path planning. Dentro de este bloque de control se utilizan ganancias proporcionales para mandar la velocidad tanto lineal como angular a la Hackerboard.

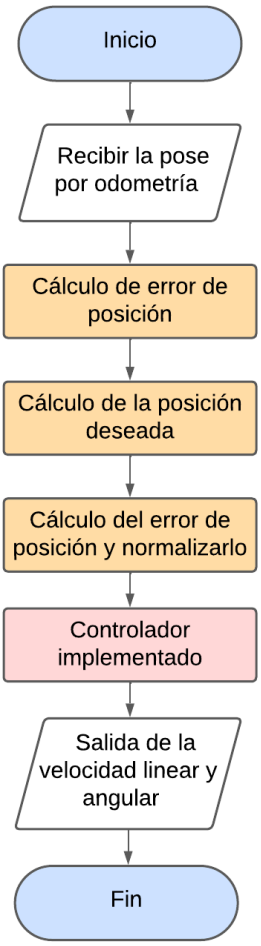


Imagen 4. Diagrama de flujo del bloque de control

Más adelante se explicarán las modificaciones necesarias en el diagrama dentro del bloque de **controlador implementado** para poder realizar los controladores mencionados anteriormente. Aquí podemos observar que el **error de posición** entre dos puntos en un plano bidimensional (2D) se calcula con la posición actual del robot dadas por la odometría y el punto deseado dado por el path planning. Ahora bien, para el **error de orientación** primero obtenemos nuestro heading deseado, el cual, es la función que calcula el ángulo entre la línea que une los puntos con respecto al eje de las x.

Después se hace el cálculo del error para después normalizar en un rango de $[-\pi, \pi]$ para que este gire de manera correcta hacia la izquierda o derecha dependiendo el error. Las velocidades lineal y angular del robot se calculan con un controlador proporcional, donde cada una depende del error correspondiente (posición u orientación). Para mantener la estabilidad del sistema y prevenir movimientos excesivos, ambas velocidades se limitan mediante saturación a valores máximos y mínimos predefinidos.

Una vez elaborado esto, las velocidades obtenidas las recibirá nuestra Hackerboard (Imagen 1). En donde tendremos nuestra cinemática inversa y encontraremos que mandar a nuestro motor derecho y motor izquierdo. Aquí tomamos en cuenta nuestro radio de la llanta y la distancia del eje entre rueda y rueda. Por lo que obtenemos las siguientes ecuaciones que describen las velocidades angulares de cada llanta.

4. $$W_{r_{des}} = \frac{V}{R} + \frac{w \cdot b}{2R}$$
5. $$W_{l_{des}} = \frac{V}{R} - \frac{w \cdot b}{2R}$$

Con esto, le damos las velocidades angulares de cada llanta para que se realice el respectivo control a cada uno. De igual forma, estas velocidades son requeridas para hacer el cálculo de la odometría como se mencionó anteriormente. Finalmente se dan las velocidades al robot para que este se mueva dentro de la simulación en el bloque de la física (Imagen 3).

Turn & Go

Simulink

Para este controlador, necesitamos establecer condiciones para realizar el movimiento deseado. Para ello se presenta el siguiente agregado al código del controlador.

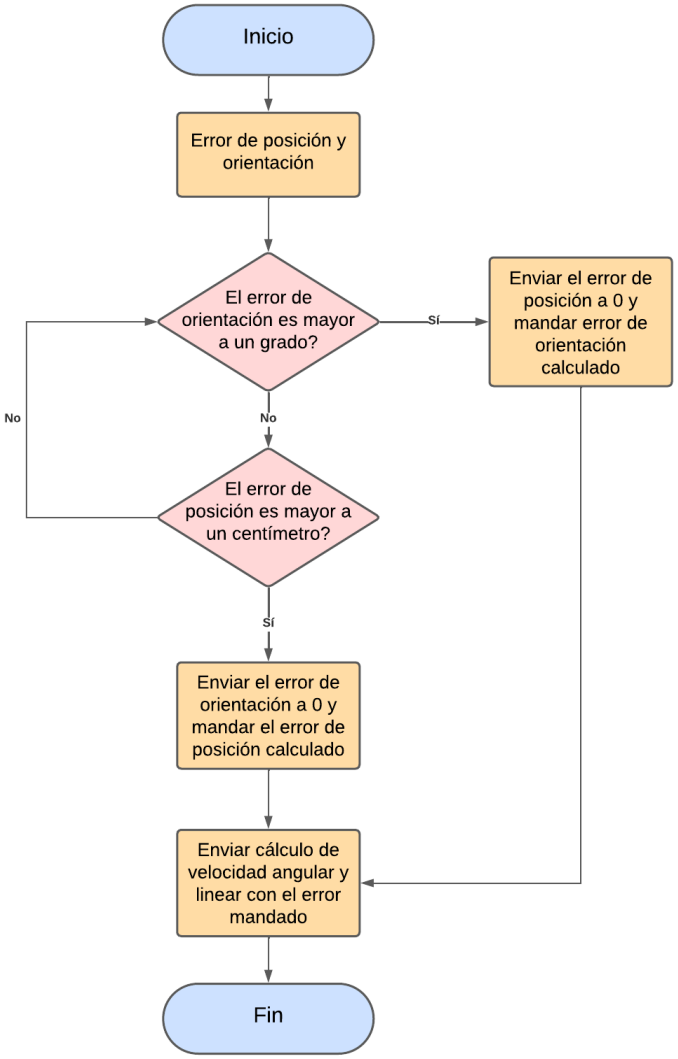


Imagen 5. Diagrama de flujo del controlador Turn & Go

Aquí observamos que si el error normalizado es mayor a 1 grado, el error de posición mandará 0 haciendo que no avance el robot y se concentre solo en el giro. Para el segundo condicional encontramos que si el error de posición es mayor a 1 centímetro, este mandará 0 al error de posición, haciendo que solo avance. Haciendo que llegue al punto marcado por el path planning.

Turtlesim

Para llevar una simulación más visual en el puzzlebot, utilizamos turtlesim en ROS que nos permite observar en tiempo real el movimiento simulado de nuestro robot. Para esto podemos observar nuestro diagrama a bloques para ver el sentido de la simulación.

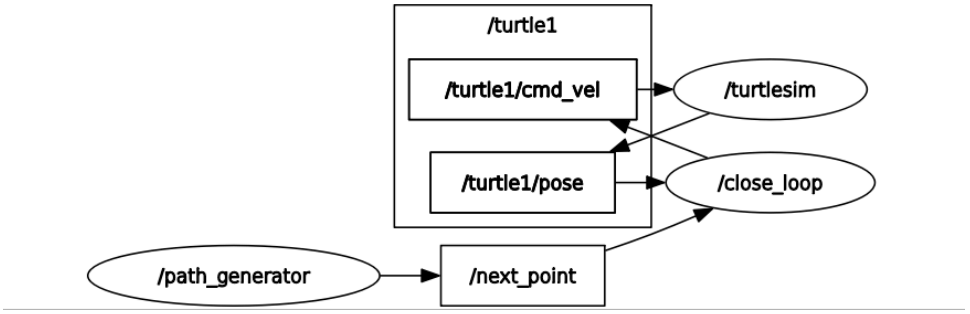


Imagen 6. Diagrama a bloques de los nodos en ROS.

Aquí observamos la conexión de los nodos en donde nuestro controlador llamado **/close_loop** es el encargado de realizar el movimiento del simulador. En este caso fue turtlesim donde obtenemos su pose y el punto hacia el cual nos dirigimos. Publicando así las velocidades para el movimiento del robot.

Turn while Go

Simulink

Para el siguiente controlador, se modifica el condicional para poder avanzar mientras se corrige la orientación. Ahora se presenta el condicional que realiza el controlador.

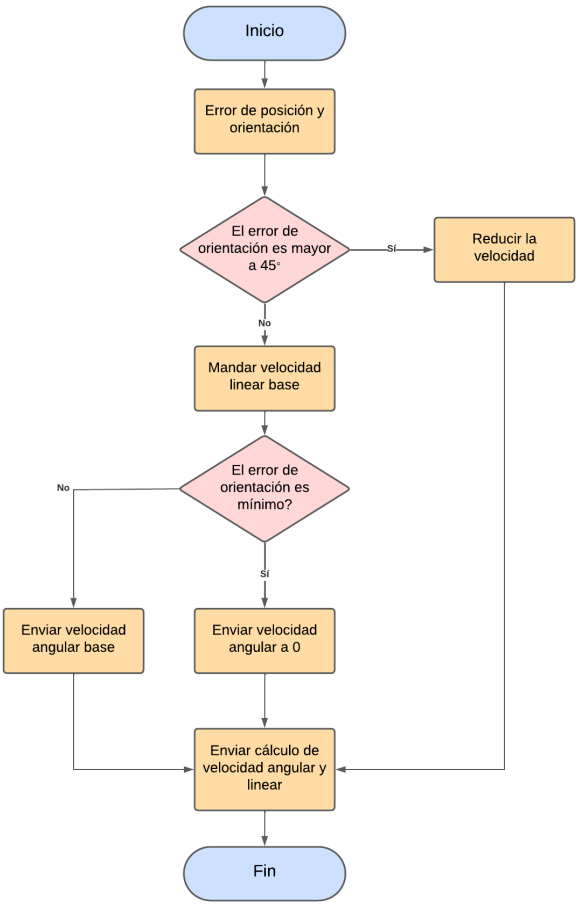


Imagen 7. Diagrama de flujo de controlador Turn While Go

Para este controlador, el robot ajusta su trayectoria mediante un control proporcional que calcula tanto la **velocidad lineal** como la **velocidad angular** en función de los errores de posición y orientación. La velocidad lineal se puede reducir adaptativamente si el error de orientación es considerable, pero **no se anula por completo**, permitiendo así que el robot avance lentamente mientras se orienta corrigiendo su posición y orientación. De igual manera, para la implementación en ROS se mantuvo el mismo principio del diagrama de flujo en donde la velocidad lineal no será cero y corregirá el movimiento mientras avanza.

Controladores investigados

Lyapunov-Based Controller

El diseño de control basado en Lyapunov es un método para lograr analizar y diseñar controladores para sistemas no lineales. Se pueden utilizar las funciones de Lyapunov para demostrar la estabilidad sin resolver ecuaciones complejas, lo que lo hace versátil para varias aplicaciones. Su objetivo es proyectar una ley de control que pueda garantizar la estabilidad o una estabilidad asintótica del sistema de bucle cerrado mediante la construcción de una función Lyapunov adecuada. Los controladores que están basados en Lyapunov ofrecen el enfoque sistemático para la estabilización de los sistemas no lineales (IEEE, 2020).

El método consiste en:

1. Proponer la función de Lyapunov escalar positiva $V_{(x)}$, que usualmente representa una *energía* general del sistema.
2. Verificar que su derivada temporal $V_{(x)}$ sea negativa, lo cual implica que el sistema pierde energía con el tiempo y se estabiliza.

Entonces, el sistema está estable en el sentido de Lyapunov y tiende a un equilibrio.

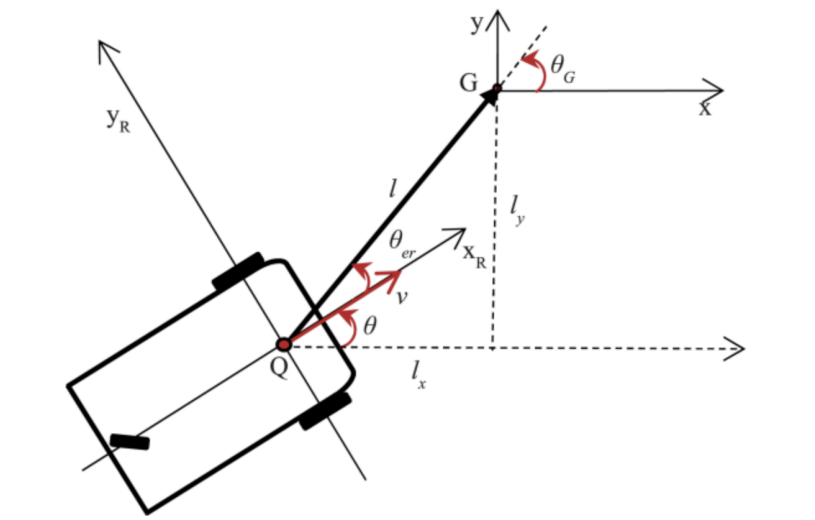


Imagen 8. Una geometría de un robot móvil con tracción diferencial de 3 ruedas se mueve hacia un objetivo obtenido de IEEE (2020).

Basándonos en la función de Lyapunov, la velocidad lineal y la velocidad angular del *DDWMR* se calculan de la siguiente manera:

$$6.\ v\ =\ v_{ri} cos\Theta_{er}$$

v_r: velocidad de referencia del robot.

Θ_{er}: error angular respecto al objetivo.

Esta fórmula ajusta la velocidad lineal para alinearse con la dirección deseada. Si el error angular es grande, $cos\Theta_{er}$ será pequeño y por lo tanto el robot reducirá su avance hasta estar correctamente orientado (Fiveable, 2024).

$$7.\ w\ =\ k_1\Theta_{er} + \frac{v_r cos\Theta_{er} sin\Theta_{er}(\Theta_{er} + k_2\Theta_G)}{\Theta_{er}}$$

- ➔ w: velocidad angular.
- ➔ Θ_{er}: error de orientación.
- ➔ Θ_G: Orientación deseada.
- ➔ $k_1,\ k_2$: ganancias de control.

Esta ecuación ajusta la velocidad angular considerando tanto el error angular actual como la interacción entre la velocidad de avance y la desviación en el ángulo. Tiene dos términos:

- El primero, $k_1\Theta_{er}$, es un control proporcional.
- El segundo término permite una corrección más compleja basada en la dinámica del movimiento y ayuda a garantizar la estabilidad del sistema, siguiendo la función de Lyapunov.

Desarrollo del código para ROS y Matlab

Para la integración del controlador, se modifica la función para adaptar las ecuaciones descritas anteriormente. A continuación se muestra el diagrama del funcionamiento de la implementación del controlador.

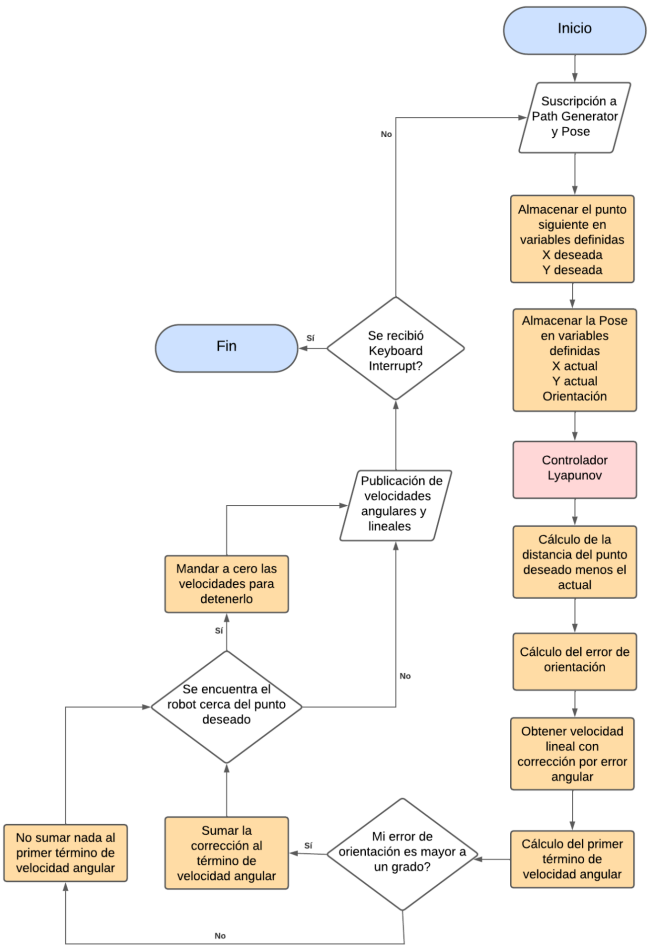


Imagen 9. Diagrama de flujo para el nodo de ROS para el controlador de Lyapunov

En este caso el segundo término que describe la velocidad angular del controlador lo tomamos como la corrección que genera para seguir alcanzando la trayectoria o el waypoint. Cabe mencionar que la velocidad referencia marcada en la fórmula de la velocidad angular la marcamos como 0.2 que es la máxima velocidad lineal que soporta nuestro puzzlebot.

Pure Pursuit

El controlador de pure pursuit es un método de dirección automática calculando la velocidad angular necesaria para que un robot se mantenga en trayectorias. En este controlador se asume que la velocidad lineal es constante. Por lo que se necesita considerar un controlador de velocidad angular como lo es un controlador proporcional en este caso. El algoritmo básico consiste en usar un bucle for para recorrer cada par de puntos y determinar si existen intersecciones dentro de cada segmento de línea. Si se encuentran puntos de destino, se sigue el más adecuado. Esto quiere decir que el vehículo puede ir de reversa para hacer la ruta más adecuada y rápida (Matlab, 2025).

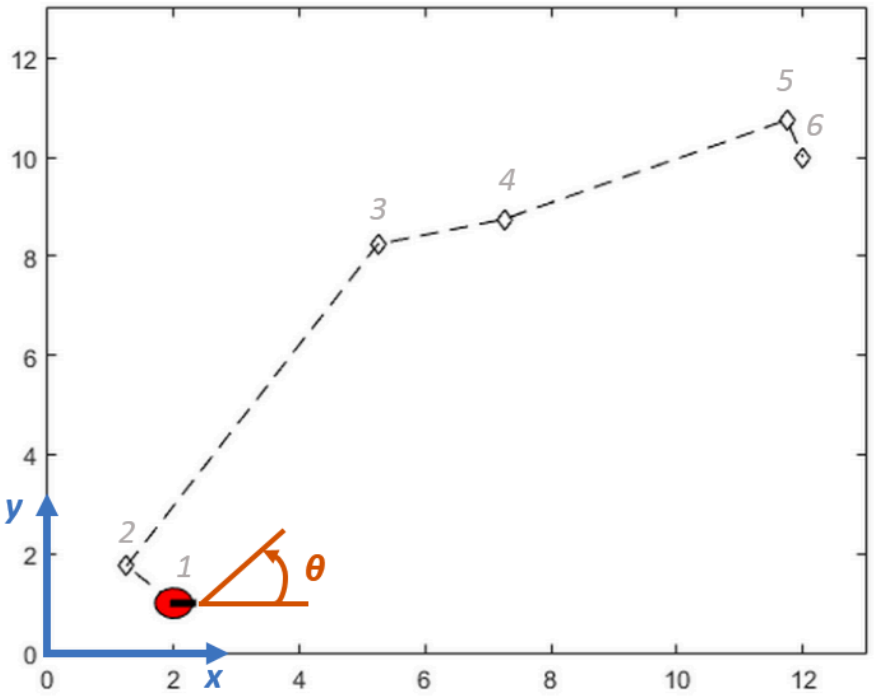


Imagen 10. Ilustrativa del funcionamiento del controlador pure pursuit

Este es un controlador geométrico que es ideal para seguir caminos. Su dependencia viene del lookahead que es una distancia objetivo de donde se marca. A partir de la proyección donde se encuentra se calcula una curvatura llamada *k* que el robot deberá seguir para alcanzar, siguiendo un arco de circunferencia. La selección del lookahead es importante para el desempeño del controlador por lo cual se debe de calibrar de buena manera para realizar la acción que sea necesaria.

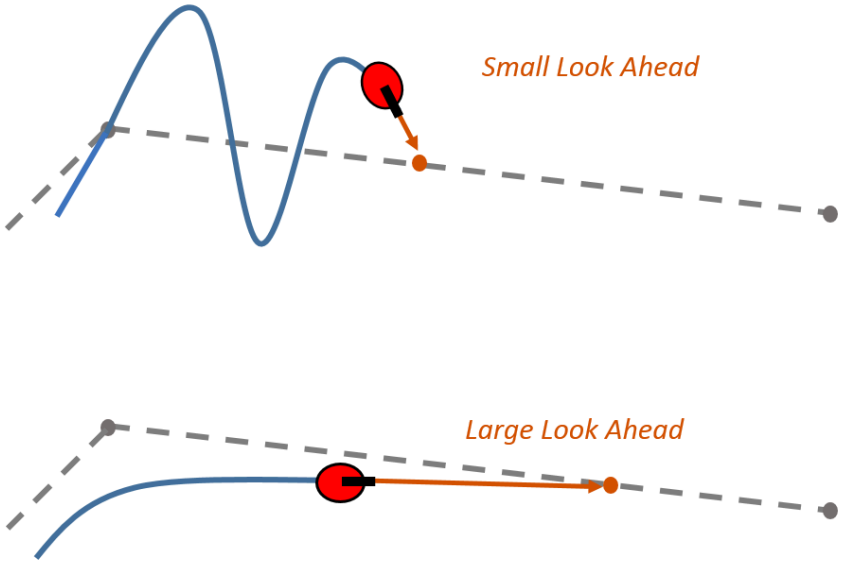


Imagen 11. Lookahead pequeño vs Lookahead grande.

De igual manera esto va de la mano con las velocidades angulares y lineales haciendo que sea una **proyección del vector objetivo** sobre el eje **y local** del robot, dividida entre la distancia de anticipación. Es decir, qué tanto debe **girar** el robot para alcanzar el punto objetivo en esa circunferencia definida por la siguiente ecuación.

$$8.\ \omega = v \cdot \kappa = \frac{2vy_r}{L^2}$$

- ➔ k: Curvatura necesaria para alcanzar el punto.
- ➔ y_r: Error lateral de giro
- ➔ L: Lookahead definido

Así definimos la velocidad angular para mandar al robot y que se anticipe al punto, disminuyendo o aumentando la velocidad lineal según sea el caso (Xiang, 2025).

Desarrollo del código para ROS y Matlab

Para la integración en ROS se supone que la velocidad lineal es constante, por lo tanto, puede cambiar la velocidad lineal del robot en cualquier punto. Esto se define con la siguiente ecuación.

$$9.\ Vx = Dx \cdot cos(\theta) + Dy \cdot sin(\theta)$$

- ➔ Dx: Diferencia de distancia entre *x* deseada y *x* actual.
- ➔ Dy: Diferencia de distancia entre *y* deseada y *y* actual.
- ➔ θ: Orientación actual del robot

En donde decimos que la velocidad lineal será adaptativa dependiendo del punto deseado. Esto quiere decir que si el objetivo se encuentra directamente frente al robot entonces nuestra Vx será grande y positiva para alcanzar el punto. Si se encuentra a un lado o atrás, Vx será baja haciendo que se priorice primero el girar antes que avanzar.

Ahora bien, para nuestra velocidad angular queremos saber qué tanto debe de girar el robot para perseguir el punto. Entonces tenemos que hacer la proyección del eje lateral patria poder saber la curvatura

$$10.\ Wz = \frac{1}{L}(Dy \cdot cos(\theta) - Dx \cdot sin(\theta))$$

El error lateral de giro (y_r) se calcula con la operación dentro del paréntesis, la cual indica que tanto se debe de girar el robot. Haciendo que lleguemos a la ecuación simplificada de:

$$11.\ \omega = \frac{y_r}{L}$$

Ya que nosotros estimamos la velocidad y así lograríamos obtener la velocidad angular para nuestro controlador. Permitiendo igualmente un giro orientado hacia la dirección de menor recorrido. Esto quiere decir que:

- Si el punto está a la izquierda, entonces el robot gira a la izquierda. $y_r > 0$ entonces $\omega > 0$
- Si el punto está a la derecha, entonces el robot gira a la derecha. $y_r < 0$ entonces $\omega < 0$

La implementación del código en ROS se explica en el siguiente diagrama de flujo.

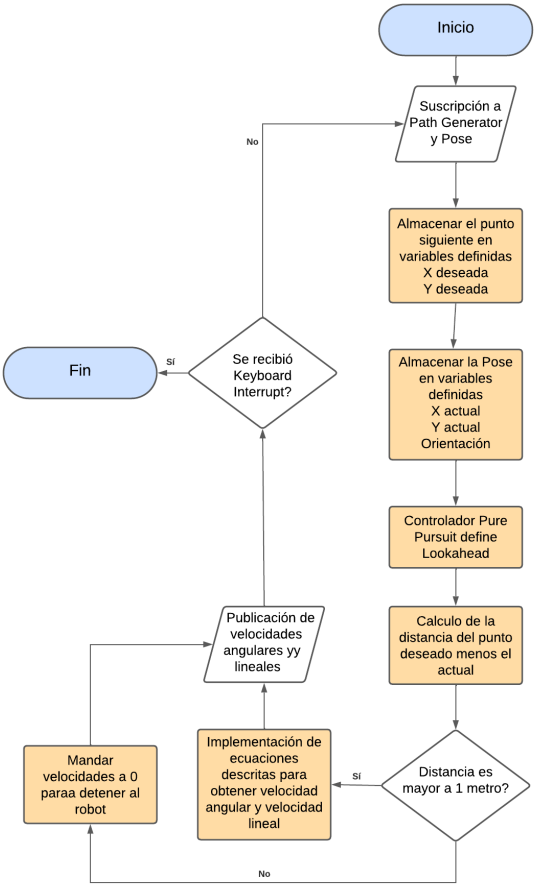


Imagen 12. Diagrama de flujo de la implementación del controlador Pure Pursuit.

Cabe mencionar que las velocidades tanto angulares y lineales se saturan haciendo la aproximación en las simulaciones de las velocidades angulares que maneja nuestro puzzlebot. El lookahead definido funcionará para cambiar el comportamiento del robot, haciendo que la persecución del punto sea cercana o lejana dependiendo del objetivo. Finalmente, si ya se llega al final de la trayectoria o waypoints no se mueva.

Calibración de controladores

Se realizaron calibraciones en los controladores para ajustar las ganancias y así poder manejar una velocidad tanto angular como lineal dentro de los parámetros del puzzlebot. Esto para tener una comparativa de resultados más claras en las pruebas que se realizarán más adelante. De igual forma, mantener las velocidades en los rangos de funcionamiento del puzzlebot, nos permite integrarlos a él y así poder seleccionarlos dependiendo del objetivo que buscamos. A continuación se dejan en claro las velocidades máximas que se alcanzarán en ambas velocidades.

- Para Vx el rango va de $0.0\ a\ 0.2$
- Para Wz el rango va de $0.0\ a\ 0.6$

Turn then Go y Turn While Go

Para estos dos controladores, dentro del código utilizan una Kv y Kw para poder ajustar las velocidades durante el funcionamiento del puzzlebot. Para ello el ajuste de las ganancias se haría de la siguiente manera.

$$12.\ Wz = error_{orientación} \cdot Kw$$

Si realizamos el despeje para nuestra ganancia y conocemos que en nuestras condiciones el error máximo que se puede alcanzar es de π entonces podemos decir que:

$$13.\ Kw = \frac{Wz}{error_{orientación}} \approx 1.191$$

Para nuestra ganancia la dejaremos en 1 haciendo que se mantenga nuestra velocidad de 0.2 en todo momento.

$$14.\ Kl = 1$$

Lyapunov

Para este tercer controlador solo nos enfocaremos en la parte de la velocidad angular donde afecta al primer y segundo término de la ecuación 7. Para ello modificamos nuestra ganancia 1 que multiplica al error de orientación. Siguiendo que nuestro máximo error es igual a π entonces nuestra ganancia tendría un valor de:

$$15.\ Kw \approx 1.191$$

Y si consideramos un error de 3 radianes sería que nuestra $K1$ alcance el valor de 2.0.

Pure pursuit

Para este controlador lo que se realizó fue saturar ambas velocidades a los rangos permitidos, ya que la velocidad angular está dada por el eje lateral proyectado. Por lo que saturamos las velocidades para obtener igualdad de condiciones para los controladores.

Prueba 1: Cuadrantes

Para esta primera prueba se colocó el robot en la posición inicial $x = 0,\ y = 0,\ \Theta = 0$. El objetivo será llevarlo a un punto meta distinto para cada cuadrante (I, II, III, IV). Es por esto que se realizó un path planning el cual no dependiera del tiempo. Siendo que exista un nuevo suscriptor que reciba el estado del controlador, haciendo que si es

que ya terminó el punto, publique el segundo punto. Para el caso del nodo controlador cambia para mantener un cálculo del tiempo en cuanto reciba el punto. El tiempo empieza a correr en cuanto tiene el punto para poder asegurarnos de evaluar los controladores. Para la primera prueba se aplicarán los siguientes criterios de evaluación para los controladores.

- Tiempo promedio de llegada a cada punto por controlador.
- Error promedio al alcanzar cada punto.

Los cuatro puntos a recibir el robot será a una distancia de 2 metros cada uno para la visita de cada cuadrante.

Tabla I. Waypoints definidos.

Puntos	Coordenadas (x,y)
1	(7.5, 5.5)
2	(7.5, 7.5)
3	(7.5, 5.5)
4	(5.5, 5.5)

A continuación se muestran los comportamientos en simulación del robot.

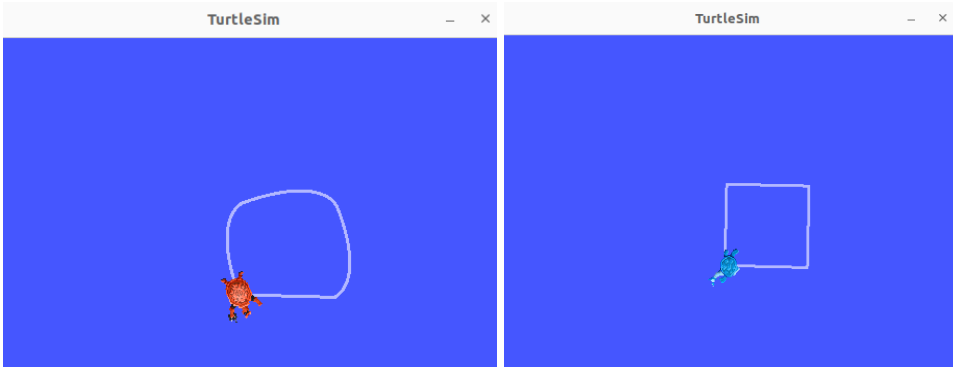


Imagen 13 y 14. Simulación en turtlesim del controlador Turn while Go y Turn & Go.

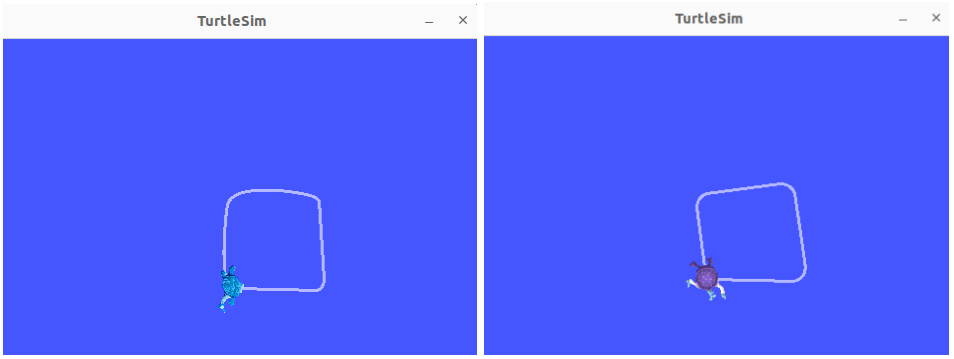


Imagen 15 y 16. Simulación en turtlesim del controlador Lyapunov y Pure pursuit.

Tabla II. Resultados para cada waypoint.

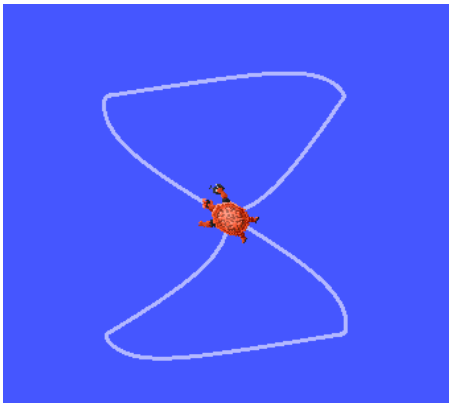
Punto	Controlador	Tiempo de llegada a cada punto (s)	Error al alcanzar un punto (m)
1	**Turn & Go**	**9.68**	**0.036**
2	**Turn & Go**	**13.78**	**0.042**
3	**Turn & Go**	**13.78**	**0.038**
4	**Turn & Go**	**13.78**	**0.039**
1	Turn while Go	9.67	0.026
2	Turn while Go	11.47	0.051
3	Turn while Go	10.9	0.038
4	Turn while Go	10.9	0.045
1	**Lyapunov**	**11.92**	**0.048**
2	**Lyapunov**	**13.52**	**0.049**
3	**Lyapunov**	**14.12**	**0.049**
4	**Lyapunov**	**12.72**	**0.05**
1	Pure pursuit	10.14	0.05
2	Pure pursuit	11.44	0.04
3	Pure pursuit	11.14	0.045
4	Pure pursuit	11.14	0.05

Se realizó una segunda prueba en donde se aumentaron 2 waypoints más con la intención de realizar movimientos en diagonal para el robot. De igual forma se buscó ir a los diferentes cuadrantes alrededor del punto inicial del robot para poder tener más resultados y tener un mejor análisis.

Tabla III. Waypoints definidos para la segunda prueba.

Puntos	Coordenadas (x,y)
1	(7.5, 7.5)
2	(3.5, 7.5)
3	(5.5, 5.5)
4	(3.5, 3.5)
5	(7.5, 3.5)
6	(5.5, 5.5)

Estos waypoints nos darán los siguientes comportamientos.



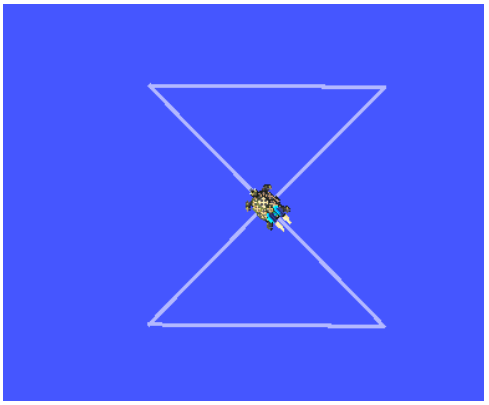
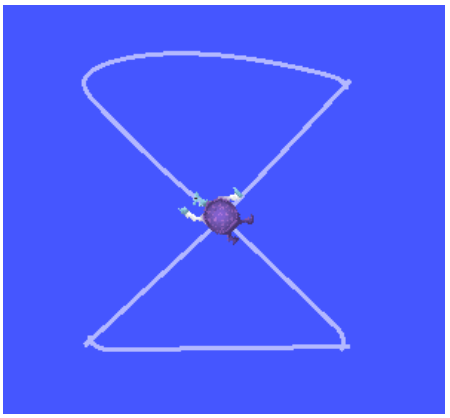


Imagen 17 y 18. Simulación en turtlesim del controlador Turn while Go y Turn & Go para la segunda prueba de waypoints.



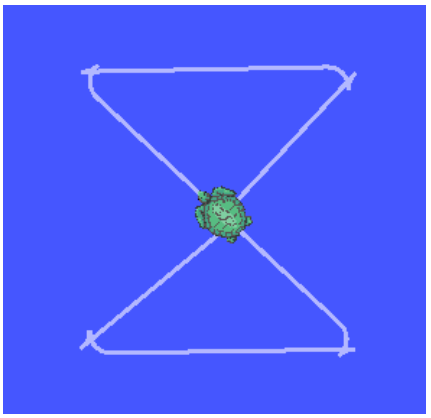


Imagen 19 y 20. Simulación en turtlesim del controlador Lyapunov y Pure pursuit para la segunda prueba de waypoints.

Tabla IV. Resultados para cada waypoint (segunda prueba).

Punto	Controlador	Tiempo de llegada a cada punto (s)	Error al alcanzar un punto (m)
1	**Turn & Go**	**16.88**	**0.046**
2	**Turn & Go**	**24.08**	**0.048**
3	**Turn & Go**	**18.28**	**0.036**
4	**Turn & Go**	**17.88**	**0.048**
5	**Turn & Go**	**24.08**	**0.046**
6	**Turn & Go**	**18.28**	**0.037**
1	Turn while Go	14.07	0.044
2	Turn while Go	22.07	0.033
3	Turn while Go	16.27	0.033
4	Turn while Go	15.87	0.032
5	Turn while Go	22.47	0.043
6	Turn while Go	16.17	0.048
1	**Lyapunov**	**16.20**	**0.048**
2	**Lyapunov**	**25.71**	**0.049**
3	**Lyapunov**	**17.51**	**0.05**
4	**Lyapunov**	**17.91**	**0.05**
5	**Lyapunov**	**24.41**	**0.048**
6	**Lyapunov**	**18.51**	**0.048**
1	Pure pursuit	14.35	0.048
2	Pure pursuit	21.85	0.049
3	Pure pursuit	16.05	0.046
4	Pure pursuit	15.55	0.046
5	Pure pursuit	21.95	0.045
6	Pure pursuit	16.05	0.047

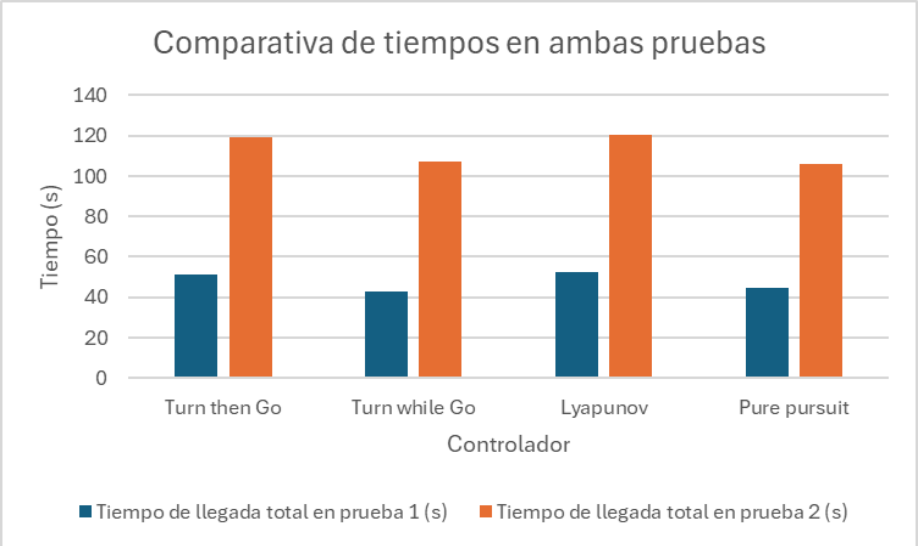
Análisis de resultados de Prueba 1 (Waypoints):

Después de realizar los dos experimentos podemos obtener una tabla con el tiempo total que tardó el robot en recorrer los waypoints mandados. Al igual que el promedio del error que tuvo al llegar a los diferentes puntos. Esto es de suma importancia, ya que al ser waypoints buscamos que lleguen a las coordenadas solicitadas. Cabe mencionar que a todos los controladores se les pide parar 5 centímetros del punto objetivo. Así que para asegurarnos de una buena precisión para los waypoints nuestro mejor controlador para la exactitud de la coordenada será el que se encuentre su promedio cercano a los 5 centímetros solicitados. A continuación se muestra una tabla con los valores obtenidos.

Tabla V. Tabla de resultados generales.

Controlador	Tiempo de llegada total en prueba 1 (s)	Error promedio en prueba 1 (m)	Tiempo de llegada total en prueba 2 (s)	Error promedio en prueba 2 (m)
Turn then Go	**51.02**	**0.03875**	**119.48**	**0.044**
Turn while Go	42.94	0.04	106.92	0.039
Lyapunov	**52.28**	**0.049**	**120.25**	**0.049**
Pure pursuit	44.64	0.04625	105.8	0.047

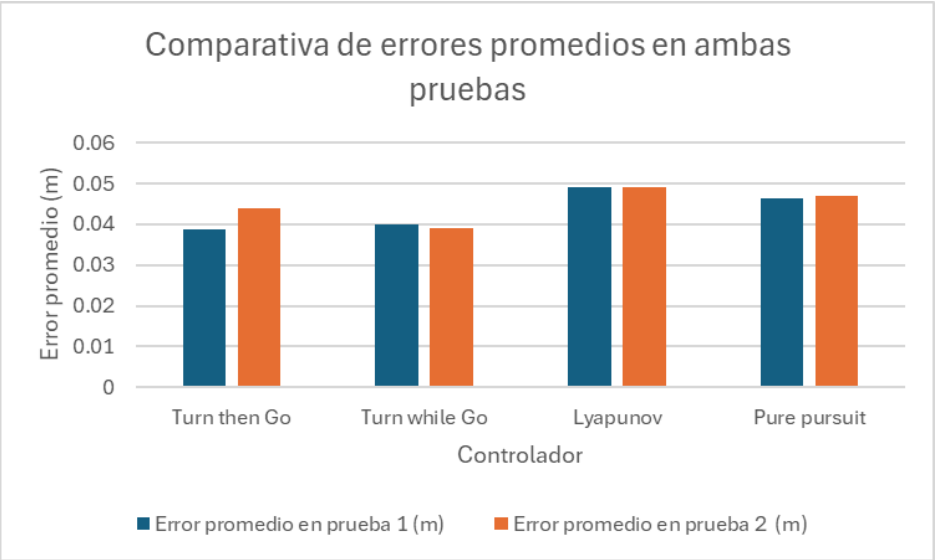
Con esto elaborado se realizaron gráficas para un mejor entendimiento de los resultados obtenidos.



Gráfica 1. Comparativa de tiempos en ambas pruebas

Tomando en cuenta el tiempo en el que se realizó cada lista de waypoints se nota una gran diferencia en la **gráfica 1** entre el tiempo del controlador **Turn while Go** ante todos los

controladores. Seguido por el controlador **Pure pursuit, Turn then Go y Lyapunov** para la primera prueba. Esto cambia para la segunda prueba, ya que **Pure pursuit** y **Turn while Go** se asemejan en los tiempos.



Gráfica 2. Comparativa de errores en ambas pruebas

Ahora bien si lo unimos con los errores promedios obtenidos, se observa que **Lyapunov** fue el mejor método en ambas pruebas en cuanto a la exactitud en el punto como se muestra en la **gráfica 2**, manejando un error del **0.49** que prácticamente cumple con lo planteado en el código. De igual manera, **Pure pursuit** maneja un error cercano a la posición marcada en el código.

Con esto podemos concluir que para la parte de Waypoints el que mejor balance tiene en cuanto al tiempo y error es el controlador **Pure Pursuit** que realiza lasa pistas en un tiempo similar al **Turn while Go,** pero en la parte del error, este se encuentra más cercano a la posición que deseamos con los Waypoints. Si queremos un controlador rápido y con el menor error posible, **Pure pursuit** es nuestra mejor opción. De igual forma si buscamos la mejor precisión y no dependemos del tiempo para completar las pruebas, **Lyapunov** nos ayuda a cumplir con esto de muy buena manera.

Prueba 2: Trayectoria continua

Para esta segunda prueba se diseñarán 3 trayectorias que seguirá cada controlador para poder analizar los siguientes criterios: El error promedio que quiere decir que tan cerca estuvo el robot de la trayectoria y el error máximo que es la máxima desviación que tuvo el robot en algún instante de la trayectoria. De igual manera, cuando el robot finaliza el seguimiento de la trayectoria es importante saber que tan cerca se quedó del final real de la trayectoria. Estas tres trayectorias incluirán, curvas amplias, curvas cerradas y esquinas pronunciadas. A continuación se muestran las trayectorias a realizar, estas se realizan en un minuto para poder tener un estándar en cada una y así asegurarnos de resultados más precisos.

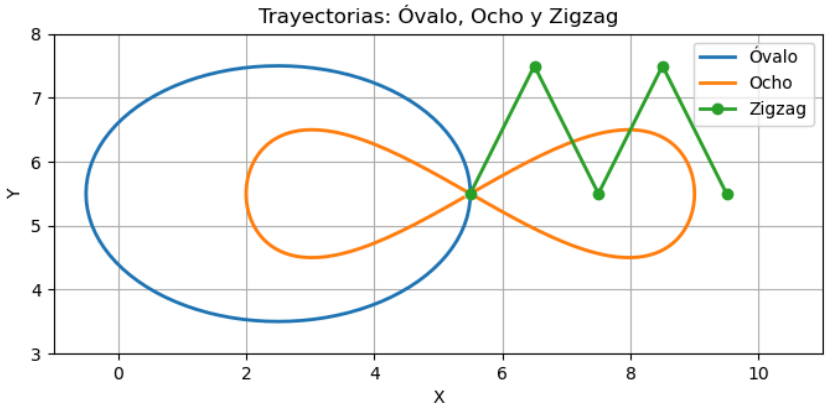


Imagen 21. Trayectorias a realizar por los diferentes controladores (Óvalo, ocho y Zig Zag).

Con las trayectorias mostradas, pasaremos al desempeño de los controladores. A continuación se muestran las imágenes de las trayectorias seguidas por el robot con los diferentes controladores.

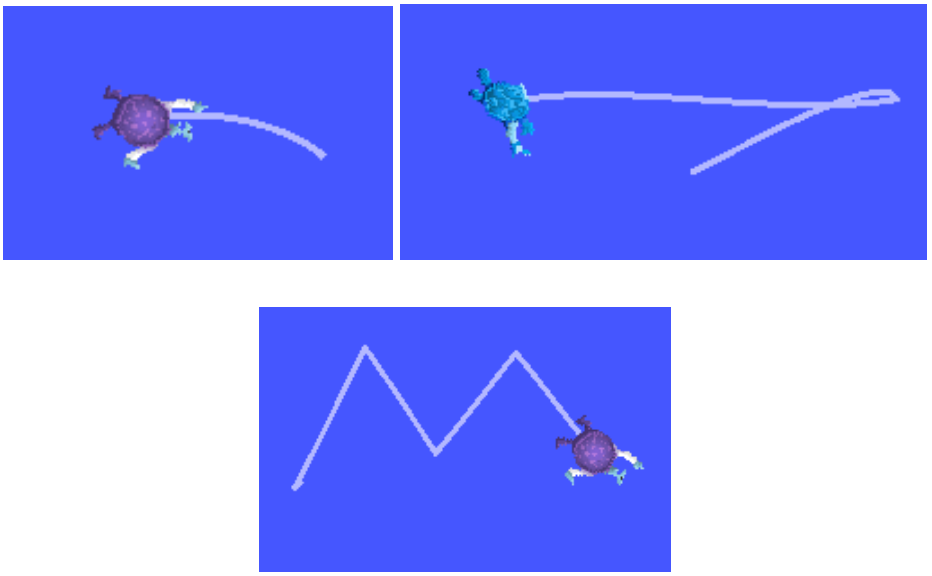
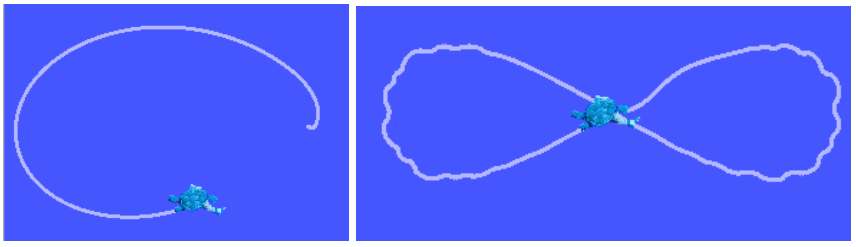


Imagen 22. Trayectorias realizadas por el controlador Turn & Go.



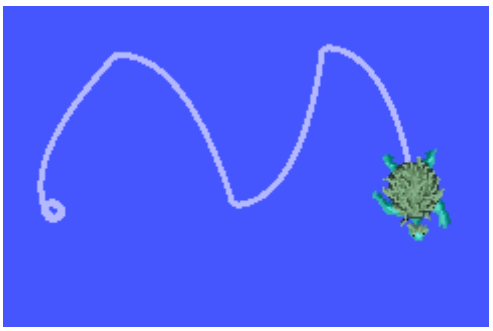


Imagen 23. Trayectorias realizadas por el controlador Turn while Go.

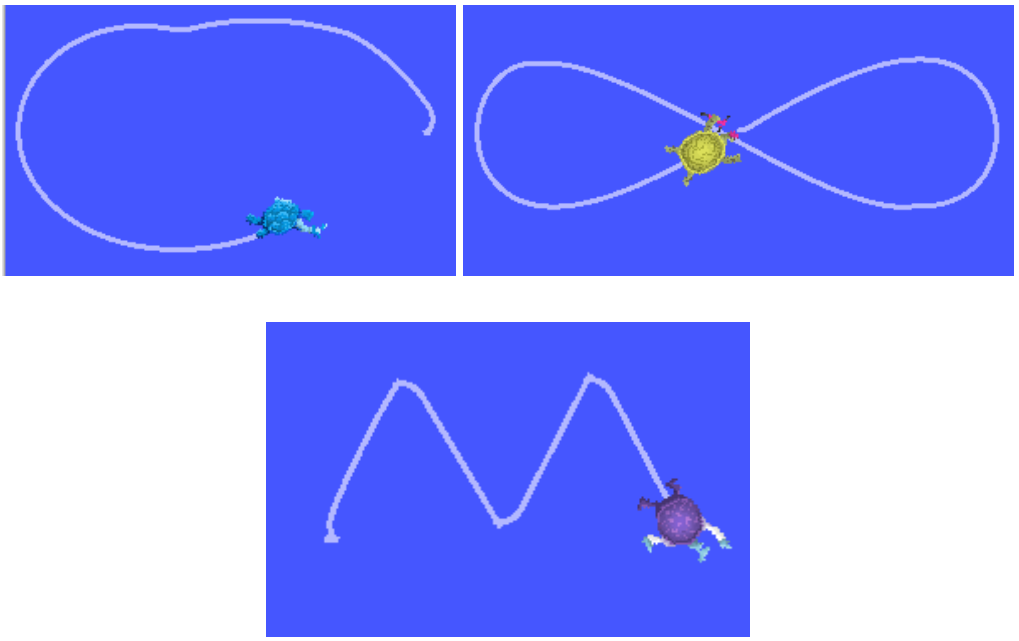


Imagen 24. Trayectorias realizadas por el controlador Lyapunov.

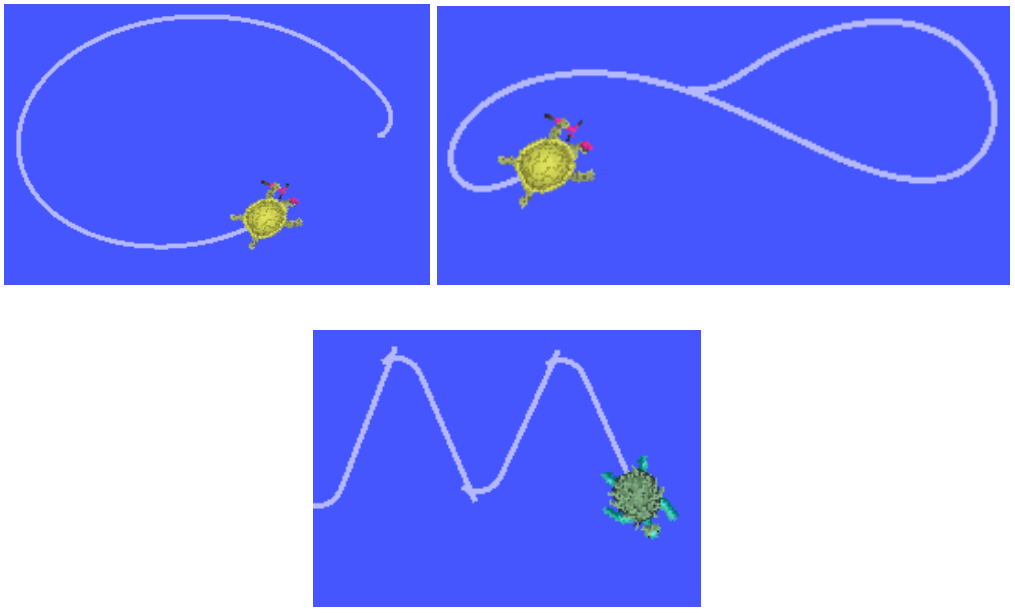


Imagen 25. Trayectorias realizadas por el controlador Pure pursuit.

A continuación, se plantea la tabla de resultados con los criterios mencionados anteriormente. Pero primero definiremos cómo se obtuvieron y qué funcionalidad tendrán para seleccionar al mejor controlador en trayectorias.

Cálculo del error promedio, error máximo y error de distancia

Se suman los errores instantáneos al llegar a cada punto discretizado por el tiempo. Donde se puede explicar con la siguiente ecuación.

$$16.\ \bar{e} = \frac{1}{N}\sum_{i=1}^{N} e_i$$

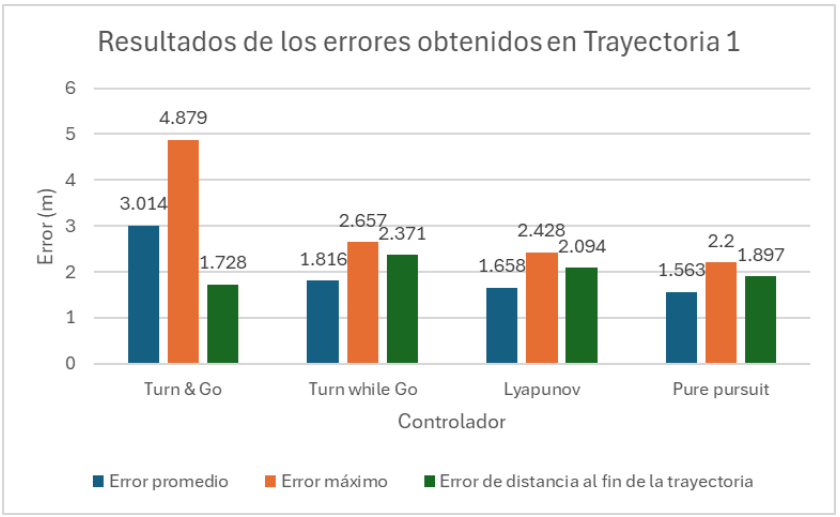
Donde N se interpreta como el número total de mediciones. En donde se obtendrá el valor promedio de los errores en la trayectoria, así podemos definir que tan cerca estuvo el robot de esta. El error máximo nos dará el valor en el cual nuestro robot se alejó más siendo así un parámetro valioso para tener un comportamiento del seguimiento de la trayectoria.
Finalmente con el error de distancia solo buscamos saber a cuanto se quedó nuestro robot para completar la trayectoria definida.

Tabla VI. Resultados de cada trayectoria realizada.

Trayectoria	Controlador	Error promedio	Error máximo	Error de distancia
Óvalo	**Turn & Go**	**3.014**	**4.879**	**1.728**
Ocho	**Turn & Go**	**1.747**	**3.131**	**2.027**
Zig Zag	**Turn & Go**	**1.234**	**2.248**	**0.503**
Óvalo	Turn while Go	1.816	2.657	2.371
Ocho	Turn while Go	0.063	0.266	0.016
Zig Zag	Turn while Go	1.129	2.306	0.155
Óvalo	**Lyapunov**	**1.658**	**2.428**	**2.094**
Ocho	**Lyapunov**	**0.292**	**0.583**	**0.365**
Zig Zag	**Lyapunov**	**1.086**	**2.236**	**0.212**
Óvalo	Pure pursuit	1.563	2.2	1.897
Ocho	Pure pursuit	1.410	2.970	1.519
Zig Zag	Pure pursuit	0.991	2.204	0.117

Análisis de resultados de Prueba 2 (Trayectorias):

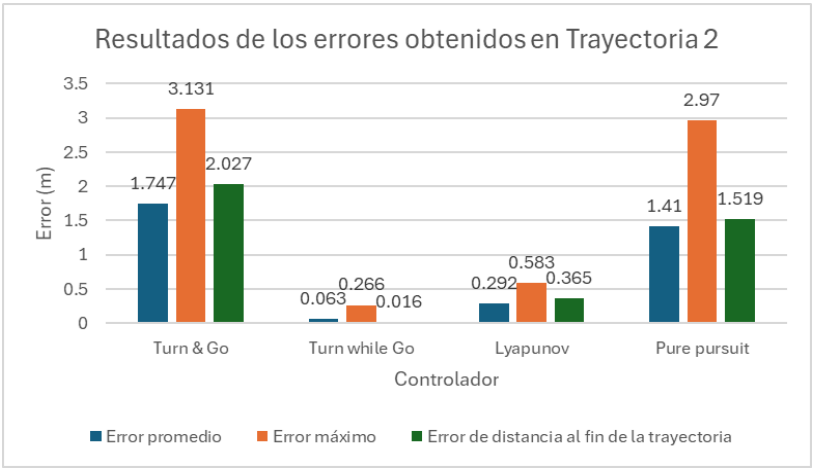
Con esto realizado pasaremos a graficar los comportamientos de cada controlador en las trayectorias definidas.



Gráfica 3. Comparativa de la trayectoria 1.

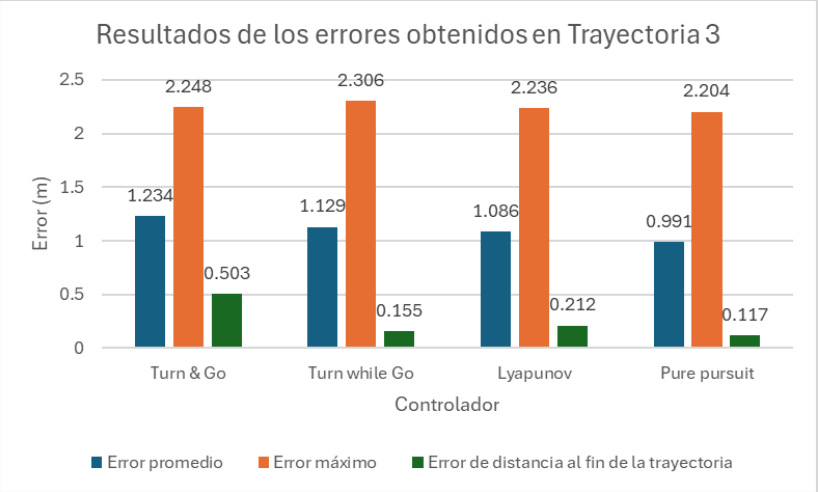
Para la primera trayectoria tenemos curvas cerradas debido al óvalo que al momento de dar el giro se debe corregir rápidamente para no perder los puntos. El desempeño de los controladores en la **gráfica 3** nos muestra que el peor controlador fue el **Turn & Go** debido a que muestra errores muy grandes, además de mencionar que en la **imagen 22** observamos que no realizó la trayectoria.

Para los 3 controladores siguientes, se mostró un comportamiento similar, ya que los tres controladores son buenos para realizar vueltas. Sin embargo, podemos destacar que el controlador **Pure pursuit** tuvo el mejor desempeño debido a la comparación de errores en la **gráfica 3.** Esto también lo podemos observar en la **imagen 25** que nos muestra un mejor trazo de la trayectoria ovalada.



Gráfica 4. Comparativa de la trayectoria 2.

Para esta segunda trayectoria observamos complicaciones con los controladores **Turn & Go** y **Pure pursuit.** Por un lado el primer controlador mencionado queda descartado debido a que tampoco realizó la trayectoria planteada. Para el segundo controlador si sorprende en el desempeño, mostrando errores muy altos y complicaciones al completar el ocho. Siendo así que no concluyó la trayectoria como se aprecia en la **imagen 25.** Observando que para esta segunda trayectoria nuestra mejor opción es el **Turn While Go** a pesar de no tener una corrección de mayor complejidad como lo es el controlador implementado de **Lyapunov.** Sin embargo, al observar las simulaciones, **Turn while Go** empieza a tener correcciones innecesarias haciendo que el robot se vea con oscilaciones. Algo que no sucede con **Lyapunov** que va corrigiendo de mejor manera y manteniendo un buen ritmo para realizar la trayectoria de mejor manera.



Gráfica 5. Comparativa de la trayectoria 3.

Para esta última trayectoria se vieron resultados similares. Aquí vemos como todos los controladores terminaron la trayectoria, pero, como se pueden ver en las **imágenes 23, 24 y 25** controladores como **Turn while Go** y **Pure pursuit** muestran mucha curvatura en las zonas rectas de la trayectoria. Por lo cual, el que tiene el mejor desempeño es el controlador implementado de **Lyapunov** que realizó de muy buena forma las correcciones y teniendo un mejor desempeño.

En conclusión para esta segunda prueba el controlador con mejor desempeño en las tres trayectorias fue el controlador implementado de **Lyapunov.** En la primera prueba el error entre controladores son centímetros, entonces podemos decir que el destacado fue **Lyapunov** por su mejor desempeño en la trayectoria 2, donde las correcciones fueron suaves y realizaban de mejor manera la figura.

Conclusión

Para finalizar, el controlador que mejor se desempeñó tanto en **Waypoints** como **Trayectorias** fue **Lyapunov**, ya que sus correcciones tan ligeras hacían que su comportamiento se viera similar. Es verdad que en la primera prueba el controlador **Pure Pursuit**. Sin embargo, este no completó algunas trayectorias volviéndose lento sorpresivamente. Así que podemos decir que si buscamos un comportamiento amigable, en donde la corrección sea ligera y sea constante, **Lyapunov** es nuestra mejor opción. Si estás buscando velocidad pero donde se puedan generar movimientos bruscos. **Pure pursuit** es la mejor opción.

A continuación se muestra una comparativa de los cuatro controladores integrados después de las pruebas realizadas:

Tabla VII. Comparativa:Turn & Go vs **Turn While Go vs Lyapunov vs Pure pursuit**

Característica	Turn While Go	Turn & Go	Lyapunov	Pure pursuit
Tipo de movimiento	Girar y avanzar al mismo tiempo, básico	Gira completamente, luego avanza	Girar y avanzar suavemente	Avanza siguiendo puntos de una trayectoria objetivo
Suavidad del movimiento	Suave en general	Movimiento brusco, por etapas	Muy suave, trayectoria óptima	Suave si la trayectoria es amigable, puede ser brusco si no lo es
Reacción ante errores	Moderada	Baja	Alta	Moderada-alta, depende del radio de giro y anticipación
Fácil de implementar	Sí	Muy fácil	Requiere análisis y diseño no lineal	Requiere cálculo geométrico del punto de mira

Se deja el link al repositorio para los códigos desarrollados en ROS y el Simulink:https://github.com/A01750047/An-lisis-de-controladores

Referencias

Fiveable. (2024, August 20). 10.4 Lyapunov-based control – Control Theory. https://library.fiveable.me/control-theory/unit-10/lyapunov-based-control/study-guide/YrYwbb2R8rn9wrtj

IEEE. (2020). Energy Comparison of Controllers Used for a Differential Drive Wheeled Mobile Robot. Recuperado de https://ieeexplore.ieee.org/document/9193941

Matlab. (2025). Controlador Pure Pursuit. Recuperado de https://la.mathworks.com/help/nav/ug/pure-pursuit-controller.html

Sarah, X. (2025). Basic Pure Pursuit. Recuperado de https://wiki.purduesigbots.com/software/control-algorithms/basic-pure-pursuit