Universidad de Alcalá

TRABAJO FIN DE GRADO

GRADO EN INGENIERÍA DE COMPUTADORES

ESCUELA POLITÉCNICA SUPERIOR

Planificación y control de trayectorias para vehículos autónomos en carreras utilizando Aprendizaje por Refuerzo aplicadas al robot Turtlebot 4.

Autor: Nicolás Carminati

Tutor: Rafael Barea Navarro

Departamento de Electrónica

2026

UNIVERSIDAD DE ALCALÁ
ESCUELA POLITÉCNICA SUPERIOR

Grado en Ingeniería de Computadores

Trabajo de Fin de Grado

Planificación y control de trayectorias para vehículos autónomos en carreras utilizando Aprendizaje por Refuerzo aplicadas al robot Turtlebot 4.

Autor: Nicolás Carminati

Tutor: Rafael Barea Navarro

Tribunal:

Presidente: Pastor Mendoza, Julio

Vocal: Llamazares Llamazares, Ángel

Suplente: Martín Arguedas, Carlos Julián

2026

"Nada es particularmente difícil si lo divides en pequeños trabajos."

Henry Ford

Agradecimientos

Este trabajo es el resultado de muchas horas de dedicación e investigación a lo largo del último semestre. Ha sido un camino largo, exigente y, en los momentos cuando no salían las cosas, un poco estresante. Pero todo ello ha sido parte del proceso de aprendizaje, demostrando que el esfuerzo, tarde o temprano, da sus frutos. En primer lugar, quiero agradecer a mis padres, por brindarme durante todos estos años la mejor educación que me podrían haber dado, por estar siempre a mi lado y creer siempre en mí. Luego me gustaría agradecer a mi novia, Elena, que, a pesar de no haber estado involucrada en este trabajo, ha sido, y seguirá siendo, un pilar fundamental en mi vida.

Por último, me gustaría agradecer a mi tutor, Rafael Barea Navarro, quien me ha guiado y ayudado desde el principio; y a todos los chicos del grupo RobeSafe, con quienes he compartido las mejores horas de descanso, haciendo que este camino haya sido mucho más llevadero y acompañado.

Índice general

Índice de figuras ..... **xi**
Índice de fragmentos de código ..... **xiv**
Resumen ..... **xv**
Abstract ..... **xvii**
Glosario (Lista de acrónimos) ..... **xix**
1. Introducción **1**
1.1 Objetivos 1
1.2 Arquitectura de los sistemas de conducción autónoma 2
1.2.1 Módulo de percepción 3
1.2.2 Módulo de planificación 3
1.2.3 Módulo de control 3
2. Marco teórico **6**
2.1 Algoritmo Pure Pursuit 6
2.2 MPC 7
2.3 LQR 11
2.4 Aprendizaje por refuerzo 13
2.4.1 Proceso de Decisión de Markov 14
2.4.2 Familias RL 15
2.4.3 Funciones de recompensas 18
2.4.4 DDPG 20
2.5 Algoritmo ArUco 24
3. Arquitectura general del sistema **27**
3.1 MATLAB 27
3.2 MVSim 28
3.3 Mapeado del entorno 30
3.4 Modelo del TurtleBot 4 Lite 31
4. Planificación de trayectorias **35**
4.1 Generación de waypoints 35
4.2 Interpolación lineal 39
4.3 Splines 39
4.4 Optimización de trayectorias 42

5. Control 50

5.1 Implementación MPC 50
5.2 Implementación LQR 54
5.3 Implementación Pure Pursuit 56
5.4 Implementación DDPG 57

5.4.1 Creación del entorno 57
5.4.2 Step 60
5.4.3 Creación de la recompensa 62
5.4.4 Inicialización de episodios y el olvido catastrófico 66
5.4.5 Creación y arquitectura del agente DDPG 69
5.4.6 Resultados del entrenamiento y convergencia del agente 71
5.4.7 Despliegue del agente 73

6. Resultados 76

6.1 Resultados en simulación 78

6.1.1 Resultados DDPG 78
6.1.2 Resultados Pure Pursuit 82
6.1.3 Resultados LQR 86
6.1.4 Resultados MPC 90
6.1.5 Comparación de algoritmos 94

6.2 Resultados en robot real 98

7. Conclusión y líneas futuras 104

7.1 Conclusiones 104
7.2 Líneas futuras 105

Apéndice A. Presupuesto 108

A.1. Costes de software y hardware 108
A.2. Costes de personal 109
A.3. Coste final del proyecto 109

Bibliografía 111

Índice de figuras

Figura 1.1: Esquema de bloques de la arquitectura general del sistema de navegación... 2
Figura 2.1: Algoritmo Pure Pursuit. [4] ... 6
Figura 2.2: Lookahead Pequeño. [4] ... 7
Figura 2.3: Lookahead Grande. [4] ... 7
Figura 2.4: Analogía de la conducción humana para explicar el principio del control predictivo. ... 8
Figura 2.5: Diagrama controlador MPC. [6] ... 9
Figura 2.6: Principio de funcionamiento del horizonte deslizante en un controlador MPC. ... 10
Figura 2.7: Ciclo del RL. Elaboración Propia. ... 14
Figura 2.8: Categorización de las técnicas de aprendizaje por refuerzo. Fuente: Brunton y Kutz (2022) [11]. ... 15
Figura 2.9: Comparativa temporal entre una señal de recompensa densa y una dispersa. Elaboración propia. ... 18
Figura 2.10: Ejemplo de una red actor μθ(s) en DDPG. Fuente: [14] ... 21
Figura 2.11: Ejemplo de una red crítico Q(s,a) en DDPG. Fuente: [14] ... 21
Figura 2.12: Familia ArUco 4x4 250. ... 24
Figura 2.13: Homografía de un plano. ... 25
Figura 2.14: Detección simultánea y estimación de la pose 3D de múltiples marcadores ArUco con sus respectivos identificadores (IDs). ... 25
Figura 3.1: Flujo de arquitectura del sistema. ... 27
Figura 3.2: Interfaz MATLAB 2024a. ... 28
Figura 3.3: Entorno MVSim. ... 29
Figura 3.4: Paneles de Información en MVSim. ... 29
Figura 3.5: Circuito 1 en MVSim. ... 30
Figura 3.6: Circuito 2 en MVSim. ... 30
Figura 3.7: Circuito 3 en MVSim. ... 31
Figura 3.8: Modelo cinemático diferencial del TurtleBot 4 y representación de los errores de trayectoria. ... 31
Figura 4.1: Waypoints obtenidos mediante distancias equiespaciadas para el circuito 1. ... 36
Figura 4.2: Códigos ArUco ordenados cronológicamente en las vallas. ... 36
Figura 4.3: Vallas del circuito a recorrer. ... 37
Figura 4.4: Marcador con ID nuevo. ... 38
Figura 4.5: Marcador con ID ya registrado. ... 38
Figura 4.6: Waypoints obtenidos de las vallas en sus coordenadas globales. ... 38

Figura 4.7: Comparación de trazadores de distinto orden. 39
Figura 4.8: Trayectoria del circuito 1 interpolada con Spline. 41
Figura 4.9: Trayectoria del circuito 2 interpolada con Spline. 41
Figura 4.10: Trayectoria del circuito 3 interpolada con Spline. 42
Figura 4.11: Trazada óptima mostrando el punto de giro, el ápex y la zona de salida (exterior-interior-exterior). [17] 43
Figura 4.12: Relación entre la variable de decisión α y los límites físicos del robot en la pista. 45
Figura 4.13: Representación del espacio libre restante cuando el centro del robot se sitúa en el límite derecho. 45
Figura 4.14: Curva cerrada, radio pequeño 46
Figura 4.15: Curva grande o recta, radio enorme 46
Figura 4.16: Zoom de la trayectoria ideal 47
Figura 4.17: Desplazamiento lateral a lo largo 47
Figura 4.18: Trayectoria ideal (roja), trayectoria central (azul) y límites (verde) del circuito 1. 47
Figura 4.19: Trayectoria ideal (roja), trayectoria central (azul) y límites (verde) del circuito 2. 48
Figura 4.20: Trayectoria ideal (roja), trayectoria central (azul) y límites (verde) del circuito 3. 48
Figura 5.1: Representación geométrica de la transformación de coordenadas globales a locales. 53
Figura 5.2: Representación geométrica de las observaciones del estado local inmediato. 58
Figura 5.3: Representación del vector de anticipación (lookahead). 59
Figura 5.4: Vectores de orientación de referencia que indican la orientación deseada en cada waypoint. 63
Figura 5.5: Sectores del circuito 1 a entrenar. 66
Figura 5.6: Entrenamiento exitoso del agente DDPG. 71
Figura 6.1: Vuelta del Circuito 1 con Agente DDPG. 78
Figura 6.2: Métricas DDPG circuito 1. 78
Figura 6.3: Vuelta del Circuito 1 Invertido con Agente DDPG. 79
Figura 6.4: 3 Vueltas Seguidas al Circuito 1. 79
Figura 6.5: Vuelta del Circuito 2 con Agente DDPG. 80
Figura 6.6: Métricas DDPG circuito 2. 80
Figura 6.7: Vuelta del Circuito 3 con Agente DDPG. 81
Figura 6.8: Métricas DDPG Circuito 3. 81
Figura 6.9: Vuelta Trayectoria Central Pure Pursuit Circuito 1. 82
Figura 6.10: Métricas Pure Pursuit Trayectoria Central Circuito 1. 82

Figura 6.11: Vuelta Trayectoria Óptima Pure Pursuit Circuito 1. 83
Figura 6.12: Métricas Pure Pursuit Trayectoria Óptima Circuito 1. 83
Figura 6.13: Vuelta Trayectoria Central Pure Pursuit Circuito 2. 84
Figura 6.14: Vuelta Trayectoria Central Pure Pursuit Circuito 3. 84
Figura 6.15: Vuelta Trayectoria Óptima Pure Pursuit Circuito 2. 85
Figura 6.16: Vuelta Trayectoria Óptima Pure Pursuit Circuito 3. 85
Figura 6.17: Vuelta Trayectoria Central LQR Circuito 1. 86
Figura 6.18: Métricas Trayectoria Central LQR Circuito 1. 86
Figura 6.19: Vuelta Trayectoria Óptima LQR Circuito 1. 87
Figura 6.20: Métricas Trayectoria Óptima LQR Circuito 1. 87
Figura 6.21: Vuelta Trayectoria Central LQR Circuito 2. 88
Figura 6.22: Vuelta Trayectoria Central LQR Circuito 3. 88
Figura 6.23: Vuelta Trayectoria Óptima LQR Circuito 2. 89
Figura 6.24: Vuelta Trayectoria Óptima LQR Circuito 3. 89
Figura 6.25: Vuelta Trayectoria Central MPC Circuito 1. 90
Figura 6.26: Métricas Trayectoria Central MPC Circuito 1. 90
Figura 6.27: Vuelta Trayectoria Óptima MPC Circuito 1. 91
Figura 6.28: Métricas Trayectoria Óptima MPC Circuito 1. 91
Figura 6.29: Vuelta Trayectoria Central MPC Circuito 2. 92
Figura 6.30: Vuelta Trayectoria Central MPC Circuito 3. 92
Figura 6.31: Vuelta Trayectoria Óptima MPC Circuito 2. 93
Figura 6.32: Vuelta Trayectoria Óptima MPC Circuito 3. 93
Figura 6.33: Circuito real. 98
Figura 6.34: Trayectoria del circuito real. 98
Figura 6.35: Vuelta MPC con circuito real. 99
Figura 6.36: Vuelta Pure Pursuit con circuito real. 99
Figura 6.37: Vuelta LQR con circuito real. 100
Figura 6.38: Vuelta DDPG con circuito real. 100

Índice de fragmentos de código

Listing 4.1: Spline 1 40
Listing 4.2: Spline 2 40
Listing 4.3: Spline 3 41
Listing 5.1: Configuración MPC 1. 51
Listing 5.2: Configuración MPC 2. 52
Listing 5.3: Configuración MPC 3. 53
Listing 5.4: Configuración LQR 1. 54
Listing 5.5: Configuración LQR 2. 55
Listing 5.6: Configuración Pure Pursuit. 56
Listing 5.7: Configuración step DDPG. 62
Listing 5.8: Configuración de recompensa DDPG. 65
Listing 5.9: Configuración del entorno de entrenamiento DDPG. 68
Listing 5.10: Configuración despliegue agente DDPG. 74

Resumen

Este trabajo aborda la planificación y el control de trayectorias para vehículos autónomos, aplicados al robot diferencial *TurtleBot 4 Lite*. A partir de trayectorias de referencia definidas mediante *waypoints*, se entrena un agente de Aprendizaje por Refuerzo (RL) para la navegación autónoma de carreras. Posteriormente, los resultados se comparan frente a tres controladores analíticos clásicos (Pure Pursuit, MPC y LQR), evaluando exhaustivamente las ventajas e inconvenientes de cada paradigma en términos de velocidad, precisión de seguimiento y capacidad de generalización.

Palabras clave: RL, Waypoints, TurtleBot, MPC, LQR, Pure Pursuit, Planificación de Trayectorias.

Abstract

This work addresses trajectory planning and control for autonomous vehicles, applied to the differential-drive robot *TurtleBot 4 Lite*. Using refe [1]rence trajectories defined by *waypoints*, a Reinforcement Learning (RL) agent is trained for autonomous racing navigation. Afterwards, the results are compared against three classical analytical controllers (Pure Pursuit, MPC, and LQR), thoroughly evaluating the advantages and disadvantages of each paradigm in terms of speed, tracking accuracy, and generalization capability.

Keywords: RL, Waypoints, TurtleBot, MPC, LQR, Pure Pursuit, Trajectory Planning

Glosario (Lista de acrónimos)

DDPG	*Deep Deterministic Policy Gradient* (Gradiente de Política Determinista Profunda)
DQN	*Deep Q-Network* (Red Q Profunda)
LQR	*Linear Quadratic Regulator* (Regulador Cuadrático Lineal)
MDP	*Markov Decision Process* (Proceso de Decisión de Markov)
MPC	*Model Predictive Control* (Control Predictivo Basado en Modelo)
MVSIM	*Multi-Vehicle Simulator* (Simulador Multi-Vehículo)
PID	*Proportional-Integral-Derivative* (Controlador Proporcional-Integral-Derivativo)
PPO	*Proximal Policy Optimization* (Optimización de Política Próxima)
Q-Learning	*Q-Learning* (Algoritmo de aprendizaje por refuerzo libre de modelo basado en la función de valor Q)
RL	*Reinforcement Learning* (Aprendizaje por Refuerzo)
ROS2	*Robot Operating System 2* (Sistema Operativo Robótico 2)
SARSA	*State-Action-Reward-State-Action* (Algoritmo de aprendizaje por refuerzo basado en Estado-Acción-Recompensa-Estado-Acción)
TD	*Temporal Difference* (Aprendizaje por Diferencia Temporal)
TD3	*Twin Delayed Deep Deterministic Policy Gradient* (Gradiente de Política Determinista Profunda con Doble Retraso)

Capítulo 1

1. Introducción

El control autónomo de vehículos a lo largo de trayectorias predefinidas es un desafío fundamental en la robótica, cobrando una especial exigencia en el emergente ámbito de las carreras de vehículos autónomos. En estos escenarios de competición, el objetivo trasciende la navegación segura, ya que se busca empujar los límites dinámicos del vehículo para recorrer el circuito siguiendo una línea óptima (*racing line*) en el menor tiempo posible. Los métodos de control clásicos han ofrecido soluciones sólidas a este problema durante años: algoritmos como Pure Pursuit, el Control Predictivo basado en Modelo (MPC) o el Regulador Cuadrático Lineal (LQR) son capaces de seguir la trayectoria con notable precisión cuando sus parámetros están bien ajustados. No obstante, su ajuste es manual, lo que los convierte en sistemas rígidos que carecen de capacidad de adaptación autónoma frente a perturbaciones, pérdidas de tracción o incertidumbres del entorno físico, propias de la conducción al límite.

El Aprendizaje por Refuerzo (RL) ofrece una perspectiva diferente para abordar este tipo de conducción competitiva. Un agente entrenado mediante esta técnica no recibe instrucciones explícitas sobre cómo actuar, sino que descubre por sí mismo qué comportamientos son beneficiosos a través de la experiencia acumulada en interacción con el entorno. Esta propiedad lo hace especialmente interesante para tareas de control de carreras donde intervienen múltiples objetivos simultáneos (maximizar la velocidad, mantener una precisión lateral en las curvas, etc.) que son difíciles de equilibrar mediante un ajuste manual y heurístico de los parámetros de control.

Para llevar estas ideas a la práctica y emular las dinámicas de un vehículo de carreras autónomo a escala, este trabajo utiliza el robot de tracción diferencial *TurtleBot 4 Lite* como plataforma experimental. Su integración nativa con el ecosistema ROS2, combinada con el simulador MVSim y el entorno de desarrollo MATLAB, proporciona un marco de trabajo altamente integrado. Esta arquitectura no solo permite implementar y entrenar estrategias de control de forma segura en un entorno virtual, sino que garantiza una transición transparente para desplegar y evaluar el código final directamente sobre el vehículo físico en pista.

1.1 Objetivos

El objetivo principal de este proyecto es analizar en qué medida el Aprendizaje por Refuerzo (RL) representa una mejora real frente a los controladores clásicos en el ámbito de las carreras de vehículos autónomos. En concreto, se busca evaluar si un agente de RL es capaz de inferir y descubrir una trayectoria óptima de carrera por su propia cuenta mediante la exploración y la interacción con el entorno. A partir de este análisis, se pretende extraer conclusiones fundamentadas sobre el rendimiento y la robustez de estas estrategias al operar tanto en un entorno de simulación como en su despliegue directo sobre el *TurtleBot* físico.

Para materializar estos objetivos, el desarrollo del proyecto se estructura en tres grandes pilares. Inicialmente, se aborda el problema analítico de la planificación de trayectorias, transformando trazados de referencia en rutas geométricas óptimas (*racing lines*) diseñadas específicamente para maximizar la velocidad sostenida y minimizar el tiempo por vuelta. En una segunda fase, se despliegan los algoritmos de control clásico (MPC, LQR y Pure Pursuit) obligándolos a seguir estos circuitos precalculados, evaluando su precisión de seguimiento y tiempos de vuelta. Finalmente, se establece el entrenamiento de un agente de RL mediante el algoritmo DDPG. A diferencia de los métodos anteriores, el propósito de este último paso es comprobar si el agente logra aprender a navegar de forma eficiente y encontrar la trazada ideal de forma autónoma. Esto permite establecer una comparativa directa frente a los métodos tradicionales, contrastando la ruta descubierta por la inteligencia artificial con la calculada matemáticamente, para así identificar las fortalezas y limitaciones de cada paradigma.

1.2 Arquitectura de los sistemas de conducción autónoma

Durante los últimos años, el campo de la conducción autónoma ha experimentado una evolución significativa impulsada por la progresiva reducción de costes y el aumento en la capacidad de procesamiento de los sistemas informáticos y sensores. Estos dos factores han sido claves para elevar gradualmente el nivel de autonomía de los vehículos contemporáneos, una tecnología que ya está presente en el 16% de las nuevas matriculaciones en España [2]. Para conseguir que un sistema opere de forma independiente, el desarrollo de su arquitectura de software se basa sobre tres pilares fundamentales que interactúan secuencialmente: la percepción, la planificación y el control.

No obstante, para adaptarse a los requerimientos de este trabajo, esta estructura clásica se ha optimizado. Tal y como ilustra la Figura 1.1, la arquitectura propuesta divide el flujo de trabajo en una fase de planificación global ejecutada de forma previa (*offline*), y un ciclo de ejecución en tiempo real (*online*) donde la estimación del estado y el control de seguimiento operan en un lazo cerrado continuo.

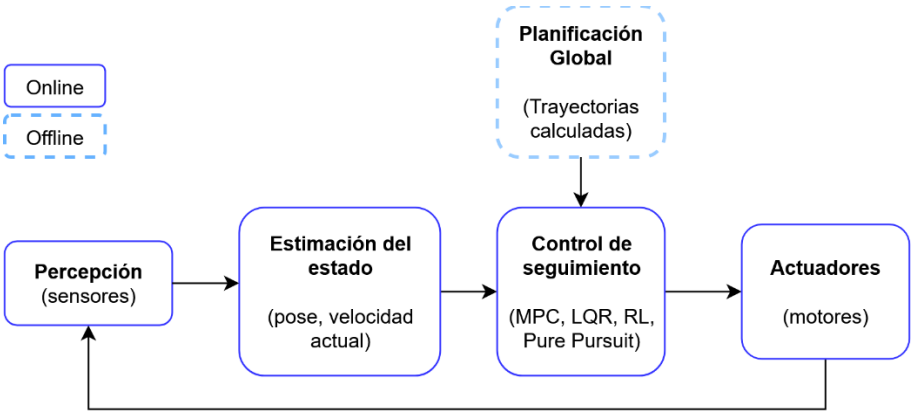


Figura 1.1: Esquema de bloques de la arquitectura general del sistema de navegación.

1.2.1 Módulo de percepción

El subsistema de percepción actúa como la interfaz sensorial del vehículo. Aunque la industria de la conducción autónoma suele emplear arquitecturas complejas con sensores LiDAR, GPS y algoritmos de *Deep Learning* para detectar obstáculos dinámicos [3], este proyecto simplifica la arquitectura al centrarse en la navegación de un único vehículo dentro de un circuito cerrado y libre de interferencias.

Como la trayectoria global se calcula de forma previa (*offline*), el vehículo no necesita mapear el entorno en tiempo real. Su percepción se apoya exclusivamente en dos fuentes: la odometría interna (basada en los *encoders* de los motores) para registrar el desplazamiento, y la cámara frontal para detectar los marcadores ArUco en la fase de planificación. Estos datos convergen en el módulo de estimación del estado, donde se calcula la pose actual (posición y orientación) y la velocidad del robot. Esta información es la que finalmente utiliza el controlador de seguimiento para corregir cualquier desviación respecto a la trayectoria a seguir.

1.2.2 Módulo de planificación

El módulo de planificación constituye el núcleo lógico dedicado a trazar la ruta que debe seguir el vehículo para desplazarse de forma eficiente. Mientras que en aplicaciones complejas de conducción urbana se emplean arquitecturas jerárquicas que incluyen planificación de misión y de comportamiento para interactuar con el tráfico [3], este trabajo omite dichas capas al no existir obstáculos dinámicos que obliguen al vehículo a realizar maniobras de evasión o a recalcular la ruta en tiempo real.

De este modo, el sistema se centra exclusivamente en la planificación global (o planificación de movimiento). Este proceso se ejecuta de manera previa (*offline*) en MATLAB, donde se transforman los *waypoints* iniciales del circuito en una trayectoria de referencia central, que es suavizada y luego optimizada. Una vez calculados los puntos de las trayectorias definitivas, el archivo generado se introduce directamente en el sistema del robot, sirviendo como la referencia estática para el controlador de seguimiento en tiempo real.

1.2.3 Módulo de control

La etapa de control constituye el último eslabón de la arquitectura. Dado que la generación de la ruta se ha delegado por completo a la fase *offline*, este módulo actúa exclusivamente como un controlador de seguimiento (*tracking controller*). Su objetivo es transformar la trayectoria precalculada en comandos de velocidad para los motores, intentando anular en cada instante tanto el error lateral como el error de orientación respecto a la trazada. Para garantizar esta precisión, el módulo opera en lazo cerrado (*feedback control*), cerrando el bucle del sistema: la acción de los actuadores modifica la posición física del robot, los sensores capturan este cambio y el estimador

de estado actualiza la pose. Al evaluar continuamente esta ubicación real frente a la línea de referencia, el sistema es capaz de compensar perturbaciones físicas y corregir sus movimientos en tiempo real.

Para abordar este desafío de seguimiento, en este trabajo se comparan dos enfoques distintos: los algoritmos de control clásicos como *Pure Pursuit*, LQR y MPC, y un enfoque de Aprendizaje por Refuerzo mediante DDPG. De este modo, se establece el núcleo comparativo del trabajo para evaluar la capacidad de adaptación de la inteligencia artificial frente a los métodos tradicionales de control.

Capítulo 2

2. Marco teórico

2.1 Algoritmo Pure Pursuit

El algoritmo *Pure Pursuit* constituye uno de los métodos de control geométrico más extendidos para el seguimiento de trayectorias en robots móviles, destacando por su robustez y simplicidad computacional. Su propósito fundamental es calcular la curvatura y velocidad angular necesaria para que un vehículo se desplace desde su posición actual hasta un punto definido en la trayectoria de referencia (*waypoints*), asumiendo una velocidad lineal constante [4].

Para ello, el controlador *Pure Pursuit* opera basándose en un enfoque geométrico. En primer lugar, define un punto objetivo sobre la trayectoria deseada, el cual se encuentra siempre a una distancia predefinida desde la posición actual del robot. Esta magnitud métrica se denomina *Lookahead Distance*. Se puede pensar como una circunferencia centrada en el robot cuyo radio es exactamente esta distancia *lookahead*, y el punto objetivo se establece en la intersección de dicha circunferencia con la trayectoria a seguir.

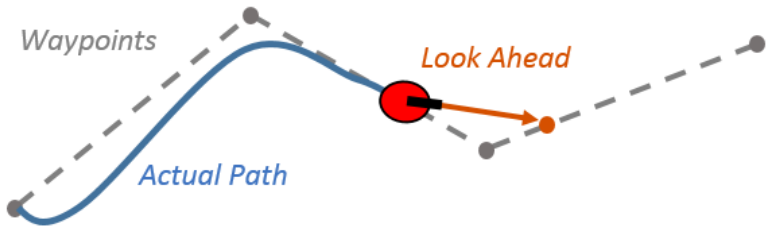


Figura 2.1: Algoritmo Pure Pursuit. [4]

De esta forma el algoritmo transforma las coordenadas de dicho punto a un sistema de referencia local al robot. En este sistema de coordenadas, se determina el ángulo θ (*theta*) que representa la desviación angular entre el vector de orientación actual del vehículo y la línea de visión directa hacia el punto objetivo. A partir de este ángulo, se calcula la curvatura del arco circular a seguir (k), la cual es proporcional al seno de θ y está condicionada por la distancia *lookahead* (L_d):

$$k = \frac{2 \sin \theta}{L_d} \tag{2.1}$$

Finalmente, multiplicando esta curvatura geométrica por la velocidad lineal impuesta al sistema (v), el controlador determina la velocidad angular (ω) exacta que los actuadores deben aplicar para que el robot trace dicho arco hacia el punto objetivo:

$$\omega = k \times v \tag{2.2}$$

Sin embargo, el rendimiento y la estabilidad de este controlador dependen críticamente del ajuste del parámetro *lookahead* ya mencionado. Si se configura una distancia corta, el robot intentará corregir rápidamente cualquier error lateral, lo que proporciona un seguimiento muy ajustado del camino, pero puede generar muchas oscilaciones. Por el contrario, una distancia *lookahead* larga suaviza significativamente la respuesta del robot, reduciendo las oscilaciones y generando movimientos más fluidos. No obstante, este enfoque "predictivo" provoca que el robot comience a girar mucho antes de llegar a las curvas, lo que introduce un error de seguimiento ya que no sigue la trayectoria a rajatabla. Por este motivo es muy importante encontrar un equilibrio perfecto entre un *lookahead* pequeño para que siga bien la trayectoria, y un *lookahead* grande para que no oscile y recorte las curvas lo mínimo necesario.

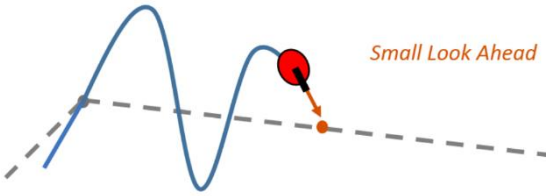


Figura 2.2: Lookahead Pequeño. [4]

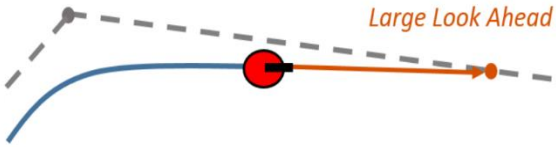


Figura 2.3: Lookahead Grande. [4]

2.2 MPC

El Control Predictivo Basado en Modelo, conocido como MPC por sus siglas en inglés (*Model Predictive Control*), es una técnica de control avanzado que surgió a finales de la década de los setenta. A diferencia del control clásico, MPC comprende una amplia variedad de métodos unidos por una misma base teórica. Esta consiste en integrar un modelo de la planta para predecir sus estados futuros y, de este modo, calcular las acciones óptimas que minimicen el error de seguimiento evaluando una función de coste [3].

Para entender su funcionamiento de manera más intuitiva, podemos compararlo con un caso diario de nuestras vidas. Cuando conducimos, conocemos nuestro destino y siempre (o casi siempre) observamos constantemente los metros de carretera que tenemos por delante. En nuestra cabeza, para seguir este camino, simulamos las distintas acciones que puede realizar el vehículo. Es decir, si tenemos en cuenta las limitaciones del coche (por ejemplo, su inercia o capacidad de frenado), debemos decidir cuánto pisar el acelerador o cuánto girar el volante para no salirnos del carril. Dado que el entorno se encuentra en cambio constante, debemos recalcular mentalmente estas acciones de forma continua [5].

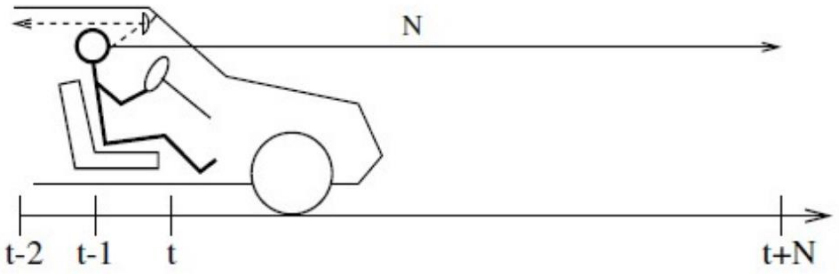


Figura 2.4: Analogía de la conducción humana para explicar el principio del control predictivo.

Por el contrario, los controladores clásicos reactivos (como PID) no miran hacia el futuro, sino que basan sus correcciones en el error instantáneo presente y pasado. Si utilizamos la misma analogía, manejar un coche con un controlador PID sería el equivalente a conducir mirando únicamente por el espejo retrovisor [5]. Por este motivo, el MPC es una herramienta muy potente para nuestro proyecto, ya que permite al robot anticiparse a las curvas y lograr una navegación mucho más suave y rápida a la hora de seguir una ruta de *waypoints*.

La capacidad de adaptación y el éxito del MPC en la industria y la conducción autónoma se deben gracias a las siguientes propiedades [3]:

- **Basado en modelo.** El controlador necesita un modelo matemático que sea lo más fiel posible a la realidad del sistema físico. Cuanto más se aproxime este modelo a la cinemática real del robot, mejores y más exactas serán las predicciones.
- **Manejo de restricciones.** A la hora de calcular la ruta, el algoritmo asegura por diseño que se respeten estrictamente las limitaciones de seguridad del *hardware*. En nuestro caso, esto se traduce en no pedirle nunca al robot unos comandos que superen su velocidad lineal y angular máximas.
- **Sistemas multivariables:** El MPC es capaz de manejar sistemas MIMO (múltiples entradas y múltiples salidas) gestionando de manera conjunta señales que interactúan entre sí sin llegar a desestabilizar el sistema. En nuestro caso, esto es clave porque las variables del robot diferencial están acopladas. Es decir, los comandos de entrada (velocidad lineal y angular) modifican simultáneamente los estados de salida (posición y orientación).
- **Retroalimentación (*Feedback*).** La evaluación de la situación actual del vehículo por parte del controlador se realiza constantemente con la lectura de los sensores (*feedback*). Con esta información real se corrigen los cálculos, lo cual es vital para evitar errores de trayectoria a largo plazo causados por perturbaciones del mundo real.

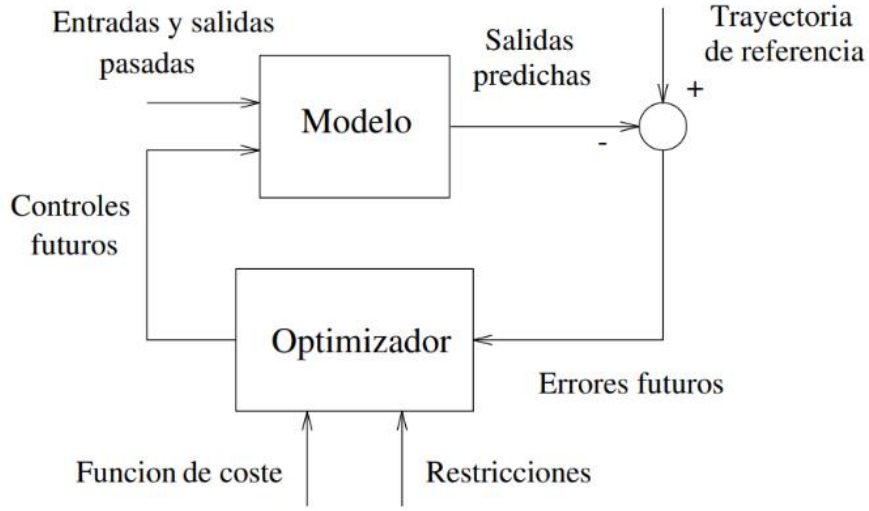


Figura 2.5: Diagrama controlador MPC. [6]

Como se puede observar en la arquitectura general de la Figura 2.5, todo controlador predictivo sigue un bucle cuya estrategia está compuesta por tres fases principales [3]:

1. **Predicción**. En cada instante actual de tiempo k, MPC utiliza el bloque del modelo matemático para predecir cuál será la evolución futura del sistema. Esta estimación se realiza evaluando una ventana de tiempo hacia adelante $(k+1)$, conocida como horizonte de predicción (Hp). A partir del estado actual y de las lecturas pasadas, el algoritmo genera una secuencia de salidas predichas:

$$y(k+i \mid k), \qquad para\ i = 1, \dots, Hp \tag{2.3}$$

   Estas salidas representan exactamente dónde acabará el vehículo si se le aplica una secuencia hipotética de futuras señales de control $u(k)$ a lo largo de un horizonte de control (Hc):

$$u(k+i \mid k), \qquad para\ i = 0, \dots, Hc-1 \tag{2.4}$$

2. **Optimización**. Por otro lado, se encuentra el bloque optimizador, cuya tarea consiste en encontrar la secuencia de señales de control (2.4) para que la ruta predicha se ajuste lo máximo posible a la trayectoria de referencia. Para lograrlo, el algoritmo minimiza una función de coste, que corresponde a la ecuación (2.5).

$$J = \sum_{i=1}^{Hp} e^T Q_e e + \sum_{i=0}^{Hc-1} \Delta u^T Q_u \Delta u \tag{2.5}$$

Por un lado, el primer término penaliza el error de seguimiento e, es decir, la desviación entre la trayectoria ideal y la predicción futura simulada por el modelo. Por otro lado, el segundo término castiga los cambios bruscos en los comandos de velocidad Δu, de tal forma que, si el robot frena de golpe o da un volantazo, el valor de Δu será enorme. Por tanto, el objetivo finalmente será encontrar la secuencia de control que mejor siga al circuito minimizando J.

3. **Horizonte Deslizante**. Una vez que el optimizador resuelve la ecuación (2.5) y encuentra la secuencia de control ideal para los próximos instantes, el controlador envía al robot solo la primera señal calculada, es decir $u(k|k)$, y el resto de la secuencia es descartada. Esto se debe a que, tras ejecutar esa primera acción, el sistema de sensores proporciona una lectura real del nuevo estado $y(k+1)$. Al disponer de esta información del entorno actualizada, las señales calculadas previamente quedan incongruentes y obsoletas, lo que obliga al algoritmo a recalcular todo de cero para el siguiente instante.
   Este concepto es conocido como "horizonte deslizante" (Figura 2.6), ya que la ventana temporal de predicción se va desplazando hacia el futuro mientras el robot avanza. Este ciclo “infinito” de planificar a largo plazo, pero ejecutar solo a corto plazo es lo que le otorga al sistema la capacidad de rechazar perturbaciones y seguir la trayectoria de forma óptima.

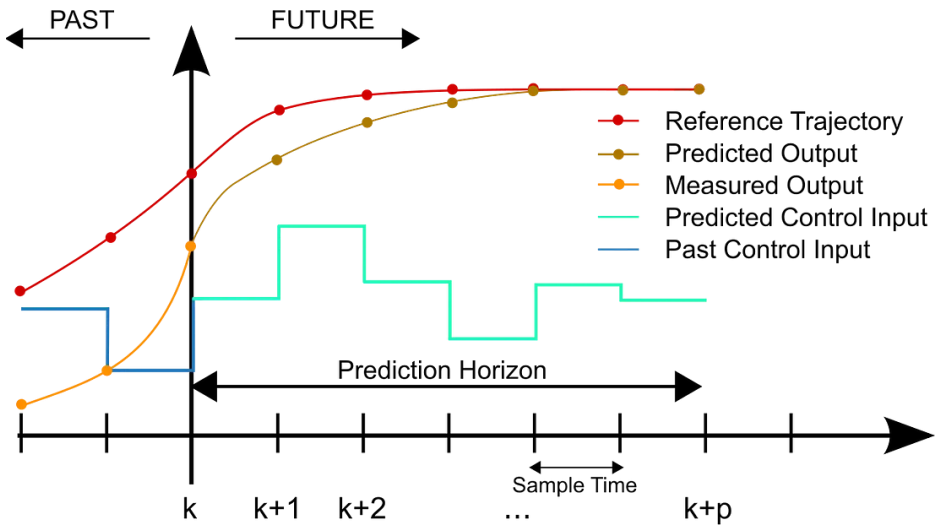


Figura 2.6: Principio de funcionamiento del horizonte deslizante en un controlador MPC.

2.3 LQR

El Regulador Cuadrático Lineal, a partir de ahora LQR por sus siglas en inglés, es un método de control muy utilizado en la industria para controlar sistemas dinámicos de manera precisa. Su objetivo principal es encontrar las entradas óptimas de control $u(t)$, como puede ser la velocidad angular de las ruedas de un vehículo, que minimicen una función de coste cuadrática, compuesta por el error del estado y por el esfuerzo de control. Por tanto, el valor global de la función de coste será menor cuanto más pequeño sea el error del estado, y cuanto menor sea el esfuerzo exigido a los actuadores.

A diferencia de MPC, que simula estados futuros con el horizonte de predicción y resuelve optimizaciones complejas en tiempo real durante el desplazamiento del vehículo, LQR plantea un enfoque mucho más ligero en cuanto a coste computacional. Este calcula una regla de control fija de forma previa (fase *offline*), lo que le ahorra al controlador de hacer cálculos complejos mientras está en funcionamiento (fase *online*), teniendo que realizar únicamente una multiplicación matricial básica (2.8). Esto resulta ideal para sistemas que necesitan cálculos muy rápidos.

Para poder aplicar este controlador, el sistema sobre el que trabaja debe ser estrictamente lineal. Dado que la cinemática de nuestro robot contiene funciones trigonométricas que describen su orientación y giro, como seno y coseno, el sistema es no lineal. Para linealizar y discretizar este modelo se pueden usar matrices Jacobianas, obteniendo la siguiente ecuación de espacio de estados [7]:

$$x(t+1) = Ax(t) + Bu(t) \tag{2.6}$$

Donde:

- $x(t+1)$ representa el vector futuro del estado del robot, es decir, su posición $[x, y]$ y orientación $[\theta]$ futuras en la pista.
- $x(t)$ representa el vector actual del estado del robot $[x, y, \theta]$.
- $u(t)$ representa el vector actual de entradas de control, es decir, los comandos de velocidad lineal $[v]$ y velocidad angular $[\omega]$.
- La matriz A (matriz de estados) describe la dinámica interna del error, indicando cómo cambia el estado del sistema $[x, y, \theta]$ desde t a $t+1$ debido a la propia configuración y orientación del robot.
- La matriz B (matriz de entrada) describe cómo cambia el estado del sistema $[x, y, \theta]$ desde t a $t+1$ debido a los comandos de control de entrada $[v, \omega]$ [8].

El objetivo por tanto será encontrar el valor $u(t)$ óptimo (2.8), de tal forma que:

- El error del estado ($x(t)_{actual} - x(t)_{deseado}$) sea mínimo,
- $u(t)$ a su vez sea lo más pequeño posible para que el esfuerzo de los actuadores también sea mínimo, y
- Se encuentre un equilibrio entre ambos [7].

Una vez definido el modelo lineal del sistema, el algoritmo busca la mejor secuencia de control minimizando la función de coste cuadrática ya mencionada anteriormente, denominada J. Esta función evalúa el rendimiento del vehículo y se expresa de la siguiente forma:

$$J = \sum_{t=0}^{\infty} (x(t)^T Q x(t) + u(t)^T R u(t)) \tag{2.7}$$

El usuario o diseñador del sistema será el encargado de seleccionar cuidadosamente los valores de las matrices Q y R de la función de coste J. Por un lado, la matriz Q pondera el valor del vector de estados $[x(t)]$ y penaliza las desviaciones actuales del sistema con respecto a su estado deseado $[x(t)_{actual} - x(t)_{deseado}]$. Si se le asigna un valor alto, intentará corregir el error lo más rápido posible sin importar el esfuerzo que conlleve [7]. Por otro lado, la matriz R pondera el vector de entradas de control $[u(t)]$ y penaliza el esfuerzo de los actuadores. Si se le asigna un valor alto el sistema tardará más tiempo en corregir el error, pero priorizará la eficiencia energética y ejecutará acciones de control más suaves, ya que el esfuerzo del actuador es menor [7].

Por último, luego de definir las prioridades mediante las matrices Q y R, la minimización matemática de la función de coste J se resuelve mediante la Ecuación Algebraica de Riccati [7] [9], lo que nos da como resultado la matriz de ganancias K. Esta matriz es la que dicta la acción de control $u(t)$ que debe tomarse dado el estado actual $x(t)$:

$$u(t) = -K \cdot x(t) \tag{2.8}$$

La principal ventaja de esta formulación y del LQR es que todo el trabajo pesado de optimización fue realizado previo a que el sistema comience a funcionar, es decir, en la fase *offline*. Una vez que se obtiene la matriz de ganancias K y posteriormente el robot empiece a circular, este únicamente tendrá que leer los sensores para saber su estado o trayectoria actual $x(t)$, y multiplicar dichos valores por la matriz K ya calculada. Esto nos da como resultado el vector $u(t)$ ya optimizado que, como mencionamos anteriormente, será la nueva entrada de control que se enviará a los actuadores para controlar la velocidad lineal y angular y corregir el error de trayectoria.

2.4 Aprendizaje por refuerzo

La historia del Aprendizaje por Refuerzo, a partir de ahora RL por sus siglas en inglés, tiene raíces profundas que se remontan a la psicología del comportamiento animal, concretamente al aprendizaje por ensayo y error. Durante la década de 1940, investigadores en psicología y pioneros de la inteligencia artificial comenzaron a estudiar cómo, en la naturaleza, los organismos biológicos descubren acciones beneficiosas y mejoran su comportamiento mediante la experimentación continua y un sistema natural de recompensas y castigos. De manera análoga, un algoritmo de RL construye su conocimiento puramente a través del mecanismo de ensayo y error, descartando las estrategias fallidas y afianzando aquellas que le otorgan un beneficio.

A diferencia del aprendizaje supervisado, el cual requiere una fase de entrenamiento con grandes volúmenes de datos previamente etiquetados, RL se fundamenta en un ciclo continuo de retroalimentación interactiva. En este paradigma, la unidad de procesamiento, conocida como el agente, evalúa las condiciones de su entorno en un instante dado y ejecuta una acción sobre él. Como consecuencia de esta intervención, el entorno cambia de estado y emite una señal escalar conocida como recompensa, la cual cuantifica el beneficio inmediato de la acción elegida por el agente. Es crucial destacar que el agente carece de instrucciones explícitas sobre cómo debe resolver la tarea, por lo que su objetivo consiste en descubrir, por sus propios medios, una política de comportamiento que maximice el sumatorio de las recompensas acumuladas a lo largo del tiempo.

Para entender el funcionamiento de forma más intuitiva, resulta útil recurrir a la analogía del adiestramiento canino [10]. En términos de RL, el perro actúa como **agente** y todo lo que le rodea, incluida la adiestradora, constituye el **entorno**. Ante un comando (observación), el perro ejecuta una **acción**. Si esta se aproxima al comportamiento deseado, como por ejemplo sentarse, recibe comida como **recompensa**; en caso contrario, no recibe nada. Al principio del entrenamiento el perro actúa de forma casi aleatoria, pero con el tiempo aprende a asociar cada observación con la acción que maximiza su recompensa (la comida). Esta correspondencia entre observaciones y acciones es lo que en RL se denomina **política**. El objetivo del entrenamiento es precisamente ajustar esa política para que el agente aprenda los comportamientos deseados y obtenga la mayor recompensa acumulada posible.

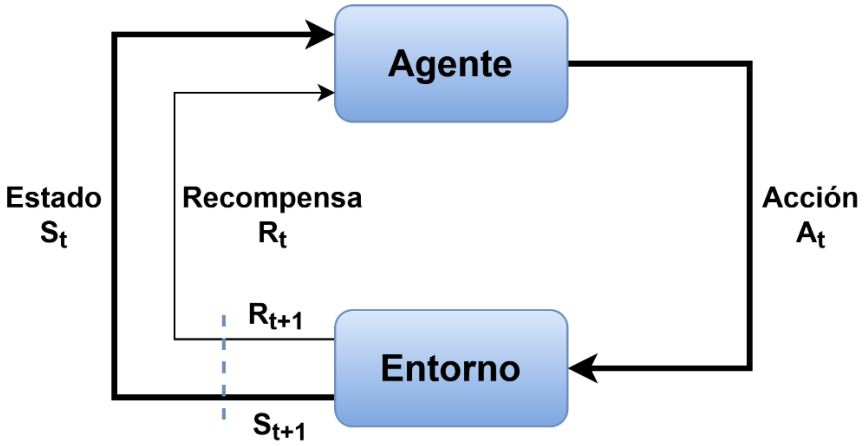


Figura 2.7: Ciclo del RL. Elaboración Propia.

2.4.1 Proceso de Decisión de Markov

Para poder aplicar los algoritmos de RL de manera precisa, es necesario formalizar el problema que el agente tiene que resolver. El marco matemático estándar para ello es el Proceso de Decisión de Markov (MDP), que proporciona una descripción completa de cómo el agente interactúa con el entorno. Un MDP queda definido por cuatro elementos fundamentales:

- **Estados S.** Conjunto de todas las situaciones posibles en las que puede encontrarse el agente. En un robot móvil, el estado podría incluir su posición, orientación y velocidad.
- **Acciones A.** Conjunto de todas las decisiones que el agente puede tomar en cada estado. En el caso de un robot diferencial, esto se traduce en las velocidades lineal y angular que se le ordenan.
- **Función de transición $\boldsymbol{P(s'|s,a)}$.** Describe la probabilidad de pasar al estado s' si, estando en el estado s, el agente ejecuta la acción a. Captura la dinámica del entorno, es decir, cómo evoluciona debido a las acciones del agente.
- **Función de recompensa $\boldsymbol{R(s,a,s')}$.** Asigna un valor numérico a cada transición, indicando cuánto beneficio inmediato obtiene el agente al pasar del estado s al estado s' mediante la acción a.

A partir de estos elementos, el agente aprende una política $\pi(a|s)$, que actúa como una regla que le indica qué acción a debe tomar cuando se encuentra en el estado actual s. El objetivo es encontrar la política óptima $\boldsymbol{\pi *}$ que maximice una función de valor, conocida como retorno o valor esperado $V_\pi(s)$, que nos indica que tan valioso es estar en el estado s. Esta función puede ser definida como la suma acumulada de recompensas futuras esperadas, normalmente ponderadas por un factor de descuento $\gamma \in [0,1]$ que reduce el peso de las recompensas lejanas en el tiempo:

$$V_\pi(s) = r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + \cdots = \sum_{k=0}^{\infty} \gamma^k r_{t+k+1} \tag{2.9}$$

Como se observa en la ecuación, si el factor es cercano a 0, el robot tendrá un comportamiento reactivo buscando el beneficio inmediato. Por el contrario, al establecer un valor cercano a 1, las recompensas futuras mantienen un peso significativo en la sumatoria, otorgando al controlador de visión a largo plazo para planificar acciones que le lleven al éxito.

2.4.2 Familias RL

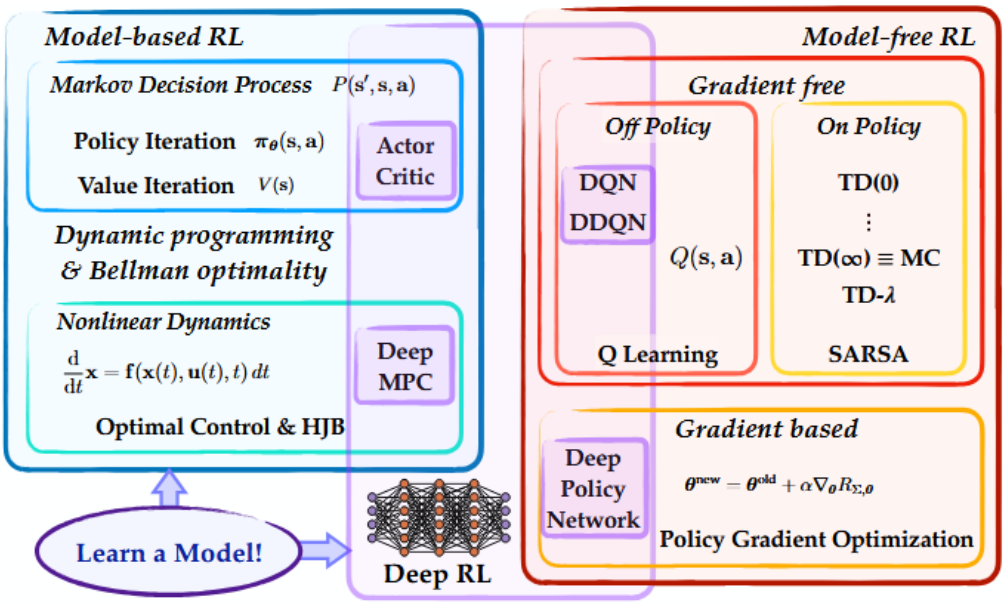


Figura 2.8: Categorización de las técnicas de aprendizaje por refuerzo. Fuente: Brunton y Kutz (2022) [11].

Una vez entendido MDP, la cuestión central para el diseño del controlador radica en si el agente posee acceso a la función de transición $P(s'|s, a)$ u otro modelo del entorno. La respuesta a dicha cuestión divide los algoritmos de RL en dos grandes familias [11], tal como se ve en la Figura 2.8:

- **RL basado en modelo (*Model-Based*).** El agente dispone de un modelo del entorno, es decir, conoce cómo sus acciones afectan al estado del sistema. Este modelo puede tomar dos formas distintas según la naturaleza del problema. La primera es el MDP discreto, ya explicado en el apartado anterior, donde la dinámica del entorno queda recogida en la función de transición $P(s'|s, a)$: matriz o tabla que indica la probabilidad de pasar de un estado a otro al ejecutar una acción determinada. La segunda forma aparece cuando el sistema es continuo y viene descrito por ecuaciones diferenciales no lineales de la forma:

$$\dot{x} = f(x, u, t) \tag{2.10}$$

donde x es el vector de estado, u es la acción de control y t el tiempo. En este caso no existe ninguna tabla de transiciones, ya que la dinámica del sistema está capturada por la función f, que puede representar, por ejemplo, las ecuaciones del movimiento de un robot. Este es el terreno del control óptimo y de la ecuación de Hamilton-Jacobi-Bellman, que permite derivar políticas de control analíticas o métodos predictivos como el MPC [11].

En ambos casos, disponer de un modelo permite al agente planificar con antelación. Es decir, antes de actuar puede simular varias secuencias de acciones, evaluar sus consecuencias y elegir la mejor opción. La ventaja principal de esta rama es la eficiencia en el uso de datos, ya que el agente extrae mucha información de pocas interacciones reales. El inconveniente es que construir un modelo fiable es difícil, ya que, si tiene errores, el agente puede aprender comportamientos subóptimos o incorrectos.

- **RL sin modelo (*Model-Free*).** El agente no dispone de la función de transición del entorno y tampoco intenta aprenderla. En lugar de planificar sobre un modelo, descubre la política óptima directamente a través de la experiencia. Este interactúa con el entorno, observa las consecuencias de sus acciones y ajusta su comportamiento mediante prueba y error. La ventaja principal es que no necesita ningún modelo intermedio, lo que simplifica el diseño del sistema. Como contrapartida, requiere muchas más interacciones con el entorno para converger, ya que toda la información debe extraerse de datos empíricos.

Dentro del RL sin modelo, se establece una primera distinción según cómo se optimiza la política [11]:

- **Métodos basados en gradiente (*gradient-based*).** La política no es una tabla ni una regla fija, sino una red neuronal cuyos parámetros θ definen completamente el comportamiento del agente. En lugar de aprender primero una función de valor y extraer la política de forma indirecta, el agente optimiza directamente esos parámetros para maximizar la recompensa acumulada esperada. Para ello calcula el gradiente de la recompensa respecto a θ, es decir, en qué dirección hay que mover los pesos para mejorar, y da un paso en esa dirección. Esto es lo que se conoce como ascenso por gradiente de política (*policy gradient*), y queda recogido en la siguiente expresión:

$$\theta_{nuevo} = \theta_{viejo} + \alpha \nabla_\theta R_{\sum,\theta} \tag{2.11}$$

Donde:

  - θ son los pesos de la red neuronal que se actualizan en cada iteración,
  - $\nabla_\theta R_{\sum,\theta}$ es el gradiente de la recompensa esperada respecto a esos pesos, es decir, la dirección en la que hay que moverlos para mejorar la política, y
  - α es la tasa de aprendizaje (*learning rate*), un hiperparámetro fijo que controla el tamaño de cada paso.

La salida de esta red neuronal define la política del agente. Dependiendo del algoritmo, la red puede producir directamente una acción (política determinista) o una distribución de probabilidad sobre las acciones posibles (política estocástica). Los algoritmos REINFORCE y PPO son ejemplos de métodos basados en políticas estocásticas, mientras que DDPG y TD3 emplean políticas deterministas.

- **Métodos libres de gradiente (*gradient-free*).** Aquí el agente no calcula ningún gradiente de la política. En su lugar, aprende a estimar la función de valor $Q(s, a)$, que indica cuánta recompensa acumulada puede esperar el agente si, estando en el estado s, ejecuta la acción a y luego sigue una política determinada. A partir de esta función de valor se puede extraer la política óptima sin necesidad de derivar nada explícitamente, ya que simplemente se elige la acción que maximiza Q en cada estado. Q-learning y DQN son los representantes más conocidos de este grupo.

Por último, dentro de la familia RL sin modelo y dentro de estos métodos libres de gradiente, aparece una segunda distinción fundamental, basada en cómo se gestiona la experiencia recopilada:

- **Basados en política (*on-policy*).** El agente aprende exclusivamente de las experiencias generadas por su propia política actual. Cada lote de datos se usa para actualizar los pesos de la red y luego se descarta, porque en cuanto la política cambia esos datos dejan de ser representativos. Este mecanismo garantiza un aprendizaje más estable y monótono, pero penaliza la eficiencia de datos. Los algoritmos más representativos de esta rama son SARSA, TD(0), TD(λ) y Monte Carlo [11].
- **Basados sin política (*off-policy*).** El agente puede aprender de experiencias recogidas por una política distinta a la que está intentando mejorar. Esto se logra implementando una memoria de repetición (*replay buffer*), donde se almacenan todas las transiciones (experiencias) pasadas (s, a, r, s'). Durante el entrenamiento, el algoritmo extrae minilotes aleatorios de este *buffer*, lo que permite reutilizar los datos múltiples veces y acelera el aprendizaje. La desventaja es que dicho entrenamiento puede ser menos estable si el *buffer* contiene experiencias de políticas muy alejadas de la actual. Esta arquitectura es la base de algoritmos punteros en control continuo como DDPG y DQN [11].

2.4.3 Funciones de recompensas

Tal como se ha explicado en la formulación del MDP, la función de recompensa $R(s, a, s')$ constituye la señal fundamental que guía al agente durante su entrenamiento, indicándole cuantitativamente el beneficio obtenido tras cada transición. Su diseño es uno de los aspectos más críticos en la ingeniería de cualquier sistema de RL, ya que una función mal condicionada puede llevar al controlador a desarrollar comportamientos erráticos o indeseables.

Una de las primeras decisiones arquitectónicas es definir la densidad temporal de esta señal:

- **Recompensa densa.** El agente recibe una retroalimentación escalar en cada instante de tiempo o paso de simulación. Esta continuidad facilita la convergencia, ya que el modelo dispone de información constante sobre su rendimiento temporal. No obstante, si la señal intermedia no está perfectamente calibrada, puede arrastrar al robot hacia óptimos locales que no resuelven el problema real [12].
- **Recompensa dispersa.** Solo se otorga una penalización o bonificación cuando el agente alcanza un estado terminal (como "llegar a la meta" o "colisionar"). Aunque simplifica la especificación del objetivo general, ralentiza bastante el entrenamiento, ya que el agente debe explorar a ciegas durante múltiples iteraciones hasta encontrar una señal útil [12].

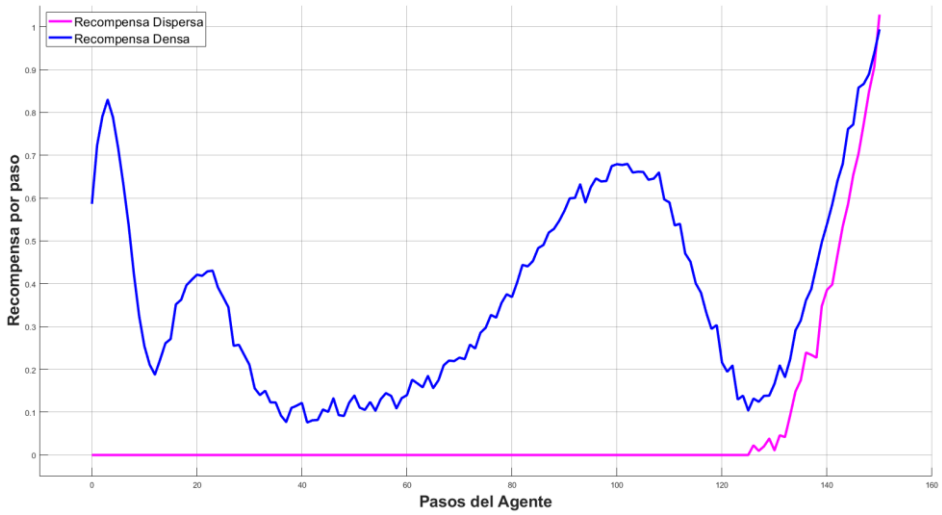


Figura 2.9: Comparativa temporal entre una señal de recompensa densa y una dispersa. Elaboración propia.

Como se muestra en la Figura 2.9, la elección de la densidad temporal de la señal determina por completo la curva de aprendizaje del agente. Para abordar este desafío estructural, el arte de diseñar cómo y cuándo se entregan los incentivos en RL se divide habitualmente en dos áreas conceptuales diferenciadas: el *Reward Engineering* y el *Reward Shaping* [13].

Por un lado, el *Reward Engineering* (ingeniería de recompensa) abarca la etapa de creación inicial de la función de recompensa base. Tal como se ha explicado anteriormente, esta

función $R(s, a, s')$ es la encargada de mapear los estados, las acciones y los estados sucesores hacia un valor numérico escalar. El reto principal durante esta fase de diseño radica en encontrar el equilibrio adecuado: la función debe proporcionar retroalimentación (*feedback*) lo suficientemente informativa como para facilitar el aprendizaje, pero debe ser lo bastante concisa para evitar que el agente descubra soluciones triviales que no resuelvan el problema real.

Un ejemplo clásico de ingeniería de recompensa, planteado para un problema de navegación básica en una cuadrícula (*grid-world*), se formula asignando puntuaciones directas a los estados resultantes [13]:

$$R(s, a, s') = \begin{cases} +10 & si\ s' es\ el\ estado\ objetivo \\ -1 & si\ s' no\ es\ el\ estado\ objetivo \end{cases} \tag{2.12}$$

Por otro lado, una vez que la función base ha sido establecida, entra en juego el *Reward Shaping*, cuyo objetivo es afinar y enriquecer estas señales iniciales. Para ello, modifica la función original inyectando incentivos intermedios para acelerar el proceso de aprendizaje, pero bajo la estricta condición de no alterar la política óptima teórica del sistema [13].

El enfoque más extendido para lograr este comportamiento es el *Reward Shaping* basado en funciones de potencial, donde se introduce una función auxiliar para modificar la estructura de las recompensas. La ecuación que rige esta transformación se define como:

$$R'(s, a, s') = R(s, a, s') + \gamma \Phi(s') - \Phi(s) \tag{2.13}$$

Donde:

- $R'(s, a, s')$ representa la nueva recompensa total percibida por el agente,
- γ actúa como el factor de descuento del proceso de Markov, y
- La función de potencial $\Phi(s)$ actúa como un indicador que cuantifica qué tan bien posicionado está el sistema respecto a su objetivo.

La clave de esta formulación reside en calcular la diferencia entre el potencial del estado al que acaba de llegar (s') y el estado anterior del que partió (s). Al restar ambos valores, el algoritmo evalúa instantáneamente si el agente ha mejorado su situación tras realizar un movimiento. Si la acción ha sido acertada y el sistema se aproxima a la meta, esta diferencia de potenciales se traduce en una bonificación que se suma a la recompensa principal. Estos pequeños incentivos son los que ayudan a guiar el controlador paso a paso y aceleran radicalmente el aprendizaje. A modo de ejemplo práctico, en la navegación de un vehículo, este mecanismo de potenciales se traduce en otorgar una retroalimentación positiva al alcanzar "submetas" o al evidenciar un progreso constante hacia el destino, recompensando la buena trayectoria independientemente de si ha llegado a la meta final.

2.4.4 DDPG

El algoritmo Deep Deterministic Policy Gradient, a partir de ahora DDPG, es un método de RL que pertenece a la familia *model-free* de los métodos actor-crítico basados en gradiente y *off-policy*. Para entender por qué y de donde surge DDPG, es útil partir de las limitaciones de los dos enfoques que lo preceden.

Por un lado, los métodos de gradiente basados en políticas estocásticas, como REINFORCE o PPO, representan la política como una distribución de probabilidad $\pi(a|s)$ sobre las acciones posibles. Esto significa que, ante un mismo estado, el agente no siempre ejecuta la misma acción, sino que muestrea una acción de dicha distribución. Aunque esto favorece la exploración del entorno durante el aprendizaje, también introduce un componente aleatorio en la toma de decisiones del agente, lo que puede resultar poco adecuado en aplicaciones que requieren comportamientos más predecibles.

Por otro lado, los métodos basados en valor o libres de gradiente, como DQN, aprenden una función $Q(s,a)$ que estima la recompensa acumulada esperada para cada par estado-acción. La política se obtiene seleccionando la acción con mayor valor Q, lo que produce comportamientos deterministas y estables. Sin embargo, al hacer esto se requiere evaluar las acciones disponibles para encontrar la mejor, que resulta adecuado para espacios de acción discretos, pero poco práctico cuando el espacio de acciones es continuo, como ocurre con velocidades o ángulos de control [14].

DDPG surge precisamente para superar estas limitaciones y cubrir el hueco existente, ya que combina lo mejor de ambos mundos. Del aprendizaje basado en valor hereda el uso de una función $Q(s,a)$ para evaluar acciones, mientras que de los métodos de gradiente adopta la representación de la política mediante una red neuronal optimizada directamente por gradiente. Gracias a esta combinación, DDPG puede aprender políticas deterministas en espacios de acción continuos, evitando tanto la necesidad de discretizar las acciones como la aleatoriedad inherente de las políticas estocásticas [14].

2.4.4.1 Arquitectura: actor y crítico

Una vez entendido el origen de DDPG, debemos explicar las dos redes neuronales que lo componen, tal como se ve en las figuras (2.10) y (2.11). Por un lado, se tiene la red "actor" que recibe el estado actual como entrada y devuelve directamente una acción determinista $\mu_\theta(s)$, es decir, siempre produce la misma acción para un mismo estado. Por otro lado, se tiene la red "crítico" que recibe tanto el estado como la acción como entrada y devuelve el valor $Q_\phi(s,a)$, es decir, la recompensa acumulada esperada si se ejecuta esa acción en ese estado. En términos sencillos, el crítico actúa como guía del actor, ya que le indica cuánto vale cada acción que saca a su salida, y según esto, pueda mejorar su política.

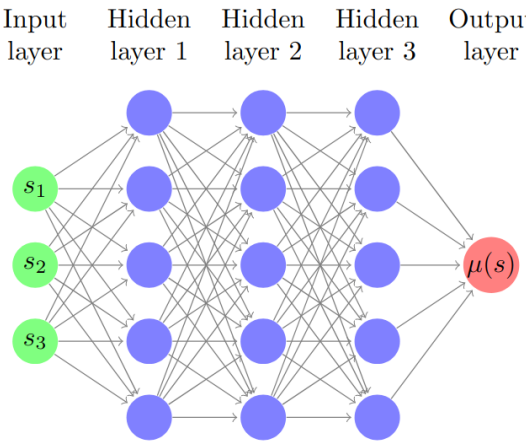


Figura 2.10: Ejemplo de una red actor μθ(s) en DDPG. Fuente: [14]

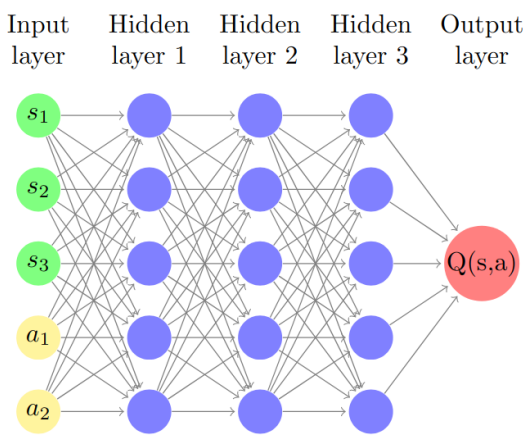


Figura 2.11: Ejemplo de una red crítico Q(s,a) en DDPG. Fuente: [14]

Para entender mejor el funcionamiento que hay por detrás de ambas redes, a continuación vamos a proceder a profundizar un poco más.

2.4.4.2 Actualización del crítico

El crítico necesita aprender a estimar $Q(s,a)$, es decir, cuánta recompensa acumulada va a obtener si está en el estado s y ejecuta la acción a. Sin embargo, para aprender esto necesita una referencia, un valor optimo al que debería converger, para luego poder compararse. Este valor se basa en las ecuaciones de Bellman y es conocido como "*TD target*" o y:

$$y = r + \gamma \cdot Q_{\phi targ}(s', \mu_{\phi targ}(s')) \tag{2.14}$$

Donde:

- r es la recompensa inmediata observada en el entorno, siendo un dato real.
- $\gamma \cdot Q_{\phi targ}(s', \mu_{\phi targ}(s'))$ es la estimación de cuánto vale el siguiente estado futuro, usando las redes target para que sea estable

Una vez se tiene y, el critico principal se entrena minimizando el error cuadrático entre su estimación actual $Q_\phi(s,a)$ y dicho valor y:

$$L_{(\phi)} = \mathbb{E}[(Q_\phi(s,a) - y)^2] \tag{2.15}$$

De esta forma, el crítico aprende a que su estimación $Q_\phi(s,a)$ se parezca lo más posible a y. Si $Q_\phi(s,a)$ da un valor muy distinto a y, el error será grande y los pesos ϕ se ajustarán bastante. Si la diferencia entre ambos es mínima, el ajuste será cada vez más pequeño.

2.4.4.3 Actualización del actor

El actor principal se actualiza en la dirección que maximiza el valor Q asignado por el crítico. Manteniendo la Q y los pesos ϕ del crítico fijos, el gradiente de política que guía la actualización del actor es:

$$\nabla_\theta J(\theta) = E[\nabla_\theta Q_\phi(s, \mu_\theta(s))] \tag{2.16}$$

Es decir, se ajustan los pesos θ del actor para que las acciones $\mu_\theta(s)$ que genere a su salida sean cada vez mejor valoradas por el crítico, y como consecuencia, maximicen el valor de Q [14]. En la práctica, este gradiente se obtiene "retropropagando" el error a través del crítico y posteriormente del actor.

Como DDPG utiliza una política determinista, durante el entrenamiento se añade un ruido aleatorio $\boldsymbol{\varepsilon}$ a las acciones para fomentar la exploración del entorno. Este ruido se elimina una vez finalizado el entrenamiento.

$$a = \mu_\theta(s) + \varepsilon \tag{2.17}$$

2.4.4.4 Redes target

Uno de los desafíos al entrenar redes neuronales usando la ecuación de Bellman (2.14) es el fenómeno conocido como "el problema del blanco móvil" (*moving target problem*). Si el crítico se actualiza basándose en sus propias predicciones para calcular el valor objetivo (la variable $\boldsymbol{y}$) con el que se entrena, dicho objetivo cambiaría en cada actualización de los pesos. Es decir, la red estaría intentando aprender a partir de una referencia que cambia constantemente, lo que puede impedir la convergencia del entrenamiento.

Para solucionar este problema, se introdujeron las redes *target*, que son copias de las redes principales: un actor *target* ($\mu_{\theta targ}$) y un crítico *target* ($Q_{\phi targ}$). Estas redes son las que se utilizan para calcular el valor objetivo $\boldsymbol{y}$ de la ecuación (2.14), proporcionando una referencia más estable para el aprendizaje.

Esto se debe a que las redes *target* se actualizan lentamente mediante un proceso conocido como *actualización suave* (*Polyak averaging*). Aquí se tiene un hiperparámetro τ, normalmente muy pequeño (por ejemplo, 0.001), el cual determina qué fracción de los pesos de la red principal se incorpora a la red *target* en cada iteración. De esta forma, evolucionan de manera gradual y proporcionan objetivos de aprendizaje más estables.

$$\phi_{\text{targ}} \leftarrow \tau\phi + (1-\tau)\phi_{\text{targ}} \tag{2.18}$$

$$\theta_{\text{targ}} \leftarrow \tau\theta + (1-\tau)\theta_{\text{targ}} \tag{2.19}$$

2.4.4.5 Flujo del algoritmo

Si unimos todo, el algoritmo que sigue DDPG en el entrenamiento es el siguiente [14]:

1. El actor principal observa el estado s y genera una acción $a = \mu_\theta(s) + \varepsilon$
2. La acción se ejecuta en el entorno, obteniendo la recompensa r y el nuevo estado s'.
3. La transición (s, a, r, s') se almacena en el replay buffer D.
4. Se extrae un minilote (*mini-batch*) aleatorio de transiciones del buffer D.
5. De ese *mini-batch* extraído se calcula el target $y = r + \gamma \cdot Q_{\phi targ}(s', \mu_{\phi targ}(s'))$
6. Se actualiza el crítico principal minimizando $L_{(\phi)} = \mathbb{E}[(Q_\phi(s, a) - y)^2]$
7. Se actualiza el actor principal ascendiendo por el gradiente $\nabla_\theta J(\theta)$
8. Se actualizan ambas redes target mediante *polyak averaging*.

Una vez concluido el entrenamiento, solo se necesita el actor principal para tomar decisiones, ya que recibe el estado como entrada y devuelve directamente la acción a ejecutar, sin necesidad de las redes *target* ni del crítico.

2.4.4.6 Justificación de la elección del algoritmo DDPG

La elección del algoritmo Deep Deterministic Policy Gradient (DDPG) como núcleo de la estrategia de RL para este trabajo responde a los requisitos cinemáticos de la conducción autónoma. A diferencia de los algoritmos clásicos basados en Q-Learning, como DQN, que están limitados a espacios de acción discretos, el control de un robot móvil exige ajustes continuos y progresivos sobre la velocidad lineal y angular. Discretizar estas acciones obliga al robot a seleccionar comandos rígidos y predefinidos, lo que se traduce en una conducción oscilatoria e inadecuada para un vehículo real.

El DDPG fue diseñado específicamente para operar en espacios de acción continuos. A través de su arquitectura Actor-Crítico, el Actor calcula directamente políticas deterministas que generan valores reales de velocidad lineal y angular con precisión decimal, permitiendo modular los comandos de forma suave y fluida. Esta capacidad es la que ha permitido al agente desarrollar un comportamiento competitivo de forma autónoma en los distintos circuitos, cuyos resultados se explicarán en apartados más adelante.

2.5 Algoritmo ArUco

El sistema ArUco es una herramienta de visión artificial altamente eficiente, la cual está orientada a la detección y seguimiento de objetos a través de marcadores bidimensionales. Estos marcadores se caracterizan por un perímetro cuadrado de color negro que enmarcan un patrón binario interno. La disposición de estos bits (o píxeles) codifica un identificador (ID) unívoco perteneciente a un diccionario preestablecido, por lo que no habrá dos patrones distintos que resulten en un mismo ID. De esta forma, el robot usará su cámara para ir leyendo estos marcadores, pudiendo guardar la posición de estos o ejecutar una determinada acción.

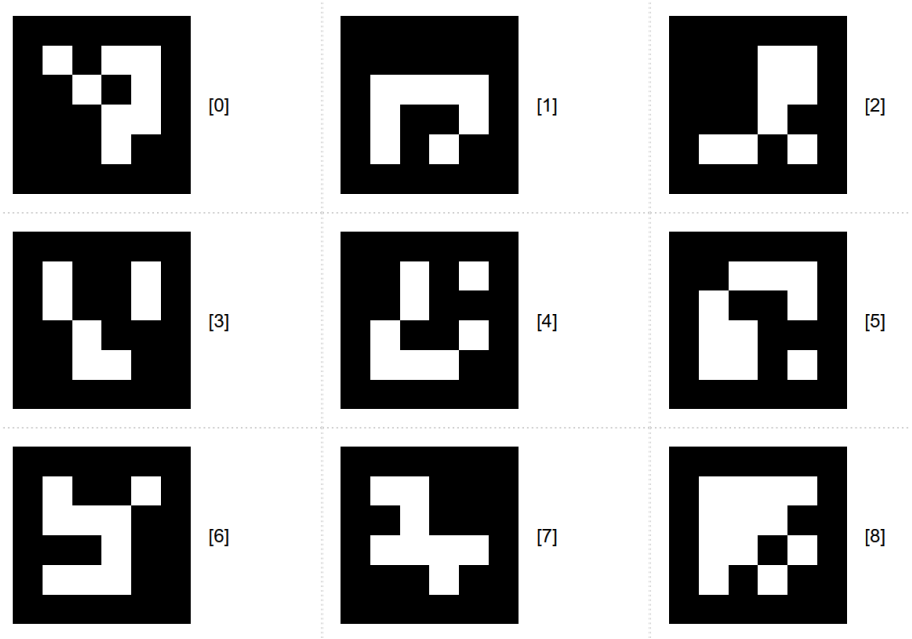


Figura 2.12: Familia ArUco 4x4 250.

El flujo de procesamiento de la imagen para extraer esta información consta de varias etapas secuenciales. En primer lugar, el fotograma capturado por la cámara se convierte a escala de grises y se somete a un algoritmo de binarización mediante umbrales adaptativos, separando las zonas claras de las oscuras. Sobre esta imagen binaria, se ejecuta una búsqueda de contornos y se filtran aquellas formas poligonales cerradas que posean exactamente cuatro vértices, es decir, los potenciales cuadrados. Al aislar un área candidata, el algoritmo aplica una transformación de homografía para corregir la distorsión de perspectiva, orientando la imagen en un plano frontal ideal que permite una lectura directa de la cuadrícula de bits.



Figura 2.13: Homografía de un plano.

En segundo lugar, una vez tenemos el marcador rectificado, se extrae la matriz interna y se compara con los marcadores o identificadores válidos del diccionario cargado en el sistema. Como el marcador es un cuadrado, el sistema comprueba sus cuatro orientaciones posibles. Si encuentra una coincidencia, no solo averigua a que ID corresponde, sino que también descubre exactamente cuánto está girado respecto a la cámara.

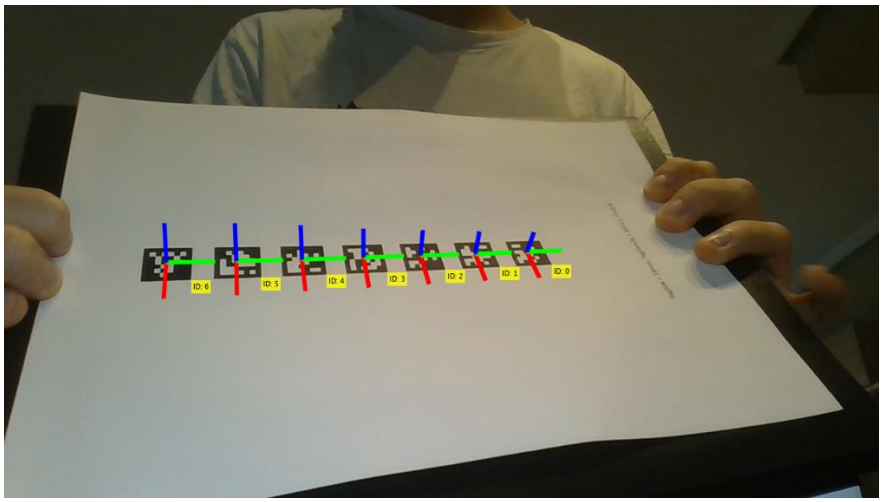


Figura 2.14: Detección simultánea y estimación de la pose 3D de múltiples marcadores ArUco con sus respectivos identificadores (IDs).

La verdadera potencia de ArUco para entornos robóticos radica en su capacidad para la estimación de pose espacial. Si se conoce de antemano la dimensión física real del lado del marcador y los parámetros intrínsecos obtenidos de la calibración de la cámara, el algoritmo es capaz de resolver la perspectiva trigonométrica. Como resultado, se calculan los vectores de traslación y rotación que definen la posición 3D exacta del marcador respecto al sistema de coordenadas de la cámara. Esta información es vital para que el robot sepa dónde está situado en un mapa y hacia dónde debe moverse.

Sin embargo, el rendimiento del algoritmo está condicionado por factores físicos del entorno. Una iluminación deficiente que arruine el contraste, el uso de lentes con muchas distorsiones o cámaras no calibradas pueden comprometer la correcta extracción de los marcadores. A pesar de estas desventajas, si se garantizan unas condiciones operativas mínimas, su facilidad de integración y su alta velocidad de ejecución hacen de ArUco una solución idónea y ampliamente consolidada para la visión artificial.

Capítulo 3

3. Arquitectura general del sistema

En este capítulo se expone la arquitectura general del sistema, analizando el flujo de trabajo entre el entorno de desarrollo en MATLAB y el entorno virtual en MVSim. Luego, se explica el proceso de diseño y mapeado de los circuitos de prueba, así como el modelo cinemático diferencial del *TurtleBot 4*. Toda esta infraestructura constituye la base necesaria para la correcta ejecución y validación de los algoritmos de control, que serán explicados más adelante.

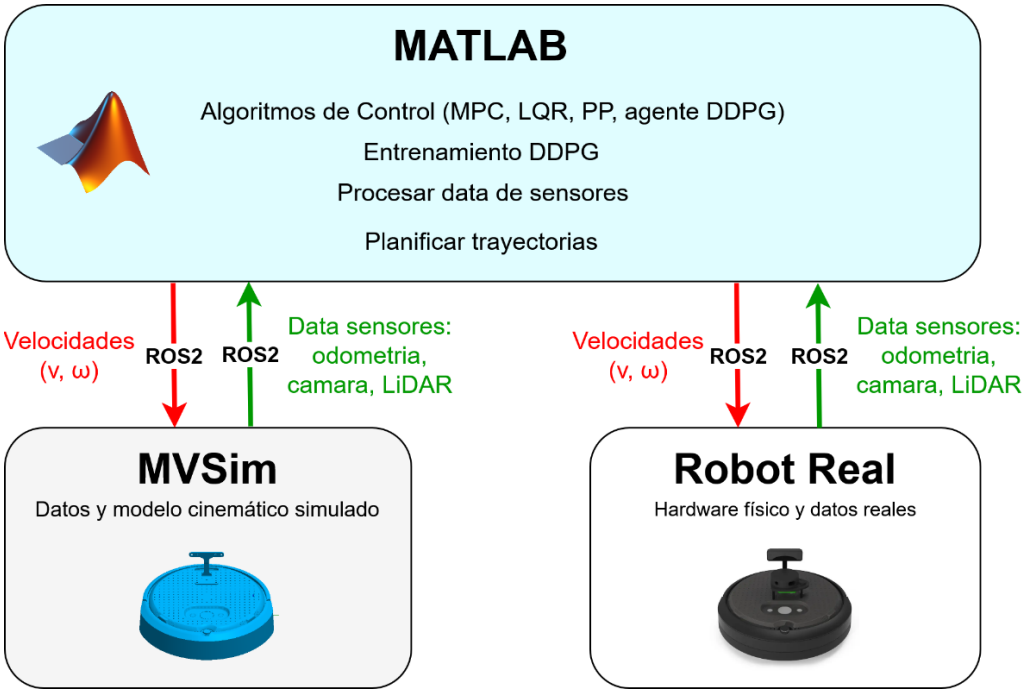


Figura 3.1: Flujo de arquitectura del sistema.

3.1 MATLAB

MATLAB es un entorno de programación y cálculo numérico avanzado, el cual constituye el núcleo computacional del proyecto. En él se ha desarrollado todo el software, abarcando tanto el entrenamiento del agente DDPG mediante aprendizaje por refuerzo (RL), como la planificación y control de trayectorias con los distintos algoritmos.

Para hacer esto posible, MATLAB se comunica de forma directa y bidireccional con el simulador MVSim (o robot real) a través del ecosistema ROS2, tal como se observa en la Figura 3.1. Esta conexión se utiliza explícitamente para enviar los comandos de velocidad (lineal y angular) generados por los algoritmos de control al vehículo y, simultáneamente, recibir de vuelta todos los datos provenientes de los sensores del robot.

Para dotar al sistema de estas capacidades avanzadas de conectividad, robótica e inteligencia artificial sin necesidad de programarlas desde cero, se ha hecho uso de varios módulos especializados de MathWorks, concretamente *ROS Toolbox*, *Reinforcement Learning Toolbox*, *Navigation Toolbox*, *Model Predictive Control Toolbox* y *Control System Toolbox*.

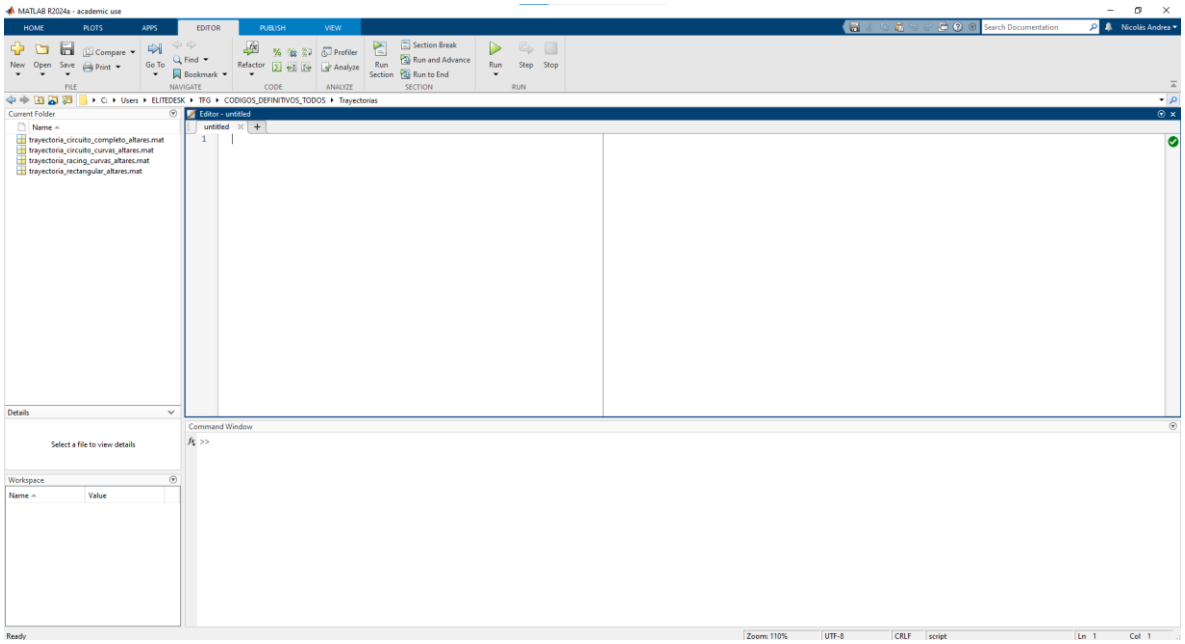


Figura 3.2: Interfaz MATLAB 2024a.

3.2 MVSim

MVSim (*Multi-Vehicle Simulator*) es un simulador de robótica móvil ligero de código abierto, el cual constituye el entorno virtual donde se validan todos los algoritmos de control del proyecto. Su función principal es actuar como la planta física del sistema, representando en tiempo real la cinemática del vehículo (*TurtleBot 4*) y simulando su comportamiento dentro de los circuitos de prueba (Figura 3.3).

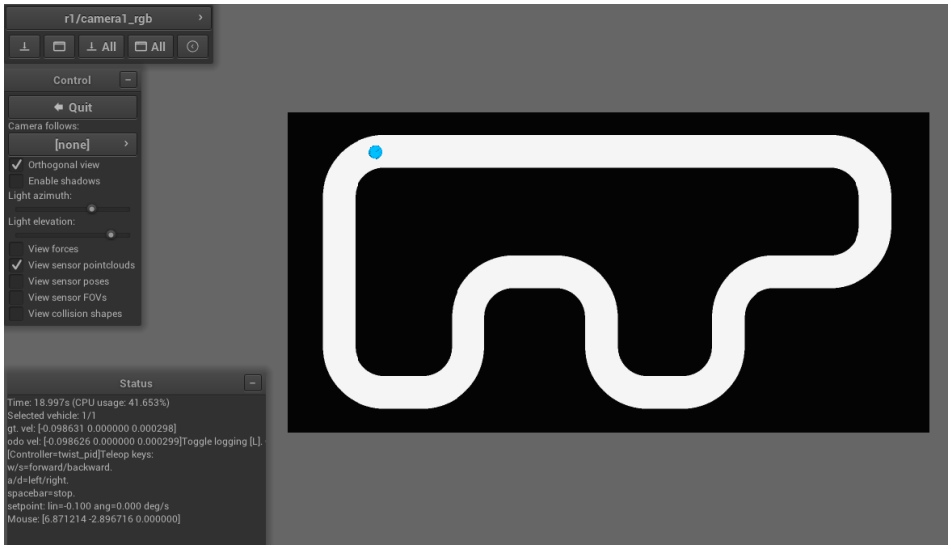


Figura 3.3: Entorno MVSim.

Este simulador se conecta de forma directa con MATLAB a través de ROS2 y se encarga de recibir los comandos de velocidad lineal y angular que calculan los controladores para aplicarlos sobre los actuadores del robot. Al mismo tiempo, procesa y envía de vuelta a MATLAB las lecturas de los sensores simulados, permitiendo que los algoritmos reaccionen al estado del entorno en cada iteración.

La interfaz gráfica de MVSim ofrece un conjunto de herramientas de monitorización que permiten supervisar el estado del sistema en tiempo real (Figura 3.4). Entre sus elementos visuales más destacados se encuentra una ventana dedicada a la cámara RGB integrada en el robot, la cual proyecta la perspectiva en primera persona. Adicionalmente, el simulador incorpora un panel de estado (*Status*) que permite monitorizar las velocidades exactas del robot, observar las consignas de control que se están aplicando en cada instante y consultar parámetros del sistema como el tiempo de simulación o el porcentaje de uso de la CPU.

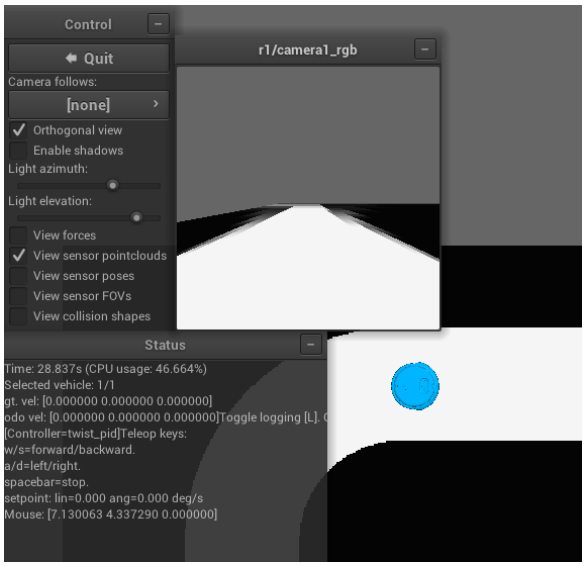


Figura 3.4: Paneles de Información en MVSim.

3.3 Mapeado del entorno

Para llevar a cabo la simulación en MVSim y someter al robot a múltiples escenarios con distintas exigencias, se han creado tres circuitos (mapas) en Paint exportados como imágenes PNG. Dichos circuitos se han diseñado de tal forma que todo el trazado en color blanco representa la pista transitable por la que el robot puede circular libremente (con un ancho de 0.8 metros), mientras que las áreas en color negro son interpretadas por el simulador como paredes físicas o límites inaccesibles del entorno.

Una vez obtenidas estas imágenes, resulta imprescindible traducirlas a un formato matemático que los algoritmos de navegación y MATLAB puedan interpretar espacialmente. Para ello, se ha llevado a cabo un proceso de conversión a rejillas de ocupación (*occupancy grids*). Gracias a este proceso las imágenes PNG se importan y se transforman en matrices bidimensionales, donde cada celda del espacio adquiere un valor binario: espacio libre para la navegación o zona ocupada por un obstáculo. Finalmente, a estas matrices se les aplica una resolución espacial para dotarlas de dimensiones físicas reales coherentes con la cinemática del robot, y se exportan como archivos de datos nativos de MATLAB en formato *.mat*.

A continuación, se ilustran los tres circuitos diseñados tras seguir este proceso.

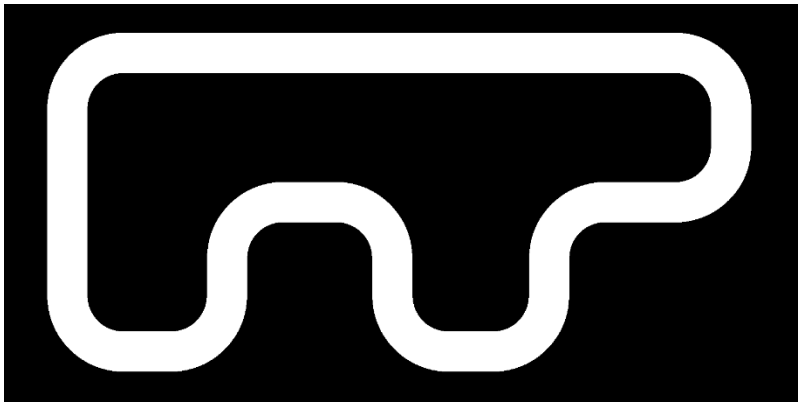


Figura 3.5: Circuito 1 en MVSim.

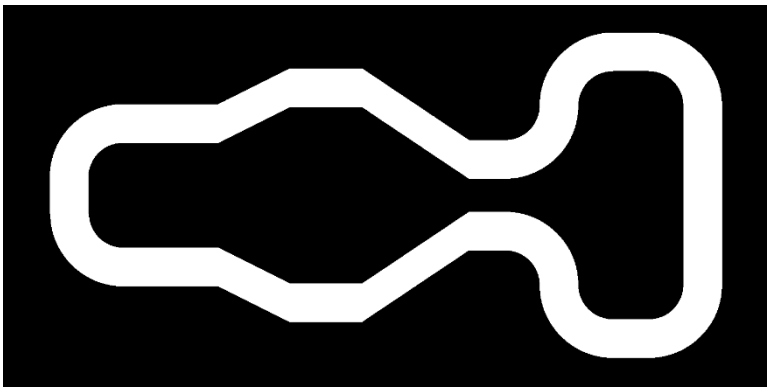


Figura 3.6: Circuito 2 en MVSim.



Figura 3.7: Circuito 3 en MVSim.

3.4 Modelo del TurtleBot 4 Lite

El *TurtleBot 4 Lite* es un robot móvil de tracción diferencial, una plataforma que se caracteriza por disponer de dos ruedas motrices independientes situadas simétricamente a ambos lados del chasis, junto con una o varias ruedas de apoyo sin tracción que simplemente dan estabilidad al conjunto. El movimiento del robot surge de la diferencia de velocidades entre ambas ruedas: si las dos giran a la misma velocidad, el robot avanza en línea recta; si existe una diferencia entre ellas, el robot describe una trayectoria curva. La Figura 3.8 ilustra esta configuración mecánica diferencial y las principales variables de error geométrico involucradas en la navegación.

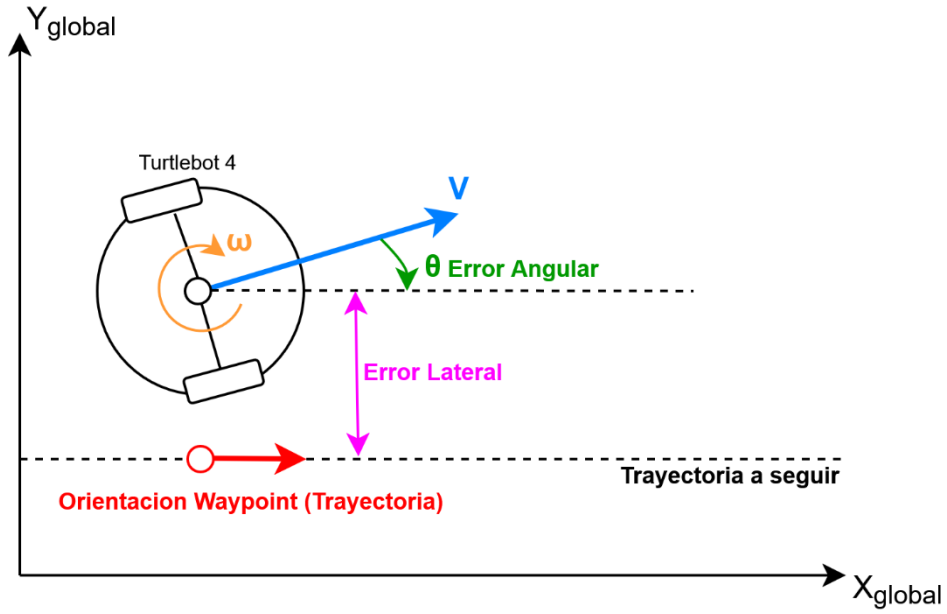


Figura 3.8: Modelo cinemático diferencial del TurtleBot 4 y representación de los errores de trayectoria.

Sin embargo, en la práctica, el controlador de alto nivel no trabaja directamente con la velocidad de cada rueda por separado. En su lugar, el sistema abstrae esta complejidad física y asimila el comportamiento del chasis al de un modelo cinemático tipo monociclo (*unicycle*). Bajo este modelo, el robot recibe únicamente dos entradas de control globales: una velocidad lineal v y una velocidad angular ω. Los algoritmos desarrollados en este proyecto, como el MPC, calculan en cada ciclo de ejecución el par de valores (v, ω), enviándolo al simulador o robot real directamente. Posteriormente, el *firmware* interno del robot se encarga de traducir esos dos valores a las velocidades concretas de cada rueda.

Por consiguiente, desde la perspectiva del diseño de control, la dinámica del robot se modela de forma equivalente a un sistema *unicycle*. Dicho modelo matemático describe la evolución del vehículo a partir de tres variables de estado y las dos entradas de control mencionadas, tal y como se detalla a continuación.

Por un lado, el estado del robot en cada instante queda completamente descrito por su posición en el plano y su orientación [8]:

$$\mathbf{x} = \begin{bmatrix} x \\ y \\ \theta \end{bmatrix} \tag{3.1}$$

donde x e y son las coordenadas del robot en el marco de referencia global, y θ es el ángulo de orientación (*yaw*), medido en sentido antihorario desde el eje X global. Por otro lado, las entradas de control son:

$$\mathbf{u} = \begin{bmatrix} v \\ \omega \end{bmatrix} \tag{3.2}$$

donde v es la velocidad lineal del robot sobre su eje longitudinal, y ω es la velocidad angular alrededor del eje vertical. Descomponiendo la velocidad lineal v en sus componentes según el marco global mediante trigonometría, se obtienen las ecuaciones cinemáticas continuas del robot:

$$\begin{aligned} \dot{x} &= v\cos(\theta) \\ \dot{y} &= v\sin(\theta) \\ \dot{\theta} &= \omega \end{aligned} \tag{3.3}$$

Estas ecuaciones indican que la velocidad con la que el robot avanza en cada dirección del plano depende de cuánto está girado en ese momento. Si el robot apunta exactamente en la dirección del eje $X(\theta = 0)$, toda la velocidad es horizontal; si apunta en la dirección del eje $Y(\theta = \pi/2)$, toda la velocidad se traduce en desplazamiento a lo largo de ese eje.

Por otra parte, para poder implementar este modelo en un controlador como el MPC o LQR, es necesario expresarlo en tiempo discreto. Asumiendo un paso de tiempo Δt suficientemente pequeño (como 0.1 segundos), el estado en el instante t se puede aproximar a partir del estado en el instante anterior $t-1$ de la siguiente forma:

$$\begin{aligned} x_t &= x_{t-1} + v_{t-1}\cos(\theta_{t-1})\,\Delta t \\ y_t &= y_{t-1} + v_{t-1}\sin(\theta_{t-1})\,\Delta t \\ \theta_t &= \theta_{t-1} + \omega_{t-1}\,\Delta t \end{aligned} \tag{3.4}$$

En definitiva, el modelo *unicycle* proporciona una descripción matemática sencilla pero suficientemente precisa del comportamiento cinemático del *TurtleBot 4 Lite*. Además, constituye la base sobre la que se apoyan los algoritmos de control basados en modelos, como MPC, ayudando a predecir la evolución futura del robot y calcular las acciones de control óptimas en cada instante. Una vez definido el modelo del sistema, el siguiente paso consiste en obtener la trayectoria de referencia que el robot deberá seguir, es decir, el conjunto de *waypoints* (puntos) que definen el circuito.

Capítulo 4

4. Planificación de trayectorias

En este capítulo se aborda la fase de planificación global del sistema, ejecutada de manera previa (*offline*) para generar la ruta de referencia del vehículo. En primer lugar, se generan unos *waypoints* para definir el circuito. En segundo lugar, se aplica una interpolación por *spline* para transformar dichos *waypoints* en una trayectoria central suave y continua. En tercer y último lugar, se aplica un algoritmo de optimización que aprovecha los márgenes de la pista para calcular la línea de carrera ideal (*racing line*). Este proceso minimiza la curvatura total del recorrido, permitiendo al vehículo maximizar su velocidad sostenida y reducir el tiempo por vuelta.

4.1 Generación de waypoints

Para definir la trayectoria de referencia central, es necesario obtener una secuencia de puntos distribuidos a lo largo del trazado del circuito, conocidos como *waypoints*. Con el fin de registrar las coordenadas de dichos puntos, se dispone de dos métodos distintos. Por un lado, estos se pueden capturar a distancias equiespaciadas, de forma que todos se encuentren separados por un mismo intervalo (por ejemplo, cada 0.5 metros). Por otro lado, se pueden emplear marcadores ArUco colocados sobre vallas, permitiendo que el robot detecte dichos marcadores mediante su cámara y posicione un *waypoint* en la coordenada fija de la valla. Para la ejecución de ambos métodos, se ha optado por una estrategia común de teleoperación manual del robot desde el ordenador. Esto nos permite mantener un control total sobre el movimiento del vehículo, guiándolo para que describa una trazada lo más centrada posible por la pista y garantizando un margen de espacio similar respecto a ambos bordes.

A partir de este guiado manual, el primer método funciona registrando automáticamente los *waypoints* a intervalos regulares de 0.5 metros apoyándose en los datos de odometría. El proceso se inicializa almacenando el punto de partida en la coordenada $(0,0)$, la cual pasa a ser la primera posición conocida de referencia. A medida que el robot avanza, un bucle captura continuamente las coordenadas espaciales X e Y actuales y evalúa la distancia euclidiana respecto al último punto guardado (en lugar de hacerlo respecto al origen absoluto del circuito), aplicando para ello la siguiente expresión:

$$d = \sqrt{(X_{actual} - X_{ultimo})^2 + (Y_{actual} - Y_{ultimo})^2} \tag{3.1}$$

Mientras la distancia d sea inferior a 0.5 metros, el algoritmo continúa con la siguiente iteración del bucle sin registrar cambios. Sin embargo, en el instante en que se detecta que dicha

distancia es igual o superior a 0.5 metros respecto al último punto almacenado, las coordenadas actuales se guardan en la matriz de *waypoints*. Acto seguido, el sistema actualiza la variable de la última posición de referencia con las coordenadas actuales, asegurando que la próxima evaluación se realice respecto a este nuevo punto. Mediante este procedimiento dinámico, se van almacenando secuencialmente todos los *waypoints* cada medio metro hasta completar el recorrido del circuito, tal y como se ilustra en la Figura 4.1.

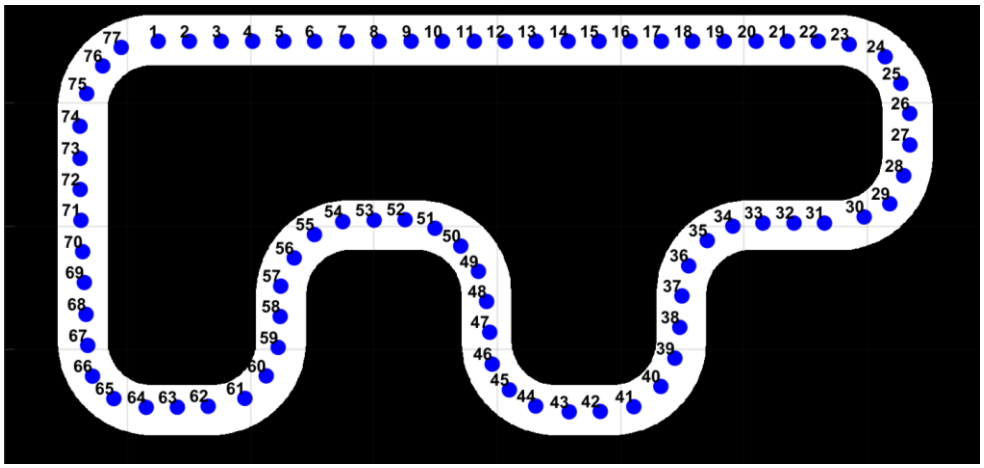


Figura 4.1: Waypoints obtenidos mediante distancias equiespaciadas para el circuito 1.

La segunda alternativa para generar los waypoints de la trayectoria de referencia (central), es usar los marcadores ArUco mencionados previamente en el marco teórico. Este método aporta una gran versatilidad al sistema, ya que permite planificar nuevas trayectorias de forma dinámica con tan solo reubicar físicamente los marcadores o las vallas del entorno.

Para que el robot real pueda identificar y leer correctamente estos marcadores, se coloca cada uno de ellos en el centro de las vallas en orden cronológico (del 1 al 12), tal como se observa en la Figura (4.2). Mediante la teleoperación del robot desde nuestro ordenador, lo iremos manejando para que vaya lo más centrado por las vallas mientras, de forma simultánea, se ejecuta en MATLAB un algoritmo que va detectando dichos marcadores.

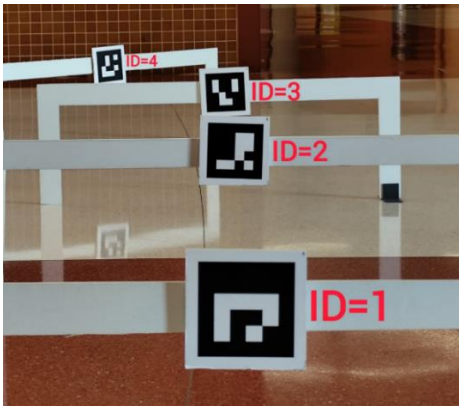


Figura 4.2: Códigos ArUco ordenados cronológicamente en las vallas.

Antes de empezar a operar el robot, se debe colocarlo en la posición inicial antes de la primera valla que contiene el ID 1. Al ejecutar el script de MATLAB, el sistema queda a la espera de recibir el primer mensaje de odometría del robot. Una vez recibido, extrae la posición espacial y la orientación para establecer ese punto exacto como el origen de coordenadas absoluto del mapa.

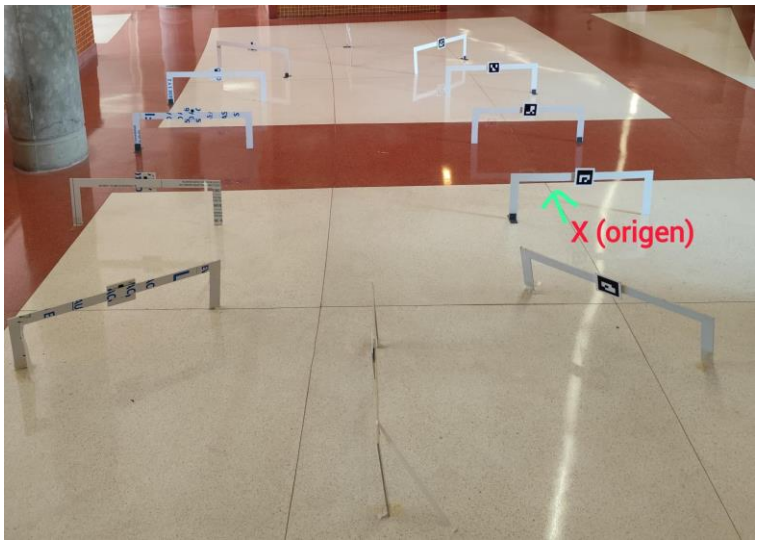


Figura 4.3: Vallas del circuito a recorrer.

Mientras el robot es teleoperado por el circuito, la cámara captura imágenes de forma constante. El algoritmo analiza cada una buscando marcadores de la familia ArUco especificada (en nuestro caso la familia 4x4 250). Si se detectan varios simultáneamente, el sistema itera sobre ellos y selecciona únicamente aquel con la menor traslación en el eje de la cámara, garantizando que el robot siempre apunte al objetivo más cercano. Además, se aplica un filtro de proximidad que descarta cualquier lectura que supere una distancia de 3 metros, evitando así detecciones erróneas o lecturas de vallas lejanas pertenecientes a otras zonas de la pista.

Cuando la cámara detecta una valla, el algoritmo obtiene su posición de forma relativa; es decir, calcula a qué distancia se encuentra el marcador con respecto al robot. Sin embargo, para construir el mapa del circuito, se debe traducir esta lectura a coordenadas globales referenciadas al origen del sistema. Para lograrlo, el sistema fusiona los datos visuales con la odometría. Utilizando la ubicación exacta del robot y su ángulo de orientación, el algoritmo aplica una transformación trigonométrica que proyecta esa distancia relativa sobre el plano general. De esta forma, la valla deja de ser un punto "frente al robot" (local) y se guarda de forma definitiva como una coordenada global fija en el mapa del circuito.

Por último, para evitar que el robot sature la memoria guardando la misma valla cientos de veces por segundo, se implementa una lógica de descarte por identificador. Si el ID del marcador detectado no existe en el array de IDs guardados, sus coordenadas globales calculadas se guardan y se muestra un recuadro de color verde sobre el propio marcador, indicando que es

nuevo (Figura 4.4). Si el ID ya fue detectado previamente, el algoritmo simplemente lo ignora visualizándolo en amarillo (Figura 4.5).



Figura 4.4: Marcador con ID nuevo.



Figura 4.5: Marcador con ID ya registrado.

Una vez se ha pasado por todas las vallas (12 en total), las coordenadas de los *waypoints* que conforman la trayectoria se exportan como un archivo *.mat*. En la Figura 4.6 se muestran dichos puntos, donde los segmentos lineales que los unen sirven únicamente como apoyo visual para representar la secuencia y el contorno del circuito.

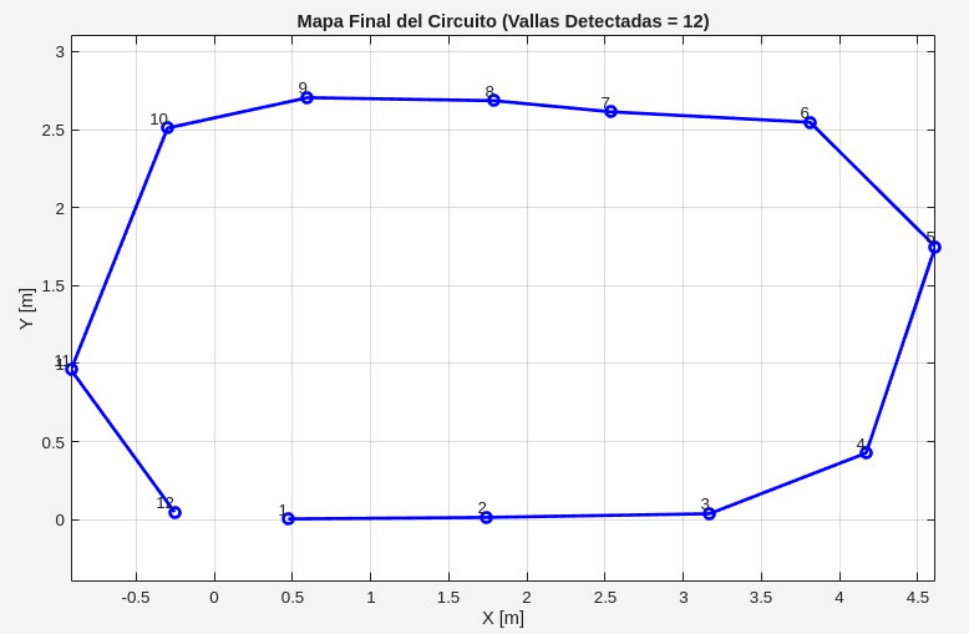


Figura 4.6: Waypoints obtenidos de las vallas en sus coordenadas globales.

4.2 Interpolación lineal

Como primera aproximación para unir los *waypoints* generados anteriormente, se puede plantear el uso de una interpolación lineal, la cual consiste en conectarlos mediante segmentos completamente rectos. Aunque a nivel computacional resulta ser una técnica rápida y sencilla, presenta desventajas críticas para la navegación de plataformas de tracción diferencial. La principal limitación radica en la discontinuidad geométrica que se genera en los vértices de unión, donde la orientación de la trayectoria cambia de forma instantánea. Para poder seguir este trazado, el controlador del vehículo se vería obligado a someter a los motores a correcciones bruscas y oscilatorias continuas. Debido a estas importantes limitaciones que comprometen la fluidez y estabilidad del sistema, se hace indispensable descartar este enfoque lineal en favor de una solución matemática superior que suavice estas transiciones: la interpolación por *splines* cúbicos.

4.3 Splines

Tal y como se ha justificado, el primer paso para la construcción de la trayectoria definitiva a partir de los *waypoints* generados, consiste en aplicar esta técnica matemática. El algoritmo de *spline* se encarga de enlazar cada par de puntos contiguos utilizando ecuaciones polinómicas de tercer grado, con el objetivo de garantizar una continuidad de segundo orden (C^2) en los nodos de unión. En otras palabras, calcula la trayectoria asegurando que tanto la orientación (primera derivada o pendiente) como la curvatura (segunda derivada) sean idénticas al entrar y salir de cada punto, transformando un trazado inicialmente poligonal, propio de la interpolación lineal, en una curva geométrica completamente suave. La Figura 4.7 presenta una comparativa de los resultados generados por los trazadores de primer orden (interpolación lineal), segundo orden y el *spline* cúbico sobre un mismo conjunto de puntos.

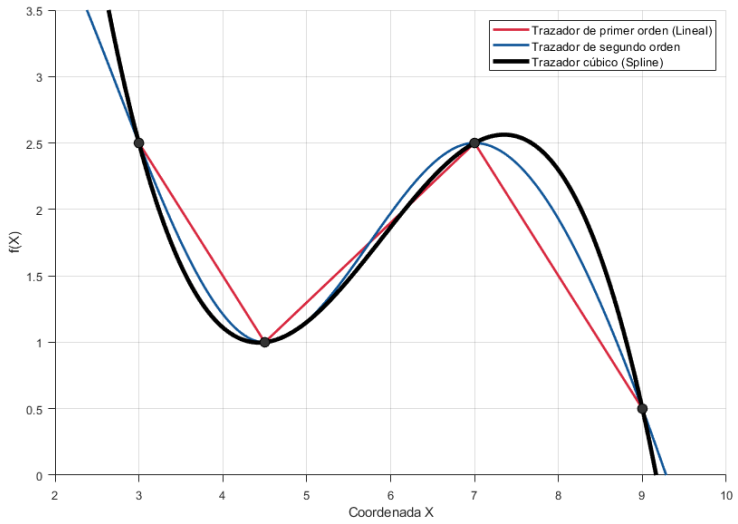


Figura 4.7: Comparación de trazadores de distinto orden.

Una vez modelada la trayectoria suave de base, el algoritmo incrementa aún más su calidad mediante un proceso de discretización en el que se inserta un nuevo punto intermedio exactamente en la mitad de cada tramo original, logrando duplicar la densidad de la ruta. Esta alta resolución espacial con *waypoints* cada 0.25 metros suaviza todavía más la trayectoria de referencia que recibe el vehículo, lo que les permite a los algoritmos de control ejecutar correcciones más precisas sobre los actuadores. Para llevar a cabo esto, la generación de dicha trayectoria para cualquier circuito se divide en los siguientes pasos secuenciales:

En un circuito, las coordenadas espaciales X e Y no pueden interpolarse directamente una respecto a la otra, ya que el vehículo puede retroceder en un eje mientras avanza en el otro. Para solucionar esto, se introduce un parámetro de índice ficticio, equivalente a un "tiempo de recorrido" (t), que podemos ver como un simple índice de orden. Por un lado, tenemos un vector original con números enteros (1, 2, 3...) asociado a cada *waypoint* inicial, por lo que si tenemos 70 *waypoints* el vector irá del 1 al 70. Por otro lado, se crea un nuevo vector de alta resolución con saltos de 0.5 (1, 1.5, 2, 2.5...), lo que le dice que mantenga los puntos viejos (los enteros) y calcule un punto nuevo exactamente en la mitad de cada tramo (los decimales). Gracias a esto logramos duplicar la densidad de la matriz.

```
% Vector de tiempo/índice original (1, 2, 3...)
t_orig = 1:N_orig;

% Nuevo vector de índices con pasos de 0.5 (1, 1.5, 2, 2.5...)
t_nuevo = 1:0.5:N_orig;
```

Listing 4.1: Spline 1

Una vez definida la referencia paramétrica, el algoritmo aborda el cálculo matemático aislando los ejes. Utilizando la función de interpolación por *splines* cúbicos, se calculan de forma independiente cómo varían las coordenadas X e Y a lo largo de ese "tiempo" t. Al evaluar estos polinomios generados en los instantes intermedios (los números decimales), se obtienen las nuevas coordenadas geométricas que reposan exactamente sobre la curva ideal.

```
% Interpolamos X e Y usando 'spline' para que las curvas sean perfectas
X_nueva = interp1(t_orig, wp_orig(:,1), t_nuevo, 'spline')';
Y_nueva = interp1(t_orig, wp_orig(:,2), t_nuevo, 'spline')';
```

Listing 4.2: Spline 2

Para que un robot móvil pueda seguir la trayectoria, no es suficiente con indicarle la coordenada geométrica (X, Y), ya que también necesita conocer la orientación θ que debe adoptar al pasar por ese punto. Para calcularlo, el algoritmo obtiene la diferencia espacial entre cada nuevo *waypoint* y el siguiente $(X_{i+1} - X_i)$. Con estos incrementos (el avance en X y el avance en Y), se aplica la función arcotangente, la cual devuelve la tangente exacta a la curva del *spline*, es decir, la dirección (ángulo) hacia la que apunta la pista en ese milímetro.

```
8   dx = diff(X_nueva);
9   dy = diff(Y_nueva);
10  theta_nueva = atan2(dy, dx);
```

Listing 4.3: Spline 3

De esta manera, se pueden interpolar los *waypoints* centrales originales que habíamos obtenido, generando así una trayectoria suavizada por la que puede circular el robot. Los resultados de interpolar por *spline* el circuito 1, 2 y 3 se observan en las Figuras 4.8, 4.9 y 4.10, respectivamente.

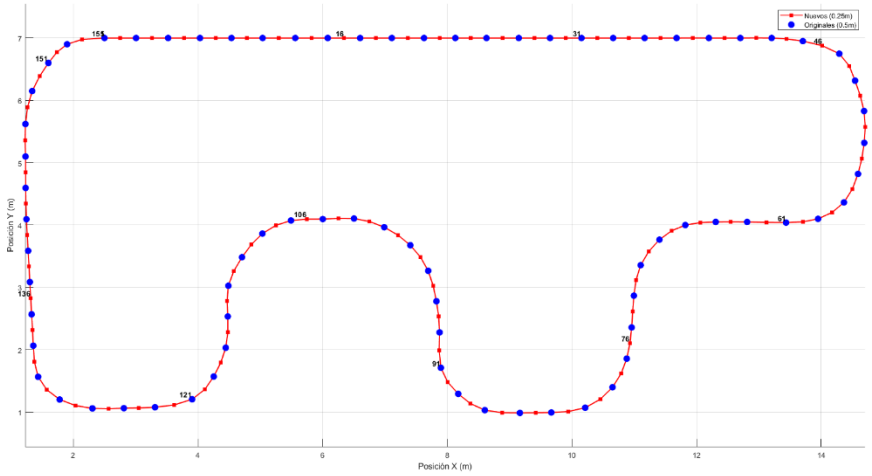


Figura 4.8: Trayectoria del circuito 1 interpolada con Spline.

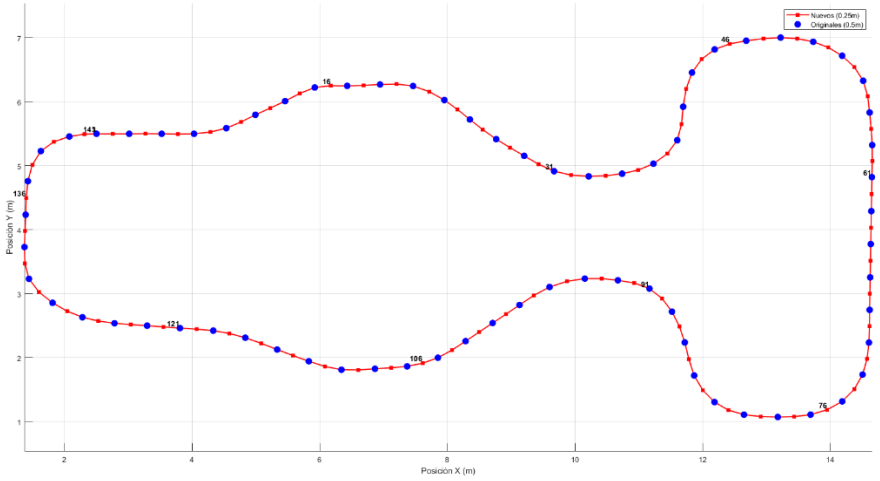


Figura 4.9: Trayectoria del circuito 2 interpolada con Spline.

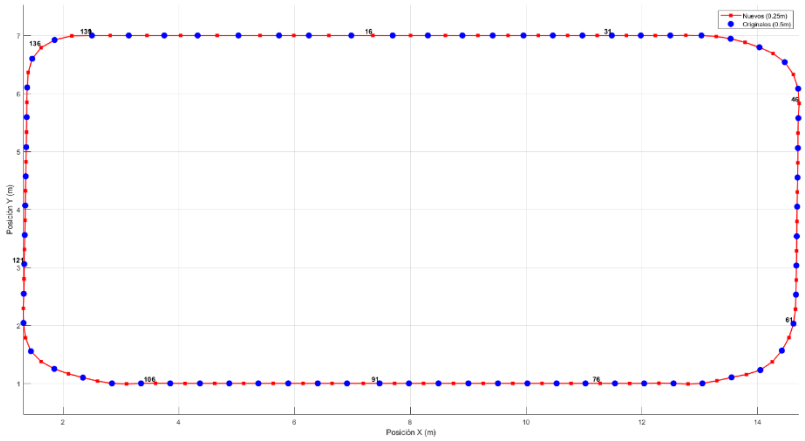


Figura 4.10: Trayectoria del circuito 3 interpolada con Spline.

4.4 Optimización de trayectorias

La metodología expuesta en los apartados anteriores ha permitido generar los waypoints, y mediante la interpolación por *spline* se ha obtenido la trayectoria central suavizada que los une. No obstante, si bien esta trayectoria central representa la alternativa más conservadora al maximizar la distancia a ambos márgenes de la pista, no constituye la solución óptima en términos de tiempo por vuelta. El propósito central de esta fase del trabajo consiste en determinar la trayectoria de carreras ideal (*racing line*). Siguiendo los principios expuestos por autores como Heilmeier *et al.* [15] y especialmente Tipping *et al.* [16], esta trayectoria se calcula minimizando la curvatura global del trazado, lo que permite maximizar la velocidad de paso por curva y, por tanto, reducir al mínimo la necesidad de deceleración del vehículo.

En el ámbito del automovilismo de alta competición, como la Fórmula 1, la estrategia fundamental para lograr esta dinámica consiste en trazar el arco más amplio posible. Esto requiere aproximarse por el extremo exterior de la pista antes de llegar a la curva, intersecar su vértice interno (conocido como *ápex*) y salir abriéndose nuevamente hacia el extremo exterior. Esta estrategia de trazada (exterior-interior-exterior) tiene como finalidad última maximizar el radio de curvatura local.

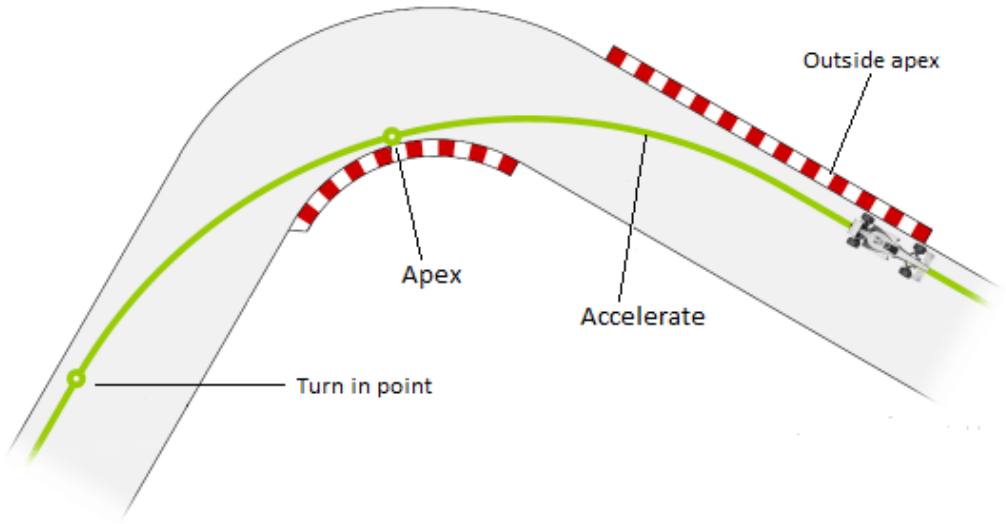


Figura 4.11: Trazada óptima mostrando el punto de giro, el ápex y la zona de salida (exterior-interior-exterior). [17]

La necesidad de incrementar este radio de giro tiene una justificación física directa. En la dinámica clásica de vehículos, un automóvil que traza una curva está limitado por la fuerza lateral máxima (F_{max}) que sus neumáticos pueden soportar por fricción antes de perder adherencia. La velocidad máxima sostenible $v(s)$ en un punto s de la trayectoria está dictada por la relación entre esta fuerza lateral, la masa del vehículo m y la curvatura de la trayectoria $\kappa(s)$:

$$v(s) \leq \sqrt{\frac{F_{max}}{m \cdot |\kappa(s)|}} \tag{4.1}$$

Si agrupamos los parámetros físicos del vehículo en una constante de aceleración lateral máxima ($K_{max} = F_{max}/m$), nos damos cuenta de que la velocidad segura de paso por curva está únicamente acotada por la geometría del trazado $\kappa(s)$:

$$v_{max}(s) = \sqrt{\frac{K_{max}}{|\kappa(s)|}} \tag{4.2}$$

donde vemos que la única forma de maximizar la velocidad $v(s)$ sin alterar las limitaciones físicas del vehículo (su agarre o su masa) es minimizar el valor de la curvatura $\kappa(s)$ a lo largo de la ruta. Aunque el robot empleado en este trabajo opera con un modelo cinemático diferencial sencillo y no tiene en cuenta fuerzas de fricción complejas, la idea básica es la misma. Al garantizar una trayectoria de mínima curvatura, se reduce el esfuerzo de los motores, lo que le permite mantener velocidades más altas, y por tanto, reducir el tiempo de vuelta.

Para alcanzar este objetivo y replicar dicha estrategia, en este trabajo se ha desarrollado una formulación basada en optimización no lineal directa. Esta elección nos permite calcular la

curvatura de forma geométricamente exacta, sin aproximaciones, y sin necesidad de un modelo dinámico del vehículo, lo cual es adecuado para un robot diferencial de bajo nivel en lugar de un coche de carreras real. El algoritmo desarrollado genera esta trayectoria óptima siguiendo un proceso secuencial de tres fases:

En primer lugar, como ya se ha mencionado previamente, el circuito está definido por una trayectoria central discreta compuesta por N puntos (*waypoints*). Para evitar que el robot colisione con las paredes físicas del entorno, se establecen unos límites de seguridad. Dado un ancho de *semipista* de 0.40 metros, y sabiendo que el *turtlebot 4 lite* tiene un semiancho de 0.17 metros, este tendrá **0.23 metros** libres a cada lado del circuito antes de que colisione con una pared lateral. Por este motivo, se ha decidido poner el límite de seguridad a 0.15 metros, dejando únicamente 8 centímetros entre robot y pared en los casos cuando la trayectoria vaya por estos límites (Figura 4.13).

Una vez definidos estos límites, se deben calcular los puntos óptimos que pertenezcan a la trayectoria ideal, siempre y cuando estén dentro de dichos límites. Cada punto P_i de la trayectoria óptima se define como un desplazamiento ortogonal desde el punto central correspondiente C_i, según la siguiente ecuación:

$$P_i = C_i + \alpha_i \cdot w \cdot \hat{n}_i \tag{4.3}$$

donde:

- C_i son las coordenadas (x, y) del *waypoint* central (inicial).
- $\hat{n}_i$ es el vector normal (perpendicular) a la trayectoria en el punto i.
- w es el ancho de la semipista.
- α_i es el coeficiente de desplazamiento asociado a cada *waypoint.*

α_i actúa como la variable de decisión principal del problema de optimización. Este parámetro determina tanto la magnitud como el sentido de la desviación lateral del robot respecto al eje central del carril. De este modo, un valor de α_i = 0 indica que el punto se sitúa exactamente sobre la línea central geométrica de la pista. Por el contrario, un valor de α_i= 0.375 desplaza de forma ortogonal el punto hacia el límite de seguridad derecho (alcanzando el margen máximo permitido de 0.15 metros), mientras que un valor de α_i= -0.375 representa el desplazamiento simétrico hacia el límite de seguridad izquierdo (Figura 4.12). Este valor se obtiene de la relación entre el desplazamiento máximo de seguridad permitido (0.15 metros) y el semiancho total de la pista (0.4 metros), es decir, $0.15 \,/\, 0.4 \;=\; 0.375$.

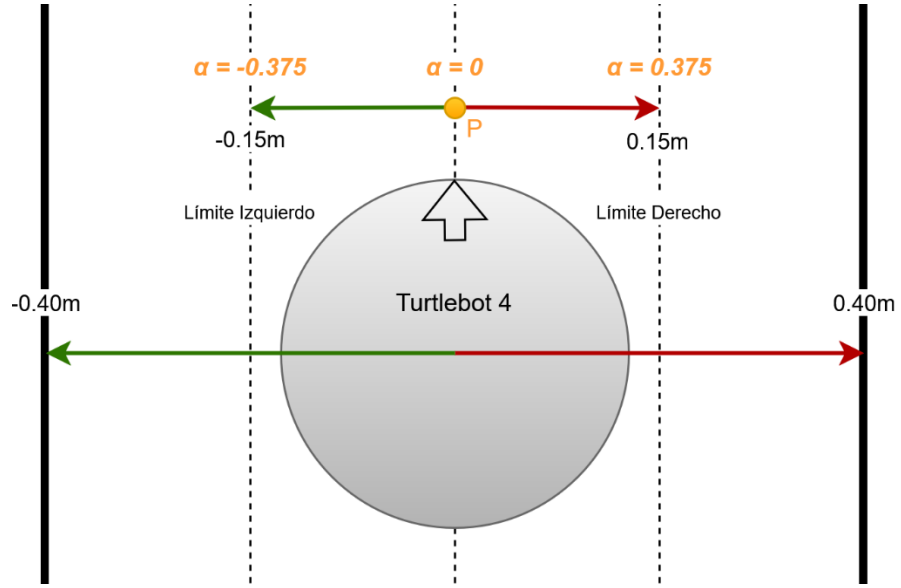


Figura 4.12: Relación entre la variable de decisión α y los límites físicos del robot en la pista.

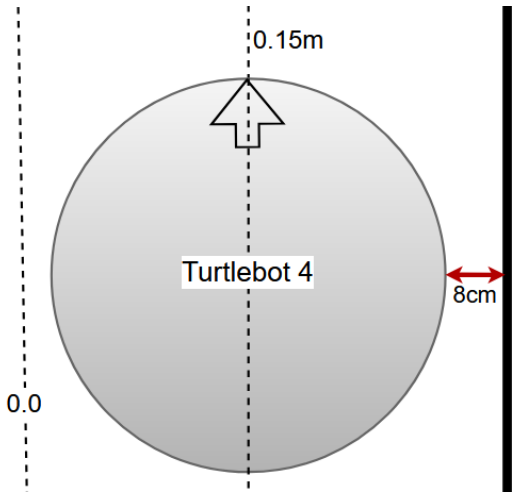


Figura 4.13: Representación del espacio libre restante cuando el centro del robot se sitúa en el límite derecho.

En segundo lugar, para dar comienzo al algoritmo, el vector completo de variables de decisión se inicializa en cero ($\alpha_i = 0$), lo que significa que el optimizador toma la trayectoria central del circuito como la configuración base y que los puntos no han sido movidos todavía. A partir de este estado inicial, el objetivo del optimizador es explorar el espacio de búsqueda acotado para encontrar el conjunto de valores óptimos de α que minimicen de forma global la función de coste J:

$$J = \sum_{i=1}^{N} \kappa_i^2 \cdot (d_{AB} + d_{BC}) \tag{4.4}$$

Donde κ_i es la evaluación exacta de la curvatura. Para cualquier tramo formado por tres puntos consecutivos (P_{i-1}, P_i, P_{i+1}), la curvatura κ_i se calcula utilizando el teorema del circunradio del triángulo que forman:

$$\kappa_i = \frac{4 \cdot A}{d_{AB} \cdot d_{BC} \cdot d_{AC}} \tag{4.5}$$

donde A es el área del triángulo formado por los tres puntos, y los términos del denominador representan las distancias euclidianas entre ellos. Geométricamente, la curvatura κ_i es inversamente proporcional al radio R de esta circunferencia que pasa por 3 puntos, cumpliéndose en todo momento la relación $\kappa_i = \frac{1}{R}$. Si los tres puntos consecutivos conforman un giro muy cerrado, la circunferencia que pasa por ellos tendrá un radio R pequeño (Figura 4.14), lo que dispara el valor de la curvatura κ_i y, en consecuencia, penaliza fuertemente la función de coste global J.

Por el contrario, al hacer la curva más abierta aproximándola a una recta (Figura 4.15), el radio R de la circunferencia que pasa por los 3 puntos se hace significativamente más grande, lo que reduce drásticamente el valor de κ_i y minimiza la función J. De esta forma, al lograr un radio de giro mayor conseguimos una trayectoria mucho más suave, garantizando un seguimiento más estable a mayores velocidades.

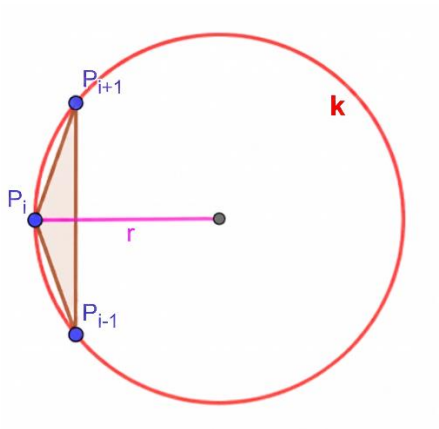


Figura 4.14: Curva cerrada, radio pequeño

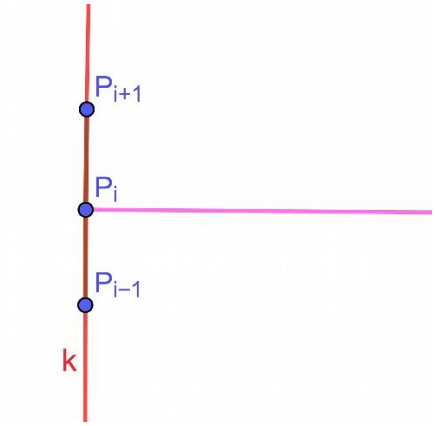


Figura 4.15: Curva grande o recta, radio enorme

En tercer y último lugar, con el espacio de búsqueda acotado y la función de coste definida, se emplea el solucionador *fmincon* de MATLAB para hallar el vector óptimo de desplazamientos α. El objetivo del optimizador es "aplanar" la trayectoria desplazando los puntos hacia los márgenes de la pista. Para hacer esto, evalúa iterativamente distintas configuraciones laterales hasta converger en la solución que minimiza la curvatura global (J) sin exceder el límite de 15 cm de desplazamiento lateral.

El resultado final es una “trayectoria de carreras” (Figura 4.18, Figura 4.19 y Figura 4.20) que aprovecha todo el ancho útil de la pista para trazar los vértices de la forma más plana posible, ideal para ser seguida posteriormente por los controladores que usaremos.

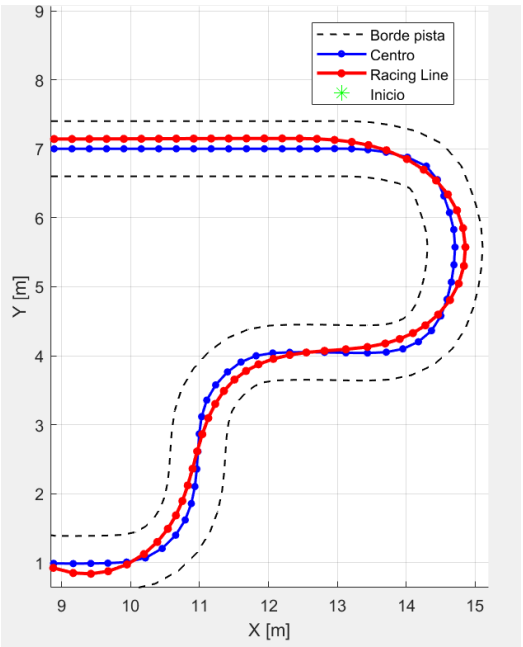


Figura 4.16: Zoom de la trayectoria ideal (roja) con respecto a la central (azul)

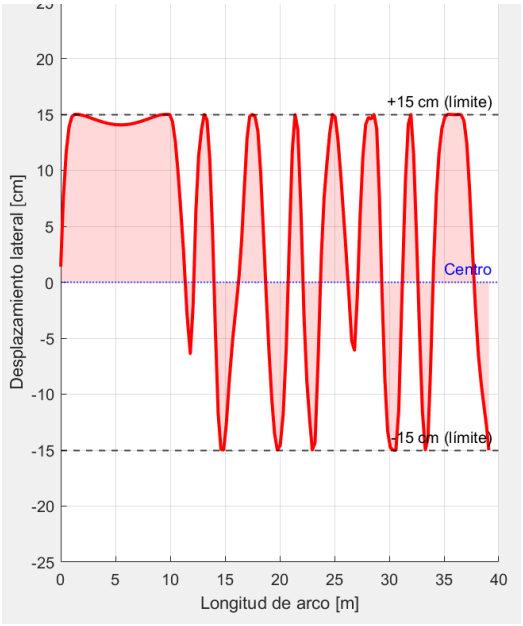


Figura 4.17: Desplazamiento lateral a lo largo del circuito 1.

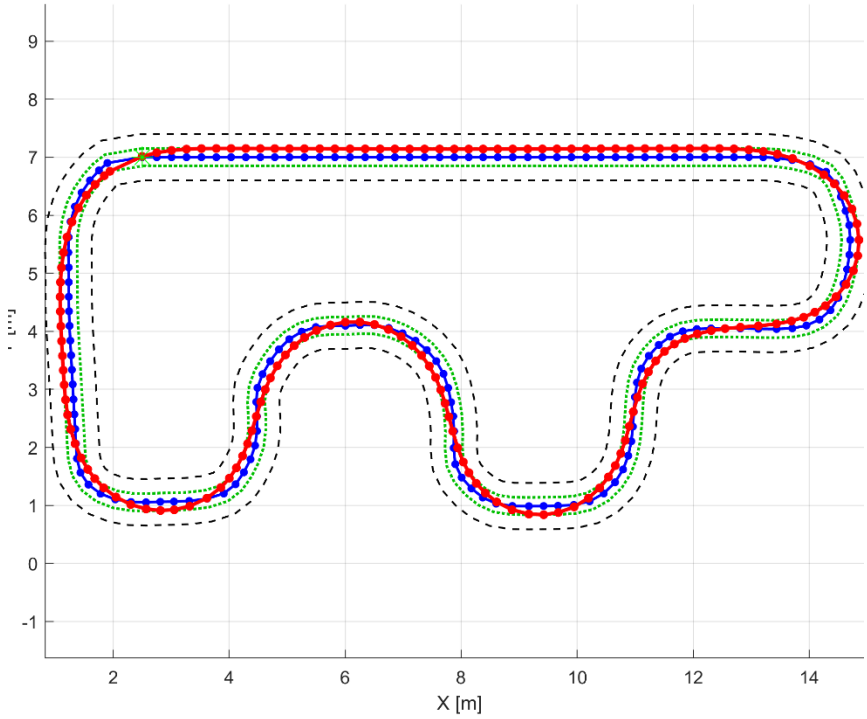


Figura 4.18: Trayectoria ideal (roja), trayectoria central (azul) y límites (verde) del circuito 1.

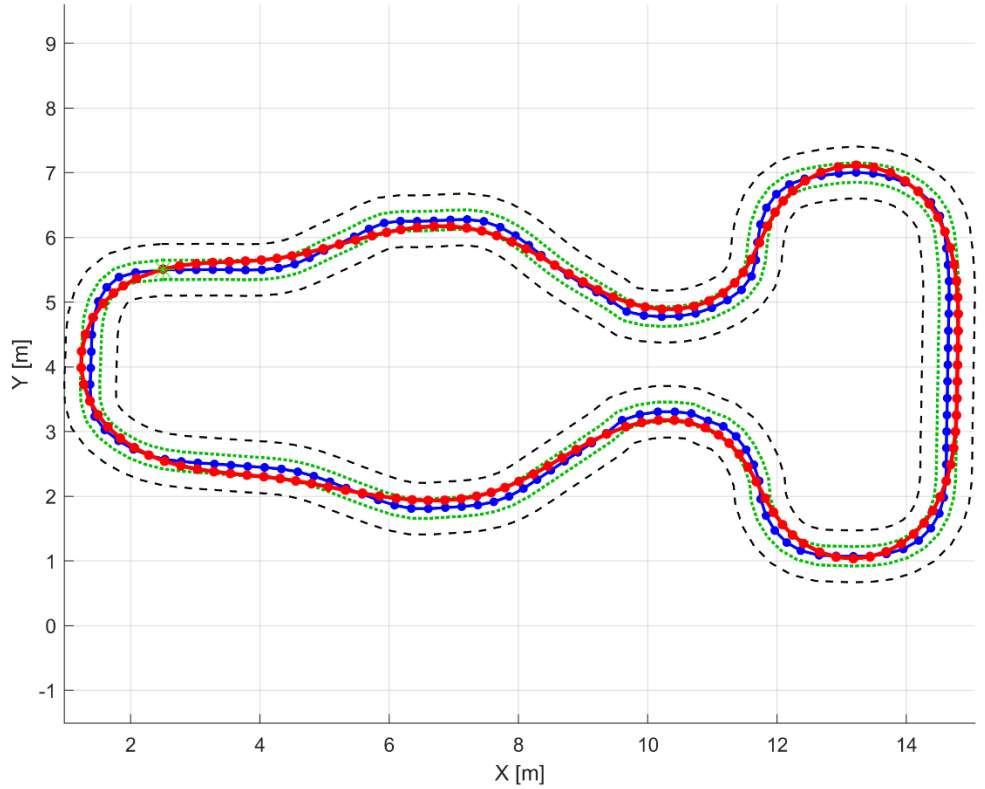


Figura 4.19: Trayectoria ideal (roja), trayectoria central (azul) y límites (verde) del circuito 2.

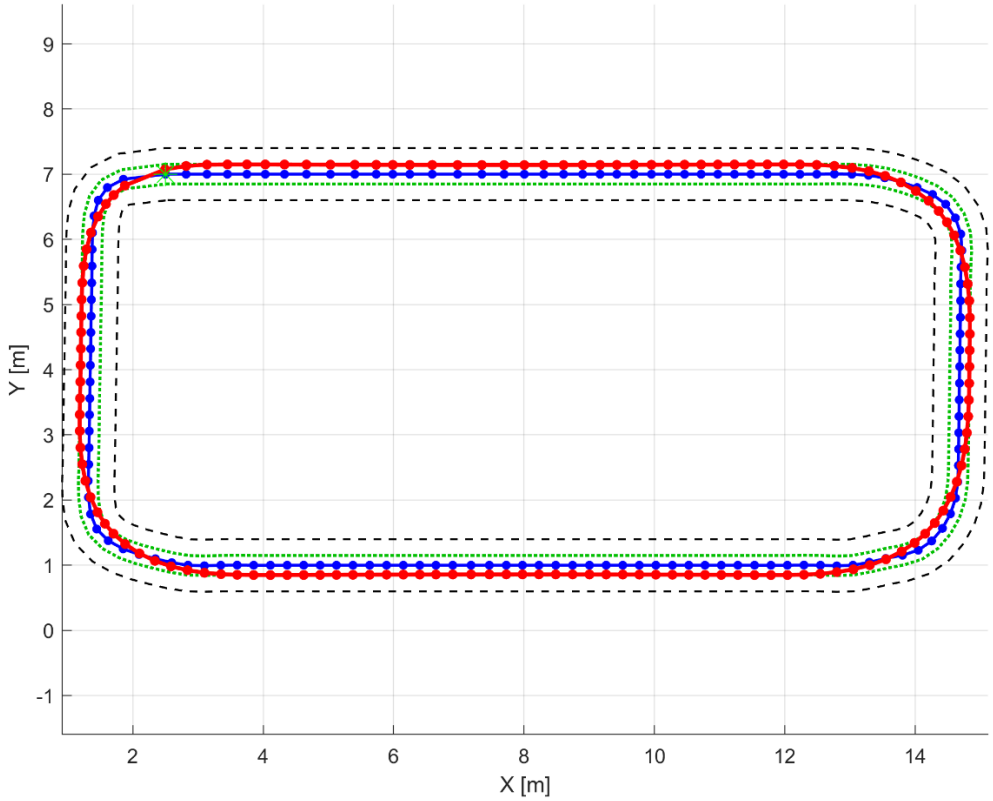


Figura 4.20: Trayectoria ideal (roja), trayectoria central (azul) y límites (verde) del circuito 3.

Capítulo 5

5. Control

En este capítulo se aborda el desarrollo y la implementación práctica de las diferentes estrategias diseñadas para el seguimiento de trayectorias del robot. Con el fin de evaluar y comparar distintas metodologías de navegación autónoma, en los siguientes apartados se detallará la configuración de cuatro algoritmos de control: por un lado, los algoritmos clásicos y analíticos MPC, LQR y Pure Pursuit; y por otro lado, una aproximación basada en RL mediante el algoritmo DDPG.

5.1 Implementación MPC

El primer paso en la implementación del algoritmo MPC consiste en definir el modelo de predicción interno que utilizará el controlador para estimar los estados futuros del sistema. Para ello, se toma como base el modelo cinemático discreto del robot de tracción diferencial descrito en el apartado 3.2, configurando un tiempo de muestreo constante de $T_s = 0.1$ segundos. Debido a que esta cinemática emplea funciones trigonométricas para determinar la posición y orientación del vehículo, la evolución de sus estados es no lineal. Utilizar un MPC "clásico" obligaría a linealizar estas ecuaciones, introduciendo errores de aproximación significativos en las curvas. Por este motivo, el desarrollo se ha realizado mediante el objeto `nlmpc` del *Model Predictive Control Toolbox* de MATLAB, herramienta que permite integrar las ecuaciones exactas del robot sin simplificaciones previas. De este modo, y tal como se explicó en el marco teórico, el optimizador resuelve la función de coste proyectando los estados futuros con la física real del sistema, lo que garantiza la máxima precisión en las acciones de control.

Establecida esta arquitectura base, se procede a configurar los parámetros de diseño del optimizador. Primero se debe establecer el horizonte de predicción (HP), el cual define la ventana temporal que el controlador simula hacia el futuro en cada iteración. En la estrategia implementada, el algoritmo persigue secuencialmente el siguiente *waypoint* activo de la ruta, estableciéndolo como su objetivo local. Al asignar un horizonte de $HP = 10$ y operar con un tiempo de muestreo de $T_s = 0.1$ segundos, el MPC simula los próximos 10 pasos temporales, lo que equivale a 1 segundo de visión futura. Durante este segundo, el controlador optimiza la secuencia de comandos de velocidad necesaria para que el robot converja hacia dicho *waypoint* de la forma más suave y eficiente posible. Adicionalmente, se definen los parámetros cinemáticos del controlador, estableciendo una velocidad lineal máxima de $0.5\ m/s$ y limitando la velocidad angular al rango comprendido entre $[-1.0, 1.0]\ rad/s$.

```
nx = 3; % Estados: x, y, theta
ny = 3; % Salidas
nu = 2; % Variables manipuladas: v, w
Ts = 0.1; % Tiempo de muestreo
nlobj = nlmpc(nx, ny, nu);
nlobj.Ts = Ts;

% Modelo interno cinemático del robot (unicycle)
nlobj.Model.StateFcn = @(x, u) [x(1) + Ts * u(1) * cos(x(3)); ...
                                x(2) + Ts * u(1) * sin(x(3)); ...
                                x(3) + Ts * u(2)];

nlobj.Model.OutputFcn = @(x, u) x;
nlobj.Model.IsContinuousTime = false;

nlobj.PredictionHorizon = 10; % Evaluar 10 instantes futuros
nlobj.ManipulatedVariables(1).Min = 0; % Vel. Lineal Mínima
nlobj.ManipulatedVariables(1).Max = 0.5; % Vel. Lineal Máxima
nlobj.ManipulatedVariables(2).Min = -1.0; % Vel. Angular Mínima
nlobj.ManipulatedVariables(2).Max = 1.0; % Vel. Angular Máxima
```

Listing 5.1: Configuración MPC 1.

Una vez configurado el controlador, se establece el lazo de control en tiempo real (*fase online*). En cada iteración del bucle, el sistema se suscribe a los tópicos de odometría del robot mediante ROS para extraer su pose global actual. Dado que la orientación en ROS se recibe en formato de cuaterniones, el algoritmo realiza en primer lugar una conversión a ángulos de Euler para obtener la coordenada direccional θ (Yaw).

Luego, para el seguimiento de la ruta, el controlador evalúa continuamente la distancia euclidiana entre la posición estimada del vehículo y el objetivo actual (*waypoint*). Para considerar un objetivo alcanzado se establece una tolerancia espacial de 0.10 metros. Es decir, en el momento en que el robot se encuentre dentro de este radio, el algoritmo da por alcanzado el objetivo y actualiza el índice hacia el siguiente *waypoint* de la trayectoria. Esto garantiza que el movimiento sea fluido y continuo sin necesidad de que el robot se detenga por completo en cada coordenada.

```
currentWaypointIdx = 2; % Empieza parado en el WP=1, debe apuntar al WP=2
tolerance = 0.10;
msg_odom = sub_odom.LatestMessage; % Obtenemos la odometria
.
.
% Obtenemos la orientación en cuaterniones
odomQuat = [msg_odom.pose.pose.orientation.w, ...
            msg_odom.pose.pose.orientation.x, ...
            msg_odom.pose.pose.orientation.y, ...
```

```
                    msg_odom.pose.pose.orientation.z];
6   odomRotation = quat2eul(odomQuat); % Convertimos los cuaterniones a ángulos de Euler
7   estimatedPose = [msg_odom.pose.pose.position.x, ...
                         msg_odom.pose.pose.position.y, ...
                         odomRotation(1)];

    % Avanzar al siguiente waypoint si el robot ya está cerca
8   while currentWaypointIdx <= size(waypoints, 1)
9       distToWaypoint = norm(waypoints(currentWaypointIdx, 1:2) - estimatedPose(1:2));
10      if distToWaypoint < tolerance
11          currentWaypointIdx = currentWaypointIdx + 1;
12      else
13          break;
14      end
15  end
```

Listing 5.2: Configuración MPC 2.

Por último, el núcleo de la implementación de este algoritmo reside en la adaptación del espacio de trabajo. Si el MPC evaluara directamente las coordenadas globales del mapa, el incremento de los valores absolutos en x e y aumentaría de forma significativa la carga computacional, dejando de ser viable en tiempo real.

Para solucionar este problema, se ha implementado una transformación de coordenadas globales a locales. En cada instante, el sistema no calcula cómo ir desde su posición global hasta el objetivo global, sino que aplica una matriz de transformación trigonométrica para realizar una traslación y rotación del entorno, creando así un mapa local relativo al propio vehículo:

$$\begin{bmatrix} x_{p_local} \\ y_{p_local} \end{bmatrix} = \begin{bmatrix} \cos(\Delta\theta) & \sin(\Delta\theta) \\ -\sin(\Delta\theta) & \cos(\Delta\theta) \end{bmatrix} \begin{bmatrix} \Delta x \\ \Delta y \end{bmatrix} \tag{5.1}$$

donde Δx y Δy son los desplazamientos relativos desde la posición actual (x, y) hasta el objetivo $(x_{p_global}, y_{p_global})$, y $\Delta\theta$ representa la orientación actual del robot respecto al eje x_{global} (ángulo *yaw*). Esta matriz de rotación compensa dicho ángulo, trasladando las coordenadas del mundo a la perspectiva local del vehículo. Una vez aplicada esta transformación, las variables resultantes $(x_{p_local}, y_{p_local})$ pasan a representar, de forma directa, el error longitudinal y el error lateral del robot, respectivamente (Figura 5.1). De esta forma, el controlador asume que el robot se encuentra siempre en el origen de coordenadas locales (0,0) y evalúa la posición del objetivo con respecto a sí mismo.

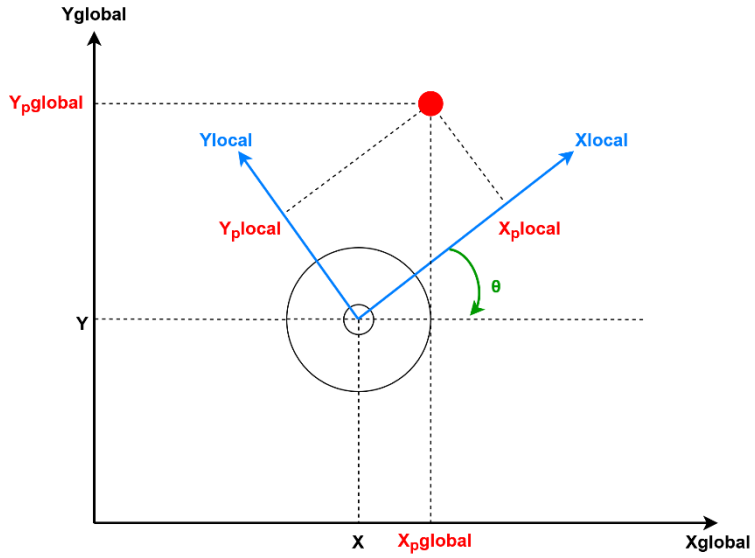


Figura 5.1: Representación geométrica de la transformación de coordenadas globales a locales.

Para entender mejor este proceso, se propone un ejemplo sencillo: supongamos que el robot se encuentra en la coordenada global (2,2) del mapa, y está alineado con el eje x_{global}. Si el siguiente *waypoint* a alcanzar se ubica en la coordenada global (3,3), el cálculo de desplazamientos iniciales es $\Delta x = 1$ metro y $\Delta y = 1$ metro. Como el robot ya se encuentra alineado con el eje x_{global} (su orientación es $\Delta\theta = 0°$), el seno de 0° es 0 y el coseno de 0° es 1, lo que convierte la matriz de la ecuación (5.1) en una matriz de identidad. Al aplicar esta operación, las coordenadas locales resultantes coinciden con los incrementos: (1, 1).

Al deducir la combinación de velocidad lineal y angular a partir de este sistema de referencia relativo, el algoritmo resuelve el problema de navegación sin necesidad de procesar las grandes magnitudes de las coordenadas globales del mapa. Esto permite que la función de optimización `nlmpcmove` de MATLAB pueda converger y generar los comandos de control de manera mucho más rápida y eficiente.

```
targetWaypoint_global = waypoints(currentWaypointIdx, :);
% Distancias absolutas en X e Y entre el robot y el waypoint objetivo
dx = targetWaypoint_global(1) - estimatedPose(1);
dy = targetWaypoint_global(2) - estimatedPose(2);
theta_robot = estimatedPose(3);
% Transformación Trigonométrica al Mapa Local (Rotación)
local_x = dx * cos(theta_robot) + dy * sin(theta_robot);
local_y = -dx * sin(theta_robot) + dy * cos(theta_robot);
local_theta = angdiff(theta_robot, targetWaypoint_global(3));
%El robot siempre cree que está en el origen (0,0) mirando al frente (0 rad)
estado_actual_local = [0, 0, 0];
objetivo_local = [local_x, local_y, local_theta];
% Calcular acción óptima usando predicción futura sobre el mapa local
uMPC = nlmpcmove(nlobj, estado_actual_local, u0, objetivo_local);
```

Listing 5.3: Configuración MPC 3.

5.2 Implementación LQR

En cuanto a la implementación del controlador LQR, el primer paso consiste en configurar todas las matrices del sistema, cuyo fundamento teórico se explicó en el apartado 2.3. Dado que el modelo cinemático de tracción diferencial del robot es no lineal, es indispensable calcular su versión linealizada para que LQR pueda operar. Esta linealización se materializa exactamente en las matrices jacobianas A y B, que pasan a ser la representación matemática del sistema en tiempo discreto (con un tiempo de muestreo $T_s = 0.1\ s$). Debido a que la linealización depende de la orientación del vehículo, ambas matrices no son fijas, sino que deben actualizarse en cada instante T_s en función del ángulo θ actual del robot.

A continuación, se definen las matrices de pesos Q y R, que son las que marcan el comportamiento del vehículo. A la matriz Q, que penaliza las desviaciones del estado, se le han asignado unos valores de [20, 20, 5], indicando que la prioridad principal es minimizar el error posicional en los ejes X e Y, siendo un poco más permisivos con el error en la orientación. Por otro lado, la matriz R, que penaliza el esfuerzo de control, se le han asignado valores bajos de [0.5, 0.5], permitiendo que el sistema utilice sin problemas la potencia de los motores con tal de seguir la línea correctamente. Es decir, priorizamos la corrección ágil de la trayectoria por encima de la suavidad o esfuerzo que tengan que hacer los motores.

```
    % Tiempo de muestreo
1   Ts = 0.1;

    % Matrices del sistema linealizado
2   A = @(theta) [1 0 -Ts*sin(theta); 0 1 Ts*cos(theta); 0 0 1];
3   B = @(theta) [Ts*cos(theta) 0; Ts*sin(theta) 0; 0 Ts];

    % Pesos del LQR
4   Q = diag([20, 20, 5]); % Importancia del error en [X, Y, Theta]
5   R = diag([0.5, 0.5]);  % Penalización por usar [Velocidad, Volante]

    % Límites físicos
6   v_max = 0.5;
7   omega_max = 1.0;
8   tolerance_v_min = 0.05;
```

Listing 5.4: Configuración LQR 1.

Una vez configuradas las matrices, el bucle principal se encarga de calcular los comandos de velocidad $u(t)$ en cada iteración. Recordando la fórmula clave de la ley de control LQR (2.8), $u(t) = -K \cdot x(t)$, el algoritmo debe resolver primero el vector de error del estado actual $x(t)$ y a posteriori la matriz de ganancias K.

Por un lado, para obtener el error del estado actual del robot $[x, y, \theta]$, el algoritmo compara su posición actual con el *waypoint* objetivo, acotando el ángulo resultante en el intervalo

$[-\pi, \pi]$ lo que garantiza que el vehículo determine siempre el sentido de giro más corto hacia la referencia y evitando comportamientos inestables.

Por otro lado, para calcular la matriz de ganancias K, se hace uso de la función `dlqr` de MATLAB, utilizando la variante discreta debido al tiempo de muestreo T_s al que opera el nodo de control. A esta función se le pasan las matrices A, B, Q y R actualizadas en ese instante.

Con la matriz y el error resueltos, se calcula la acción de control $u(t)$ definitiva. Sin embargo, en lugar de utilizar un regulador tradicional que lleve las velocidades a cero cuando no hay error, se le suma una velocidad de crucero base (*feedforward*) de $0.5\ m/s$. De esta forma, el algoritmo asume que el estado natural del robot es avanzar por el circuito, y la multiplicación $-K \cdot x(t)$ se encarga únicamente de añadir o sustraer velocidad y giro cuando se detecta un desvío de la ruta ideal.

Finalmente, antes de publicar los comandos en ROS, se aplican filtros de saturación. Mediante funciones de máximo y mínimo, la velocidad lineal se acota entre $0.5\ m/s$ y una tolerancia mínima de $0.05\ m/s$, evitando que el robot se detenga por completo en las curvas. Del mismo modo, la velocidad angular se limita al rango de $[-1.0, 1.0]\ rad/s$, protegiendo los motores y asegurando que las maniobras sean seguras.

```
    % Calculo del error en X e Y
1   x_ref = waypoints(currentWaypointIdx, :)';
2   x_error = estimatedPose' - x_ref;
    % Calculo del error de orientación (theta) acotando el intervalo en [-π, π]
3   x_error(3) = atan2(sin(x_error(3)), cos(x_error(3)));

    % Calculo de las matrices A y B actuales con la orientación actual
4   A_current = A(estimatedPose(3));
5   B_current = B(estimatedPose(3));

    % Utilizamos dlqr al ser un sistema discreto (Ts = 0.1)
    % Le pasamos las 4 matrices
6   K = dlqr(A_current, B_current, Q, R);

    % uLQR = -K * x_error
    % Inyectamos una velocidad base (feedforward) de 0.5 m/s
7   v_crucero = 0.5;
8   uLQR = [v_crucero; 0] - K * x_error;

    % Filtros de saturación
9   v_cmd = max(min(uLQR(1), v_max), tolerance_v_min);
10  w_cmd = max(min(uLQR(2), omega_max), -omega_max);

    % Enviar velocidades al robot
11  msg_vel.linear.x = v_cmd;
12  msg_vel.angular.z = w_cmd;
13  send(pub_vel, msg_vel);
```

Listing 5.5: Configuración LQR 2.

5.3 Implementación Pure Pursuit

A diferencia de las técnicas predictivas basadas en modelo, como MPC, el algoritmo *Pure Pursuit* es un método de seguimiento totalmente geométrico, lo que reduce la carga computacional del sistema. Como ya se ha mencionado anteriormente, su funcionamiento consiste en calcular la curvatura necesaria para que el vehículo se desplace desde su posición actual hasta un punto objetivo situado sobre la trayectoria de referencia, a una distancia de visión denominada *Lookahead Distance*.

En primer lugar, se instancia el controlador utilizando las librerías nativas de navegación (*Navigation Toolbox*) de MATLAB. Luego, se carga la trayectoria interpolada por *Splines* ya calculada, que al ser una ruta más densa y continua garantiza movimientos más suaves para el controlador.

En segundo lugar, se deben establecer adecuadamente sus parámetros de diseño. Tanto la velocidad lineal deseada como la velocidad angular máxima se han configurado utilizando exactamente los mismos valores empleados en las implementaciones de los controladores MPC y LQR, fijándose en 0.5 m/s y 1.0 rad/s respectivamente. Ahora bien, el núcleo de este controlador y su parámetro más determinante es la distancia de visión (*Lookahead Distance*). Un valor reducido provoca que el sistema persiga puntos demasiado cercanos, generando oscilaciones en la dirección. Por el contrario, un valor excesivamente alto suaviza la conducción a costa de un error de seguimiento mayor al "recortar" el interior de las curvas. Tras realizar múltiples pruebas iterando este parámetro, se ha determinado que un valor de $L_d = 0.5$ metros ofrece el mejor compromiso entre velocidad, suavidad de giro y precisión en el seguimiento de la trayectoria. Los detalles de este comportamiento y su comparativa serán expuestos en el apartado de Resultados.

En último lugar, una vez configurados estos parámetros, el lazo de control opera de manera similar a la fase *online* descrita en apartados anteriores, capturando la pose del robot y publicando los comandos de velocidad lineal y angular resultantes a través de los nodos de comunicación.

```
    % Configuracion del controlador Pure Pursuit
1   controller = controllerPurePursuit;
2   controller.Waypoints = ruta_spline;
3   controller.DesiredLinearVelocity = 0.5; % Velocidad de avance (m/s)
4   controller.MaxAngularVelocity    = 1.0; % Giro máximo (rad/s)
5   controller.LookaheadDistance     = 0.5; % Distancia de visión (m)
```

Listing 5.6: Configuración Pure Pursuit.

5.4 Implementación DDPG

El presente apartado detalla la implementación del algoritmo de Aprendizaje por Refuerzo DDPG. El desarrollo metodológico se estructura en tres grandes fases: la formulación del entorno virtual (que abarca la dinámica de transición o *step*, la función de recompensa y la lógica de episodios); el diseño de las redes neuronales que conforman la arquitectura Actor-Crítico del agente; y, finalmente, el análisis de las métricas de entrenamiento culminando con el despliegue del agente sobre el sistema de control del vehículo.

5.4.1 Creación del entorno

El diseño del entorno de simulación se encuentra encapsulado en la clase *RobotWaypointEnv*, cuya función principal es actuar como el nexo matemático e interactivo entre el agente DDPG y la cinemática del vehículo. En su fase de inicialización, el código define las propiedades físicas y restricciones operativas del sistema. Entre ellas se establecen los límites máximos permitidos de las velocidades lineales ($v_{max} = 0.5\,\frac{m}{s}$) y angulares ($\omega_{min} = -1.0\,\frac{rad}{s}, \omega_{max} = 1.0\,\frac{rad}{s}$), el periodo de muestreo computacional ($T_s = 0.1\ s$), las características geométricas del circuito y el error lateral máximo permitido ($0.23\ m$).

Por otro lado, atendiendo a la naturaleza libre de modelo (*model-free*) del algoritmo DDPG, el agente opera como una 'caja negra' que no requiere procesar ecuaciones cinemáticas. Para elegir las acciones adecuadas desconociendo las leyes físicas, el algoritmo se basa en un proceso empírico de prueba y error guiado por la optimización de recompensas. A base de iterar, la red neuronal logra mapear internamente los estados observados con las velocidades óptimas. Por tanto, el modelo matemático de tracción diferencial (explicado en el apartado 3.4) reside únicamente en el entorno. Proporcionar esta base física es lo que garantiza que la política inferida por el agente sea transferible y válida para la dinámica del *TurtleBot 4* real.

Además, como consideración adicional en el diseño del entorno, cabe mencionar que el agente DDPG ha sido entrenado utilizando únicamente la trayectoria central del circuito como referencia, omitiendo intencionadamente la "*racing line*" precalculada. El objetivo de esta decisión es comprobar si el algoritmo logra aprender por sí mismo a recortar las curvas y aprovechar los márgenes de la pista para encontrar la trazada más eficiente, demostrando con ello el verdadero potencial del RL.

Una vez definidas las características del entorno y el modelo del vehículo, el siguiente paso consiste en establecer cómo percibe el mundo el agente (espacio de observaciones) y cómo puede interactuar con él (espacio de acciones). Por un lado, se define un vector de 24 observaciones compuesto por el estado actual y un *lookahead*. El estado local inmediato se divide en cuatro variables (x_{local} = desviación longitudinal, y_{local} = error lateral, θ_{local} = error de orientación, v_{prev} = velocidad previa), y el bloque de *lookahead* calcula la distancia (d_i) y el ángulo

relativo (α_i) hacia los próximos 10 *waypoints* (20 valores). Como la trayectoria que usamos para entrenar es la interpolada previamente por *Splines*, la cual tiene una separación espacial de 0.25 metros entre *waypoints*, este horizonte otorga al robot una visión anticipada de 2.5 metros físicos reales. Esta magnitud de anticipación resulta clave para dotar al agente de un comportamiento anticipativo. Al observar las curvas mucho antes de adentrarse en ellas, el controlador DDPG adquiere la capacidad espacial necesaria para alejarse de la trayectoria central estricta, permitiéndole inferir de forma autónoma la trazada más directa (*racing line*) y minimizar la curvatura del recorrido.

Para evaluar estos puntos futuros, la elección de coordenadas polares (distancia y ángulo) optimiza la convergencia de la red neuronal al proporcionarle una representación directa de la curvatura del circuito. De este modo, si los ángulos de los próximos *waypoints* se mantienen cercanos a cero, el agente detecta una recta y maximiza la aceleración. Por el contrario, si los ángulos divergen rápidamente, sabe que se anticipa una curva. Esta capacidad predictiva elimina los comportamientos "miopes" o reactivos, permitiendo a la política de control trazar las curvas con suavidad. Dicha división del espacio de observaciones se puede apreciar detalladamente en la Figura 5.2 y Figura 5.3.

Finalmente, tras haber estructurado la forma en la que el agente percibe el entorno, se define el espacio de acción, compuesto por la velocidad lineal (v) y velocidad angular (ω) deseadas que se enviarán a los motores del robot.

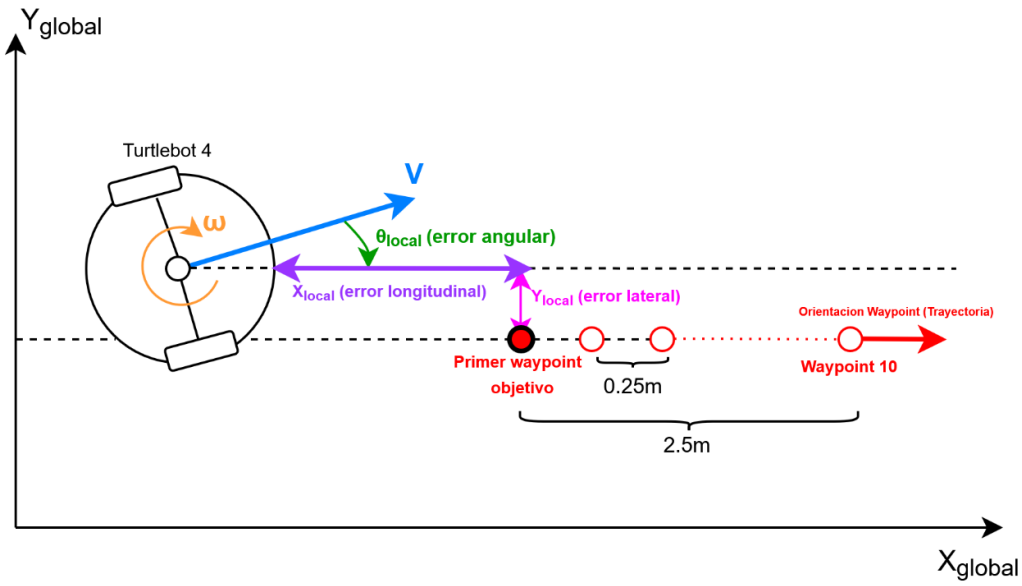


Figura 5.2: Representación geométrica de las observaciones del estado local inmediato.

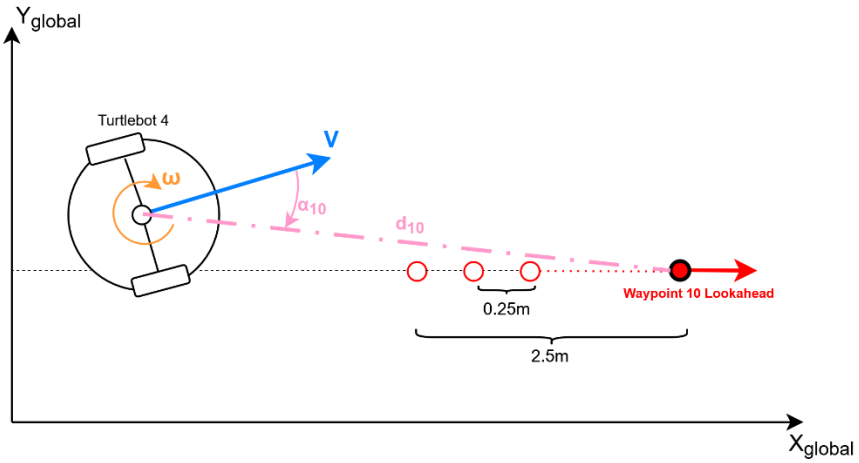


Figura 5.3: Representación del vector de anticipación (lookahead).

Para obtener los valores del vector del estado inmediato, primero se comienza calculando la diferencia cartesiana global entre la posición actual del robot y la del *waypoint* objetivo más próximo (Δx y Δy). Sin embargo, para garantizar que el agente aprenda a navegar de forma relativa, estas distancias se transforman al sistema de referencia local del vehículo. Para ello, se aplica exactamente la misma matriz de rotación trigonométrica que ya fue detallada y justificada en la ecuación (5.1) de la implementación del controlador MPC. Al proyectar estas distancias sobre los ejes del propio robot, se obtienen de forma directa el error longitudinal (x_{local}) y el error lateral (y_{local}). A su vez, el error de orientación (θ_{local}) se determina calculando la diferencia entre el ángulo de la trayectoria y el ángulo actual del robot (Figura 5.2). También es importante destacar que el error angular se normaliza al intervalo $[-\pi, \pi]$ para evitar discontinuidades matemáticas durante el entrenamiento. De este modo, se impide que una simple reorientación física (como pasar de 359° a 1°) se interprete como un error grande de $-358°$ en lugar del giro real y más corto de 2°, garantizando que la red neuronal aprenda a ejecutar siempre el movimiento más directo.

Una vez resuelto el estado inmediato, el algoritmo procede a calcular el vector de *lookahead*, iterando sobre los siguientes 10 *waypoints* de la trayectoria de referencia. Para cada punto futuro i se calculan, de la misma forma que antes, las diferencias espaciales respecto a la posición del robot ($\Delta x_i, \Delta y_i$). A continuación, estas coordenadas cartesianas intermedias se transforman en los parámetros polares mostrados en la Figura 5.3: la distancia al objetivo (d_i) y el ángulo de visión relativo (α_i). La distancia se obtiene calculando la métrica euclídea entre el robot y el *waypoint* futuro. El ángulo, por su parte, se halla aplicando la función arcotangente de cuatro cuadrantes (`atan2`) sobre los incrementos espaciales, restándole después la orientación actual del robot y normalizando al rango ya descrito $[-\pi, \pi]$.

Finalmente, la concatenación secuencial del estado local (x_{local}, y_{local}, θ_{local}, v_{prev}) junto con las diez parejas de coordenadas polares (d_i, α_i) da como resultado el vector unidimensional final que se inyecta como entrada a las redes neuronales del agente DDPG.

5.4.2 Step

El núcleo operativo de cualquier entorno de RL es la función *step*. Su objetivo principal es materializar el ciclo básico de interacción del agente con el entorno ($S \rightarrow A \rightarrow R \rightarrow S'$). En cada iteración, esta función recibe la acción elegida por el agente (los comandos de velocidad), simula la respuesta física del vehículo durante un periodo de muestreo, y devuelve el nuevo estado del sistema (S') y la recompensa obtenida (R). En otras palabras, el entorno opera de forma completamente independiente y es ajeno al diseño interno del agente DDPG; su única responsabilidad es actuar como el motor físico que procesa la acción y actualiza el estado de la simulación, sin importar cómo la red neuronal toma sus decisiones.

Por otro lado, dado que el entorno comparte la misma base matemática que el MPC diseñado previamente, el algoritmo replica sus mismos pasos de control físico: se limitan las velocidades comandadas y se calcula la nueva pose local del robot. Sin embargo, para la actualización del objetivo, a diferencia del MPC donde la transición hacia el siguiente *waypoint* se basa únicamente por la tolerancia de distancia (0.10 m), aquí se incorpora una validación extra de proyección espacial. Si el *waypoint* ha quedado a la espalda del robot ($x_{local} < 0$) y a una distancia de seguridad razonable, el sistema lo descarta y pasa al siguiente. Esta lógica evita que el agente intente realizar giros bruscos para alcanzar un punto que ya ha superado y se ha quedado atrás.

Una vez que la nueva posición física y el objetivo han sido calculados, la función *step* pasa a ejecutar la lógica del algoritmo de aprendizaje a través de las siguientes etapas:

- **Comprobación de Meta.** El sistema verifica de forma continua si el índice del *waypoint* actual ha superado el límite del sector de entrenamiento asignado (*WaypointEnd*). Si la condición se cumple, se activa un booleano indicando que el agente ha cruzado la línea de meta.
- **Generación de la Nueva Observación (S').** Utilizando la nueva posición del robot, se invoca a la función de cálculo de observaciones. Esto recalcula los valores del vector de estado explicado anteriormente (desviación lateral, distancias al próximo *waypoint*, *lookahead*, etc.), generando la "fotografía" actualizada del entorno que el agente necesitará para su próxima decisión.
- **Cálculo de la Recompensa (R).** Toda la telemetría del instante actual (velocidades, errores de seguimiento, etc.) se transfiere a la función de recompensa, la cual será explicada más adelante. Esta devuelve una señal escalar que cuantifica la calidad de la acción ejecutada, constituyendo la principal fuente de retroalimentación del sistema, tal como se explicó en la Figura 2.7.
- **Condición de Terminación.** El entorno debe poder saber cuándo el episodio actual ha llegado a su fin. Para ello se crea una bandera de terminación (*booleano*) que se activa bajo tres circunstancias:

1. *Éxito:* El robot completa satisfactoriamente el circuito o sector designado.
2. *Accidente:* El error lateral absoluto supera el límite de seguridad establecido (0.23 metros), lo que implica una salida de pista.
3. *Timeout:* El contador interno alcanza el número máximo de pasos permitidos por episodio, lo que previene simulaciones infinitas o comportamientos cíclicos donde el robot no avanza.

- **Actualización de Memoria:** Finalmente, la acción ejecutada se almacena en una variable interna de estado previo. Este registro es importante ya que permite a la función de recompensa penalizar al robot si este ha ejecutado una acción brusca o con movimientos oscilatorios.

```
function [Observation, Reward, IsDone, LoggedSignals] = step(this, Action)
    % 1. Saturación de acciones a los límites físicos del sistema
    v = max(this.V_min, min(this.V_max, Action(1)));
    w = max(this.W_min, min(this.W_max, Action(2)));

    % 2. Actualización del modelo cinemático (unicycle)
    x     = this.RobotPose(1);
    y     = this.RobotPose(2);
    theta = this.RobotPose(3);

    x_new     = x     + this.Ts * v * cos(theta);
    y_new     = y     + this.Ts * v * sin(theta);
    theta_new = wrapToPi(theta + this.Ts * w);

    this.RobotPose = [x_new, y_new, theta_new];
    this.StepCount = this.StepCount + 1;

    % 3. Actualización de seguimiento de waypoints
    wpPrev = this.CurrentWaypointIdx;
    while this.CurrentWaypointIdx <= this.NumWaypoints
        wp_curr = this.Waypoints(this.CurrentWaypointIdx, 1:2);
        d = norm(wp_curr - [x_new, y_new]);

        % Proyección longitudinal para comprobar si el waypoint quedó atrás
        dx_wp = wp_curr(1) - x_new;
        dy_wp = wp_curr(2) - y_new;
        local_x = dx_wp * cos(theta_new) + dy_wp * sin(theta_new);

        % Avanzar si se alcanza la tolerancia o si ha superado el waypoint
        if d < this.WaypointTolerance || (local_x < 0 && d < 0.35)
            this.CurrentWaypointIdx = this.CurrentWaypointIdx + 1;
        else
            break;
        end
```

```
    end
    waypointReached = (this.CurrentWaypointIdx > wpPrev);

    % 4. Comprobación de finalización de sector o circuito
    wpFinal = this.WaypointEnd;
    if wpFinal == 0
        wpFinal = this.NumWaypoints;
    end
    circuitCompleted = (this.CurrentWaypointIdx > wpFinal);

    % 5. Generación del nuevo estado y cálculo de recompensa
    Observation = this.computeObservation(v);
    Reward = this.computeReward(v, w, waypointReached, circuitCompleted);

    % 6. Evaluación de condiciones de terminación (IsDone)
    lateralError = abs(Observation(2));
    IsDone = circuitCompleted || ...
             (lateralError > this.MaxLateralError) || ...
             (this.StepCount >= this.MaxEpisodeSteps);

    % 7. Registro de la acción actual para cálculos futuros
    this.PrevAction = [v; w];
end
```

Listing 5.7: Configuración step DDPG.

5.4.3 Creación de la recompensa

La función *computeReward* constituye el núcleo del proceso de aprendizaje, ya que es la encargada de traducir el objetivo conceptual de "conducir correctamente" en señales matemáticas escalares. Para guiar al algoritmo DDPG hacia una política de control óptima y estable, se ha diseñado una función de recompensa híbrida. Esta combina señales densas continuas, que orientan al vehículo paso a paso, con señales dispersas, que marcan eventos críticos del episodio. El diseño matemático de la recompensa se divide en cuatro bloques lógicos, los cuales se ejecutan en cada iteración de la función *step*:

1. Incentivo de progreso continuo.

La base de la recompensa densa busca maximizar el avance hacia el *waypoint* objetivo minimizando simultáneamente los errores de seguimiento:

$$r_{\text{base}} = v \cdot (|\cos(\theta_{\text{local}})| - |\sin(\theta_{\text{local}})| - |y_{\text{local}}|) \tag{5.1}$$

Donde:

- v es la velocidad lineal del robot.
- $|\cos(\theta_{\text{local}})|$ da el valor máximo (1) cuando el robot mira directamente al objetivo.
- $|\sin(\theta_{\text{local}})|$ penaliza el error angular del robot con respecto al objetivo.
- $|y_{\text{local}}|$ penaliza el error lateral cuando no va centrado por la trayectoria

De tal forma que, si el robot está perfectamente alineado con el *waypoint* objetivo (es decir, su orientación coincide con los vectores de referencia verdes mostrados en la Figura 5.4), el primer término del coseno devolverá un 1, mientras que el seno un 0. Si además el error lateral es 0, es decir, el robot va centrado por la trayectoria, el valor total del paréntesis será igual a 1. Este valor luego se multiplica por la velocidad lineal máxima que hayamos elegido. Por ejemplo, si nuestra velocidad lineal máxima (V_{max}) es 0.5 m/s, la recompensa base en cada iteración (*step*) será máxima con un valor de 0.5. Ni bien el robot deje de apuntar al *waypoint* objetivo o se desvíe un poco lateralmente con respecto a la trayectoria, la recompensa será menor a 0.5. Esto además obliga al agente a aprender que cuanto más rápido vaya, más recompensa obtendrá en cada iteración. Si por cualquier motivo decide reducir su velocidad lineal a 0 m/s parándose completamente, da igual si se encuentra perfectamente orientado, ya que la recompensa será nula.

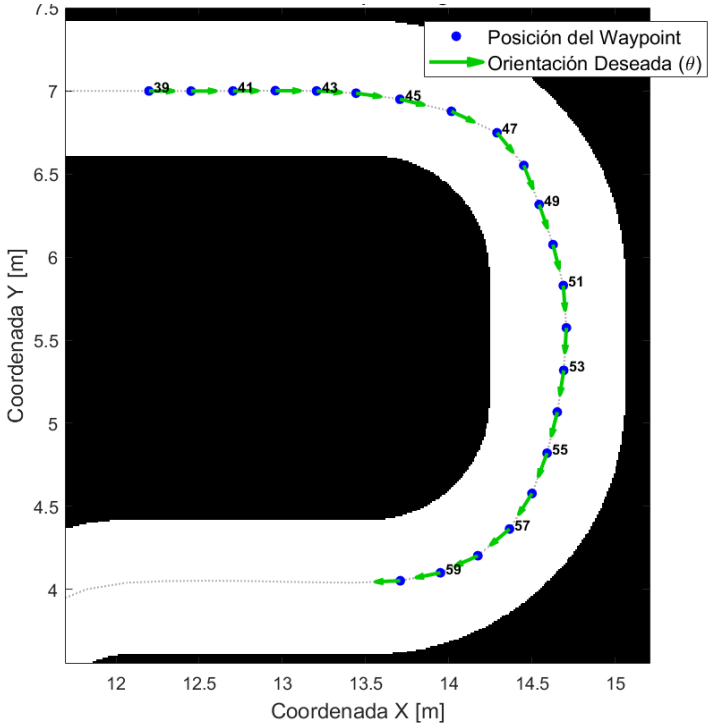


Figura 5.4: Vectores de orientación de referencia que indican la orientación deseada en cada waypoint.

2. Penalización por inactividad.

Durante las fases iniciales de entrenamiento, el agente tendía a descubrir un óptimo local perjudicial. Para evitar recibir grandes penalizaciones, detenía el vehículo por completo antes de las curvas para evitar el riesgo de salirse de la pista. Para eliminar este comportamiento conservador, se introduce una penalización dinámica proporcional a la falta de velocidad:

$$r_{\text{tax}} = -0.10 \cdot \left(1 - \frac{v}{V_{\text{max}}}\right) \tag{5.2}$$

Si el vehículo circula a la velocidad máxima permitida V_{max} el término se anula y no hay penalización. Sin embargo, a medida que la velocidad disminuye, el agente recibe un castigo continuo en cada iteración, forzándolo a resolver las trayectorias de la forma más rápida y eficiente posible.

3. Penalización por oscilaciones.

Otro de los problemas inherentes de las políticas deterministas continuas es la aparición de comportamientos oscilatorios de alta frecuencia en los actuadores. Para garantizar que la conducción sea progresiva y suave, se penalizan los cambios bruscos en el comando de la velocidad angular de un instante $t-1$ al instante actual t:

$$r_{\text{smooth}} = -W_{\text{smooth}} \cdot (w_t - w_{t-1})^2 \tag{5.3}$$

Al elevar la diferencia de giro al cuadrado las pequeñas correcciones del volante generan un castigo prácticamente nulo, mientras que los "volantazos" bruscos producen un castigo exponencial que degrada rápidamente la recompensa total de la iteración.

4. Recompensas dispersas (Eventos terminales).

Finalmente, la señal densa acumulada se complementa con dos grandes recompensas de carácter disperso que definen las condiciones de victoria y derrota, cerrando el episodio de entrenamiento:

- **Salida de pista ($W_{fail} = -20.0$).** Si el error lateral y_{local} supera el límite de seguridad físico de la pista (*MaxLateralError = 0.23m*), se aplica un castigo severo inmediato. Esta señal es fundamental para demarcar los límites de la pista y para que el agente aprenda a no superar dicho limite.

- **Finalización del circuito ($W_{finish} = 200.0$).** Si el booleano *circuitCompleted* se activa al alcanzar el último *waypoint* del sector o circuito, el agente recibe una bonificación masiva. Este valor, desproporcionado respecto a las recompensas intermedias, es el que le indica al agente que ha cumplido el objetivo principal del entrenamiento: completar todo el trazado.

Uniendo todo, la función de recompensa final será igual a:

$$r = v \cdot (|\cos(\theta_{\text{local}})| - |\sin(\theta_{\text{local}})| - |y_{\text{local}}|) + r_{tax} + r_{smooth} \tag{5.4}$$

```
function reward = computeReward(this, v, w, ~, lapCompleted)

    % 1. Extracción de variables del estado local
    obs         = this.computeObservation(v);
    local_y     = obs(2); % error lateral
    local_theta = obs(3); % error angular

    % 2. Incentivo de progreso continuo e impuesto por inactividad
    r_base = v * (abs(cos(local_theta))-abs(sin(local_theta))-abs(local_y));
    r_tax  = 0.10 * (1 - v / this.V_max);
    reward = r_base - r_tax;

    % 3. Penalización por oscilaciones en la actuación
    delta_w  = w - this.PrevAction(2);
    r_smooth = -this.W_smooth * (delta_w^2);
    reward   = reward + r_smooth;

    % 4. Condición terminal negativa: Salida de pista
    if abs(local_y) > this.MaxLateralError
        reward = reward + this.W_fail;
    end

    % 5. Condición terminal positiva: Finalización del sector o circuito
    if lapCompleted
        reward = reward + this.W_finish;
    end
end
```

Listing 5.8: Configuración de recompensa DDPG.

5.4.4 Inicialización de episodios y el olvido catastrófico

La función *resetInternal* es la encargada de restaurar el entorno al comienzo de cada nuevo episodio. Sin embargo, más allá de reiniciar variables, esta función implementa la estrategia central de entrenamiento del agente.

Durante las fases iniciales de desarrollo de este proyecto, se experimentó con un enfoque denominado *Curriculum Learning*. Este método consiste en entrenar al agente de forma progresiva, de tareas más fáciles a más difíciles. En nuestro caso, primero se entrenaba con un sector sencillo como una recta, y una vez terminado ese entrenamiento se pasaba a entrenar un sector más demandante, como una curva a derechas. No obstante, este método generó dos problemas críticos derivados de la memoria temporal de las redes neuronales. En primer lugar, al someter al agente a episodios continuos sobre un mismo tipo de trazado (por ejemplo, curvas a la derecha), este sufría un fuerte sobreajuste (*overfitting*), siendo incapaz de resolver un trazado de geometría opuesta al enfrentarse a él por primera vez. En segundo lugar, si se permitía al agente entrenar en este nuevo sector hasta dominarlo, la actualización de los gradientes reescribía por completo los pesos de la red, provocando el fenómeno conocido como *olvido catastrófico* (*Catastrophic Forgetting*). Es decir, el agente aprendía a girar a la izquierda, pero perdía la capacidad que ya poseía para resolver el problema anterior (girar a la derecha).

Para solucionar esta inestabilidad y forzar a la red neuronal a encontrar una política de control que generalice cualquier sector, se abandonó el *Curriculum* progresivo en favor de una estrategia de *Aleatorización de Tareas* (*Task Randomization*). Mediante este enfoque, se selecciona aleatoriamente el sector del circuito sobre el que se entrena en cada episodio, evitando así que el agente memorice un único tramo. Para llevar a cabo este método se ha seleccionado el circuito 1, dado que es el trazado que mayor variedad de curvas presenta. Este se ha dividido en seis sectores representativos, tal y como se puede observar en la Figura 5.5

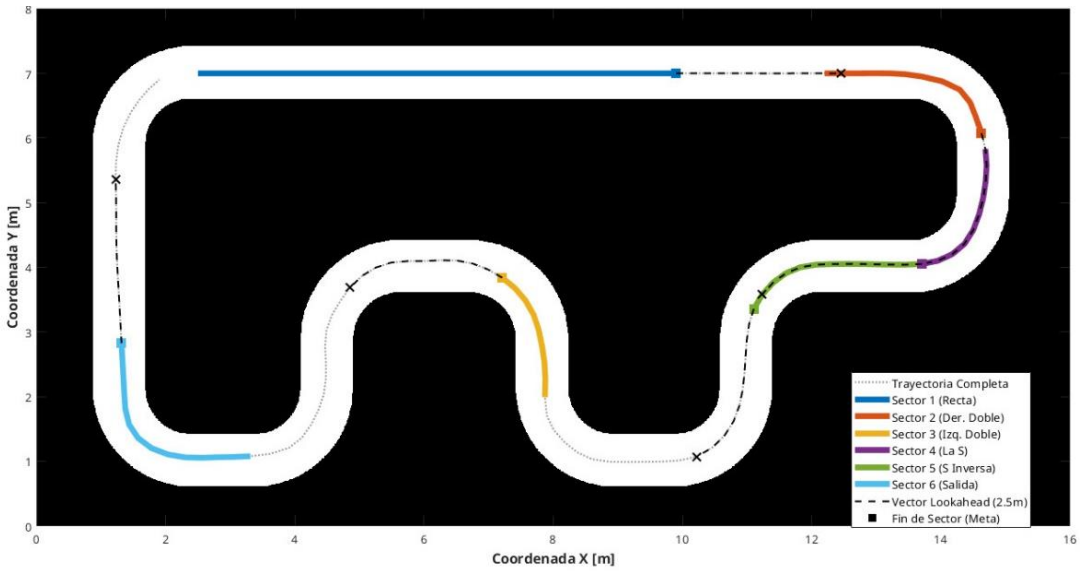


Figura 5.5: Sectores del circuito 1 a entrenar.

La finalidad de dividir la pista en los tramos ilustrados es lograr que el agente sea capaz de resolver cualquier configuración del trazado sin importar el orden en el que aparezca. Como particularidad de la implementación, al estar cada sector definido entre un *waypoint* inicial y otro final, cuando el robot llegue al límite de su tramo de entrenamiento, todavía dispone de la ventana *lookahead* de los próximos 10 puntos. Es decir, seguirá mirando la trayectoria que viene por delante, aunque ya haya terminado de entrenar el sector. Por este motivo, no es lo mismo entrenar una curva a la derecha que le siga una curva a la derecha, que entrenar otra que le siga una curva a la izquierda (formando como una "S"). De esta forma, para cubrir todos los casos geométricos y de anticipación, tenemos los siguientes seis sectores estratégicos:

- **Sector 1 (Recta).** Un tramo recto puro donde tanto la posición actual del vehículo como su horizonte de visión apuntan hacia adelante sin variaciones angulares.
- **Sector 2 (Derecha Doble).** Una curva sostenida a la derecha, donde el *lookahead* anticipa que la exigencia de giro continuará en la misma dirección.
- **Sector 3 (Izquierda Doble).** Una curva sostenida a la izquierda, obligando al agente a aprender a mantener la tasa de giro cerrada hacia el interior de ese lado.
- **Sector 4 (La "S").** Una transición compleja donde el robot se encuentra trazando una curva a la derecha, pero su visión predictiva ya está detectando el inicio de un giro hacia la izquierda.
- **Sector 5 (La "S" Inversa).** El escenario geométricamente opuesto al anterior, donde el vehículo traza a la izquierda mientras anticipa la entrada a una curva a la derecha.
- **Sector 6 (Salida de Curva a Recta).** El tramo final de un vértice donde el vehículo sigue girando, pero el *lookahead* ya detecta la estabilización del trazado hacia una recta, enseñando al agente a abrir la dirección de forma progresiva y suave.

Para entrenar con dichos sectores, la función *resetInternal* implementa esta estrategia a través de los siguientes mecanismos lógicos:

- **Alternancia rápida de tareas.** Se ha implementado un límite estricto de retención por sector. Es decir, el código fuerza un cambio de escenario de forma obligatoria tras un máximo de tres episodios, lo que evita que los gradientes de la red neuronal se acomoden en un único tipo de geometría (como pasaba con *Curriculum Learning*).
- **Selección aleatoria uniforme.** El algoritmo genera un valor probabilístico aleatorio entre 0 y 1, dividiendo el espacio de decisión en seis segmentos idénticos (1/6 de probabilidad para cada sector). De este modo, si el valor aleatorio es menor a 1/6, se entrena el sector 1; si está entre 1/6 y 2/6 se entrena el sector 2, y así sucesivamente. Esto garantiza que, a lo largo del entrenamiento, el agente se enfrente estadísticamente la misma cantidad de veces a todos los escenarios.

- **Ajuste dinámico del horizonte del episodio.** Al instanciar un nuevo sector, el entorno calcula la cantidad de *waypoints* que lo componen (por ejemplo, 15) y ajusta automáticamente la variable *MaxEpisodeSteps*. De este modo, se asigna un tiempo límite proporcional a la longitud del sector, evitando que el agente consuma recursos extra tras haber fallado la trazada.
- **Inyección de ruido.** Para evitar que el robot inicie cada episodio con una alineación perfecta, se introduce una ligera dispersión gaussiana (*PoseNoise*) en su orientación inicial, fomentando un aprendizaje más robusto.

```
    % 1. Comprobación de cambio de sector (límite de 3 episodios)
1   necesita_cambio = false;
2   if this.ActiveSector == 0 || this.EpisodesInSector >= 3
3      necesita_cambio = true;
4   end
    % 2. Selección aleatoria uniforme del nuevo sector de entrenamiento
5   if necesita_cambio
6      r = rand();
7         if     r < 1/6, this.ActiveSector = 1;     % Recta Pura
8         elseif r < 2/6, this.ActiveSector = 2;     % Derecha Continua
9         elseif r < 3/6, this.ActiveSector = 3;     % Izquierda Continua
10        elseif r < 4/6, this.ActiveSector = 4;     % Derecha -> Izquierda (S)
11        elseif r < 5/6, this.ActiveSector = 5;     % Izquierda -> Derecha
12        else,           this.ActiveSector = 6; end % Curva -> Recta
13     this.EpisodesInSector = 0; end
14  this.EpisodesInSector = this.EpisodesInSector + 1; % Contador de episodios

    % 3. Asignación de Waypoints según el sector activo
15  switch this.ActiveSector
16     case 1, this.WaypointStart = 1;   this.WaypointEnd = 30;
17     case 2, this.WaypointStart = 39;  this.WaypointEnd = 50;
18     case 3, this.WaypointStart = 92;  this.WaypointEnd = 100;
19     case 4, this.WaypointStart = 51;  this.WaypointEnd = 60;
20     case 5, this.WaypointStart = 60;  this.WaypointEnd = 71;
21     case 6, this.WaypointStart = 123; this.WaypointEnd = 136; end
22
    % 4. Ajuste dinámico de los pasos máximos del episodio
23  numWpSector = this.WaypointEnd - this.WaypointStart;
24  this.MaxEpisodeSteps = max(300, numWpSector * 20);

25  noise = this.PoseNoise * 0.3 * randn(1, 3); % 5. Ruido
26  this.RobotPose = this.Waypoints(this.WaypointStart, :) + noise;
27  this.RobotPose(3) = wrapToPi(this.RobotPose(3));

    % 6. Actualización del objetivo inmediato
28  this.CurrentWaypointIdx = this.WaypointStart + 1;
```

Listing 5.9: Configuración del entorno de entrenamiento DDPG.

5.4.5 Creación y arquitectura del agente DDPG

La implementación del agente DDPG requiere el diseño de dos redes neuronales profundas que interactúan de forma continua: el Actor (cuya función es deducir la mejor acción posible para mejorar su política de navegación) y el Crítico (encargado de juzgar dicha decisión estimando el valor esperado de la recompensa), ambas explicadas anteriormente en el marco teórico. Estas arquitecturas han sido modeladas mediante perceptrones multicapa (MLP) utilizando la librería *Deep Learning Toolbox* de MATLAB.

Por un lado, tenemos la red del Actor, cuyo propósito principal es observar el estado del entorno y emitir una acción física continua (los comandos de velocidad lineal y angular). Su red está formada por los siguientes elementos:

- **Capa de entrada.** Recibe el vector de observaciones compuesto por 24 elementos (4 variables cinemáticas del estado local y 20 valores correspondientes al horizonte predictivo del *lookahead*).
- **Capas ocultas.** Se utiliza una estructura en forma de embudo o cuello de botella para la extracción de características abstractas, compuesta por tres capas conectadas (*Fully Connected*) de 128, 128 y 64 neuronas, respectivamente.
- **Capa de salida y escalado físico.** La última capa de la red emite dos valores: el comando de aceleración (v) y el de giro (ω). Para evitar que intente exigir velocidades imposibles que saturarían al robot, la salida pasa por dos filtros. Por un lado, pasa por una limitación (*tanhLayer*) que fuerza a que los números salgan siempre en un rango cerrado entre -1 y 1. Esto se puede entender como un porcentaje que va desde el -100% (marcha atrás o giro tope a izquierda) hasta el +100% (máxima aceleración o giro tope a derecha). Por otro lado, pasa por una conversión al mundo real (*scalingLayer*) que traduce este "porcentaje" a las unidades físicas reales que los motores del robot entienden (de V_{min} a V_{max} para avanzar, y de ω_{min} a ω_{max} para girar). Esto asegura que la política sea determinista y físicamente viable.

Por otro lado, tenemos la red del Crítico, cuyo propósito es predecir la recompensa futura acumulada dado un estado y una acción concreta. Al tratarse de una arquitectura orientada a la evaluación simultánea de ambas dimensiones, se ha diseñado una red con dos ramas convergentes:

- **Rama de observación.** Procesa el vector de 24 elementos del entorno a través de dos capas ocultas (128 neuronas cada una).
- **Rama de acción.** Procesa la acción continua de 2 elementos (v y ω) recién generada por el Actor mediante una capa oculta de 128 neuronas.
- **Fusión y salida.** Suma la información abstracta de ambas ramas (*additionLayer*), y la pasa por una última capa de 64 neuronas que proporciona un único valor escalar a su

salida: el valor Q estimado, es decir, la predicción matemática de toda la recompensa futura que el robot espera obtener si sigue esa trayectoria.

Para que el comportamiento de ambas redes pueda converger, se deben configurar cuidadosamente todos sus **hiperparámetros**. Las decisiones de diseño tomadas para este entorno abordan directamente las vulnerabilidades teóricas del RL expuestas en secciones anteriores.

1. **Tasas de aprendizaje asimétricas (Actor vs. Crítico).** Para garantizar la estabilidad del entrenamiento, se ha definido a qué velocidad se actualizan los pesos de las redes. La tasa de aprendizaje del Crítico se ha fijado en $3 \cdot 10^{-4}$, mientras que la del Actor es un poco inferior (10^{-4}). Esta asimetría se debe a que el Crítico debe modelar las dinámicas físicas del circuito y comprender la función de recompensa lo más rápido posible, para poder guiar y decirle al Actor si la acción que está eligiendo es óptima. Si el Actor evolucionase a la misma velocidad o superior, optimizaría su política basándose en las evaluaciones incorrectas del Crítico, provocando el colapso del entrenamiento.

2. **Factor de descuento del horizonte de visión (γ).** Este parámetro (*DiscountFactor*) se ha configurado con un valor de 0.99. Tal y como se demostró en la ecuación del valor esperado (2.9) del marco teórico, un valor de γ cercano a 1 hace que el agente priorice las recompensas futuras sobre las inmediatas. Esto es especialmente relevante en este trabajo, ya que la función de recompensa incluye una señal de gran magnitud al cruzar la línea de meta ($W_{finish} = 200.0$), que solo se obtiene al final del episodio. Con un γ alto, ese premio futuro mantiene su peso matemático al propagarse hacia atrás estado por estado, lo que permite al agente aprender que las decisiones tomadas al inicio del circuito influyen directamente en la recompensa final.

3. **Actualización suave de redes objetivo (τ).** Para estabilizar los objetivos de la ecuación de Bellman y evitar oscilaciones en la convergencia, DDPG utiliza redes *target*. Como se detalló previamente, estas redes están condicionadas por un valor τ, el cual tiene que ser cercano a 0. Para ello, se ha definido $\tau = 0.001$, lo que implica que las redes *target* solo asimilan un 0.1% de los nuevos pesos en cada iteración. Esto hace que se actualicen lentamente y proporcionen un objetivo estable para el agente.

4. **Memoria de experiencia (*Replay Buffer*)**. Tal y como se explicó al abordar los métodos *off-policy*, el agente requiere de un registro histórico para su entrenamiento. Para ello, se ha implementado un búfer con capacidad para 10^6 transiciones, donde cada transición representa una "experiencia" completa del vehículo ($S \rightarrow A \rightarrow R \rightarrow S'$). En cada iteración, en lugar de aprender únicamente del último movimiento realizado, el

algoritmo selecciona aleatoriamente un lote (*mini-batch*) de 256 muestras extraídas de esta memoria. Al aprender de un conjunto desordenado de experiencias pasadas en lugar de pasos consecutivos, se rompe la fuerte correlación temporal de la conducción y se evita el olvido catastrófico (*catastrophic forgetting*), impidiendo que la red neuronal sobreajuste sus pesos a las trayectorias más recientes y pierda la capacidad de resolver escenarios previamente aprendidos.

5. **Ruido para la exploración continua.** Por último, puesto que la política del Actor es determinista, la capacidad de descubrir nuevas trazadas se inyecta de forma externa añadiendo ruido al vector de acción. Durante las primeras fases de desconocimiento del entorno, este ruido es alto para fomentar una exploración activa. Sin embargo, a medida que avanza el entrenamiento, el nivel de ruido se va reduciendo automáticamente gracias al factor de decaimiento (*StandardDeviationDecayRate* = 10^{-6}). Esto permite que el agente pase de dar giros agresivos al inicio, a afinar gradualmente su conducción, manteniendo únicamente un ruido mínimo residual de 0.01 en etapas avanzadas.

5.4.6 Resultados del entrenamiento y convergencia del agente

La validación empírica de la arquitectura diseñada se observa en la curva de aprendizaje generada durante la fase de entrenamiento (Figura 5.6).

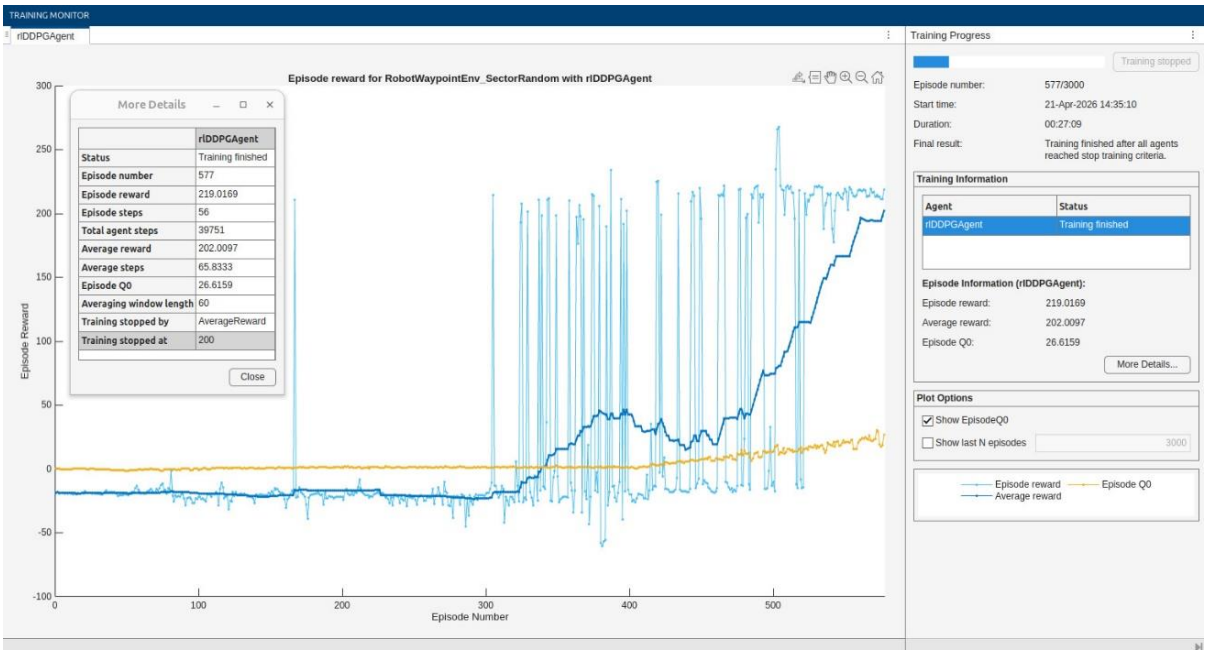


Figura 5.6: Entrenamiento exitoso del agente DDPG.

En primer lugar, antes de analizar la convergencia del entrenamiento, es preciso recordar las dos unidades de medida fundamentales que conforman el avance temporal en el entorno de

simulación. Por un lado, tenemos las iteraciones o *steps*, que como fueron explicadas anteriormente, representan el ciclo básico y discreto de interacción en RL. Dado que se ha definido un tiempo de muestreo T_s de 0.1 segundos, cada iteración equivale exactamente a una décima de segundo en el mundo físico. Por otro lado, tenemos los episodios, que constituyen una secuencia continua de *steps* desde que el robot se inicializa en el circuito hasta que termina por cualquier condición de terminación. Si observamos las estadísticas globales de convergencia de la Figura (5.6), el agente consolidado requiere una media de 65.8 *steps* para completar exitosamente un episodio de un sector. Multiplicado por el periodo de muestreo, esto indica que el robot es capaz de trazar y superar los sectores en una media 6.5 segundos, demostrando una navegación ágil.

En segundo lugar, aunque el entrenamiento se ha configurado con un límite máximo permisivo de 2000 episodios, el algoritmo DDPG ha demostrado una alta eficiencia de muestreo, alcanzando el criterio de parada de forma prematura y exitosa en el **episodio 577**. Dicho criterio, que nos indica cuando se alcanza el éxito, se ha diseñado de la siguiente forma: la simulación debe detenerse automáticamente si la media móvil de las recompensas (calculada sobre una ventana de 60 episodios) supera los 200 puntos, valor que coincide con la bonificación por alcanzar la línea de meta ($W_{finish} = 200.0$). Es decir, que los últimos 60 episodios del entrenamiento hayan terminado la trayectoria con éxito obteniendo los 200 puntos de la meta. Estadísticamente, esta ventana de validación equivale a asegurar unas 10 pruebas exitosas por cada uno de los 6 sectores que se han definido. A su vez, se ha configurado un guardado condicional para almacenar en disco únicamente aquellos agentes (episodios) puntuales que hayan superado dicha barrera de los 200 puntos, garantizando la recolección de las políticas más óptimas.

En tercer lugar, el comportamiento del agente se puede dividir en tres fases bien diferenciadas (Figura 5.6):

1. **Fase de Exploración (Episodios 0 - 300)**. Durante el primer tercio del entrenamiento, la recompensa por episodio (línea azul clara) se mantiene en valores negativos y oscila cerca del cero. En esta fase, como el ruido inyectado en las acciones es máximo, el agente explora los límites físicos del entorno, sufriendo penalizaciones continuas por salidas de pista o errores de orientación. La media de recompensa (línea azul oscura) también se mantiene plana, ya que el algoritmo está recopilando transiciones fallidas en su memoria de experiencia (*Replay Buffer*) para que el Crítico comience a entender la física del vehículo y del entorno.
2. **Punto de Inflexión (Episodios 300 - 500)**. A partir del episodio 300, se observa una mejora significativa en el rendimiento. Aparecen picos repentinos en la recompensa de episodios que superan cómodamente la barrera de los 200 puntos, indicando que el agente ha logrado cruzar la línea de meta por primera vez. Gracias a que la tasa de decaimiento del ruido comienza a surtir efecto, la política abandona los comportamientos erráticos.
3. **Fase de Convergencia (Episodios 500 - 577)**. En los últimos episodios, la varianza disminuye y la política se estabiliza. La media móvil cruza el criterio de parada objetivo

en el **episodio 577**, deteniendo el proceso con una recompensa media final de **202.01** y un pico de **219.02** en su último intento. Este último episodio 577 contiene el cerebro optimizado de toda la red neuronal, y es el que se ejecutará tanto en el simulador MVSim como en el robot real, lo cual será detallado más adelante.

En último lugar, otro indicador fundamental del éxito de esta arquitectura es la curva amarilla Q_0 que representa la estimación de valor que el Crítico hace al inicio de cada episodio. Como se aprecia en la gráfica, esta curva asciende de forma suave y continua conforme avanza el entrenamiento, sin presentar picos abruptos ni crecimientos descontrolados. Esto es importante ya que demuestra que se ha logrado evitar el problema del sesgo de sobreestimación (un fallo común donde el Crítico predice recompensas irreales repentinas que no se pueden alcanzar, y que engañan al Actor). Por tanto, la configuración asimétrica de los optimizadores ha funcionado a la perfección, ya que el Crítico ha sido capaz de predecir de forma estable y realista que el comportamiento del Actor estaba mejorando progresivamente.

5.4.7 Despliegue del agente

El paso final del desarrollo de DDPG consiste en transferir todo el conocimiento adquirido por el agente durante el entrenamiento y desplegarlo en el simulador MVSim. Para ello se recupera el modelo guardado en el último episodio (*Agent577.mat*), el cual, si bien almacena la arquitectura completa del agente (Actor y Crítico), solo se requiere la red neuronal del Actor para la conducción real. Por lo tanto, el código aísla esta red y la instancia con sus pesos definitivos ya congelados y optimizados. Al desactivar los mecanismos de actualización y el ruido de exploración, el Actor asume un control completamente determinista, limitándose a traducir las observaciones del circuito en acciones de velocidad lineales y angulares inmediatas.

Para que el agente pueda seleccionar dichas acciones, el entorno real debe proporcionarle la información exactamente en el mismo formato matemático con el que fue entrenado. Es decir, en cada instante de muestreo T_s se le brinda el vector completo de observación, que incluye la pose actual del robot y el *lookahead* de los siguientes 10 *waypoints*, ya explicado anteriormente.

Con el vector de observación del estado actual ya construido, se llama a la función encargada de generar la acción del agente (*getAction*). A esta función se le pasa dicho vector del estado actual y devuelve los comandos específicos de velocidad lineal y angular que debe ejecutar el robot en ese momento. Gracias a la arquitectura de la red, estos valores ya se devuelven acotados a los límites físicos seguros de los actuadores del robot (V_{min} a V_{max}, y ω_{min} a ω_{max}).

Por otro lado, a pesar de la precisión matemática de las redes neuronales, las políticas entrenadas por RL tienden a ser "nerviosas", generando en ocasiones oscilaciones instantáneas

y cambios de dirección abruptos de un *step* a otro. Para solucionar este problema, se ha implementado un *Filtro de Media Móvil Exponencial* (EMA) sobre el canal de la velocidad angular, formado por la siguiente ecuación diferencial discreta:

$$w_{\text{filtrado}} = \alpha \cdot w_{\text{nuevo}} + (1 - \alpha) \cdot w_{\text{previo}} \tag{5.6}$$

donde se ha establecido un coeficiente de suavizado $\alpha = 0.3$. Esto implica que, en cada iteración, el simulador obtiene únicamente un 30% del nuevo giro seleccionado por el agente, manteniendo un 70% del giro del ciclo anterior.

Dado que los procedimientos de lectura de odometría global, inicialización de parámetros cinemáticos y carga de *waypoints* son iguales a los implementados en los controladores explicados en secciones anteriores, el código a continuación se centra en la integración del modelo neuronal y la gestión de sus comandos de acción.

```
1  while true
       % ... [Lectura de odometría, guardado de trayectoria real y actualiza-
   ción del waypoint objetivo] ...

       % --- Calcular observación (igual que el entrenamiento) ---
2      obs = computeObsDespliegue(estimatedPose, currentWaypointIdx, ...
                                  waypoints_full, LookaheadSteps, v_prev);

       % --- Generar acción con el agente DDPG ---
3      action = getAction(agent, {obs});
4      action = action{1};

       % Saturar velocidades
5      v = max(V_min, min(V_max, action(1)));
6      w = max(W_min, min(W_max, action(2)));

       % --- Filtro Paso Bajo para la dirección ---
7      alpha_w = 0.3;
8      w_filtrado = (alpha_w * w) + ((1 - alpha_w) * w_prev_filtrado);
9      w_prev_filtrado = w_filtrado; % Guardar memoria para el siguiente paso

       % --- Enviar velocidades al simulador ---
10     msg_vel.linear.x  = v;
11     msg_vel.angular.z = w_filtrado;
12     send(pub_vel, msg_vel);
13 end
```

Listing 5.10: Configuración despliegue agente DDPG.

Capítulo 6

6. Resultados

En este capítulo se exponen los resultados obtenidos por el *TurtleBot 4 Lite* operando bajo los diferentes algoritmos de navegación desarrollados a lo largo del proyecto. En primer lugar, se realiza una evaluación individual de cada controlador en el entorno de simulación, analizando su comportamiento y su capacidad para completar los circuitos diseñados. Posteriormente, se repite esta misma evaluación con el robot físico en el mundo real.

Para analizar más en profundidad los resultados de cada ejecución y comparar el rendimiento de las diferentes estrategias de control, se han registrado las siguientes métricas:

1. Error Lateral Medio (MLE [m]). Mide la desviación lateral media entre la trayectoria de referencia (y_i^{ref}) y la trayectoria real del vehículo (y_i^{real}). Es la métrica principal de precisión de seguimiento:

$$\text{Mean Lateral Error} = \frac{1}{N}\sum_{i=1}^{N}\left|y_i^{\text{real}} - y_i^{\text{ref}}\right| \tag{6.1}$$

2. Raíz del Error Cuadrático Medio (RMSE [m]): Su concepto es prácticamente idéntico al del MLE, con la diferencia de que eleva los errores al cuadrado antes de promediarlos. Al aplicar el cuadrado, el RMSE penaliza significativamente los desvíos más grandes. De esta forma, nos permite detectar si el controlador sufre alejamientos puntuales muy bruscos (peligrosos para la seguridad), incluso si su error medio (MLE) aparenta ser bajo:

$$\text{RMSE} = \sqrt{\frac{1}{N}\sum_{i=1}^{N}\left(y_i^{\text{real}} - y_i^{\text{ref}}\right)^2} \tag{6.2}$$

3. Desviación Típica del Error Lateral (STD [m]): Cuantifica la dispersión del error lateral respecto a su valor medio ($\bar{e}$). Un valor bajo indica que la desviación del robot se mantiene constante en el tiempo, garantizando un seguimiento estable y libre de oscilaciones. Partiendo del error instantáneo ($e_i = y_i^{real} - y_i^{ref}$) definido en las métricas anteriores, su valor se obtiene calculando la raíz cuadrada de la varianza:

$$\text{STD} = \sqrt{\frac{1}{N}\sum_{i=1}^{N}(e_i - \bar{e})^2} \tag{6.3}$$

4. Oscilación de la Dirección RMS (Steering Oscillation RMS [deg/s]): Evalúa de forma directa la suavidad en las órdenes de dirección para detectar si el controlador es inestable. Dado el sistema de tracción diferencial del robot, se calcula a partir de la varianza de la velocidad angular comandada (ω). Una oscilación elevada indica "volantazos" continuos que degradan la conducción:

$$\text{Steering Oscillation RMS} = \sqrt{\frac{1}{N}\sum_{i=1}^{N}(\omega_i - \bar{\omega})^2} \tag{6.4}$$

5. Sacudida Lateral RMS (Lateral Jerk RMS [m/s³]): Constituye la métrica principal para evaluar el confort de la marcha. El *jerk* representa la derivada de la aceleración lateral (a_{lat}) en el tiempo, por lo que los valores altos evidencian variaciones bruscas o cambios agresivos en la aceleración al trazar las curvas:

$$\text{Lateral Jerk RMS} = \sqrt{\frac{1}{N}\sum_{i=1}^{N}\left(\frac{da_{\text{lat},i}}{dt}\right)^2} \tag{6.5}$$

6. Sacudida Longitudinal RMS (Longitudinal Jerk RMS [m/s³]): De forma análoga, se calcula a partir de la derivada de la aceleración longitudinal (a_{lon}). Esta métrica mide la suavidad en el eje de avance, cuantificando la brusquedad de las frenadas y aceleraciones del vehículo.

7. Esfuerzo de Control RMS (Control Effort RMS [-]): Evalúa la eficiencia y agresividad del algoritmo cuantificando la energía total exigida a los actuadores mediante la combinación de la velocidad lineal (v) y la velocidad angular (ω). Valores altos indican maniobras bruscas, mientras que valores bajos garantizan una conducción estable y minimizan el desgaste mecánico de los motores:

$$\text{Control Effort RMS} = \sqrt{\frac{1}{N}\sum_{i=1}^{N}(v_i^2 + \omega_i^2)} \tag{6.6}$$

6.1 Resultados en simulación

6.1.1 Resultados DDPG

En cuanto a los resultados obtenidos tras ejecutar el agente óptimo 577 del entrenamiento DDPG, podemos observar que este ha aprendido a seguir toda la trayectoria de la manera más rápida y precisa. Esto confirma que el entrenamiento realizado en etapas anteriores, con los hiperpárametros correctamente seleccionados, ha funcionado a la perfección. En la Figura 6.1 se puede observar el trazado central a seguir del circuito 1, representado por una línea interlineada azul; mientras que la trazada realizada por el robot controlado por DDPG se corresponde a la línea continua roja. Como se puede observar en la parte superior de la Figura 6.1, este ha tardado 77.30 segundos en dar una vuelta completa.

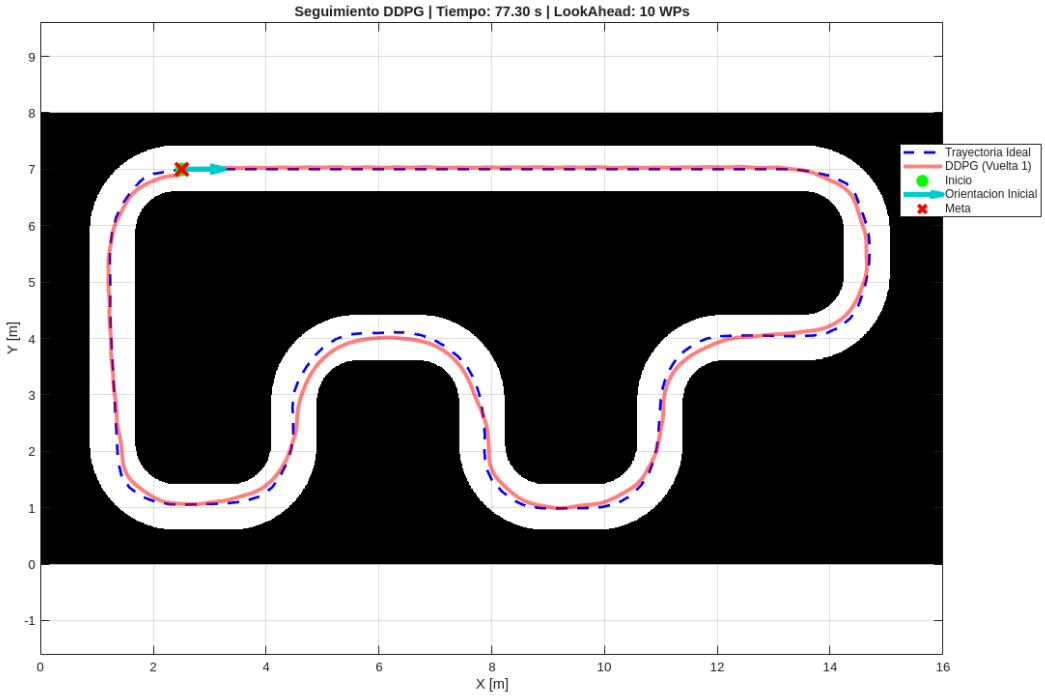


Figura 6.1: Vuelta del Circuito 1 con Agente DDPG.

```
================================================================
Tiempo de vuelta       : 77.30 s (0.498 m/s)
----------------------------------------------------------------
Mean Lateral Error       [m]    : 0.0526 (5.3 cm)
Maximum Lateral Error    [m]    : 0.1504 (15.0 cm)
Error Lateral (RMSE)     [m]    : 0.0637 (6.4 cm)
Error Lateral (STD)      [m]    : 0.0360 (3.6 cm) [Consistencia]
Steering Oscillation RMS [deg/s]: 22.1502
Lateral Jerk RMS         [m/s³] : 0.8602
Longitudinal Jerk RMS    [m/s³] : 3.5236
Control Effort RMS       [-]    : 0.6366
================================================================
```

Figura 6.2: Métricas DDPG circuito 1.

Como se observa en la Figura 6.2, la velocidad media es aproximadamente de 0.5 m/s (valor que coincide con el límite definido en el entrenamiento del DDPG), lo que indica que el robot ha circulado a su velocidad máxima permitida de forma constante. Además, la desviación típica del error lateral con respecto a la trayectoria central es de únicamente 3.6 centímetros, un resultado muy positivo que refleja una gran estabilidad; mientras que el error lateral máximo registrado es de 15 centímetros. Este último dato representa uno de los hitos más relevantes del trabajo: el agente DDPG (entrenado sobre la trayectoria central) ha sido capaz de inferir y descubrir de forma autónoma la trazada más rápida recortando curvas, igualando el límite de seguridad de 15 centímetros establecido matemáticamente para la *racing line*. Esto demuestra que el agente DDPG ha sido capaz de deducir la trayectoria óptima del circuito, coincidiendo exactamente con la ruta que se había establecido a nivel matemático minimizando la curvatura.

Para comprobar la robustez y la capacidad de generalización del agente DDPG, se ha invertido el orden de los *waypoints* del circuito 1 (Figura 6.3) y se ha evaluado su comportamiento durante múltiples vueltas (Figura 6.4). De esta forma, se verifica que el agente no se ha limitado a memorizar una trayectoria específica, sino que es capaz de trazar las curvas en sentido opuesto y mantener la estabilidad del control a largo plazo.

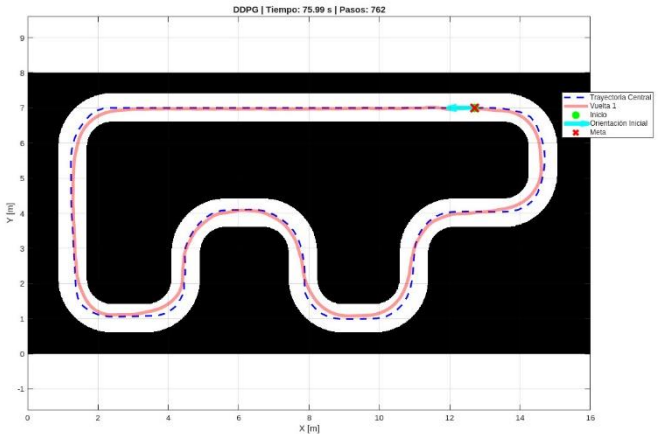


Figura 6.3: Vuelta del Circuito 1 Invertido con Agente DDPG.

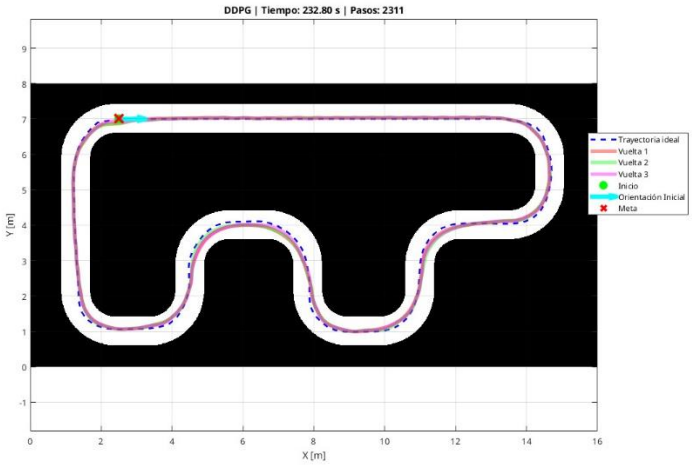


Figura 6.4: 3 Vueltas Seguidas al Circuito 1.

Por otra parte, pese a que el entrenamiento del agente se limitó a los sectores del circuito 1, se ha puesto a prueba su versatilidad desplegándolo en los demás circuitos del simulador para analizar su respuesta frente a estos escenarios desconocidos.

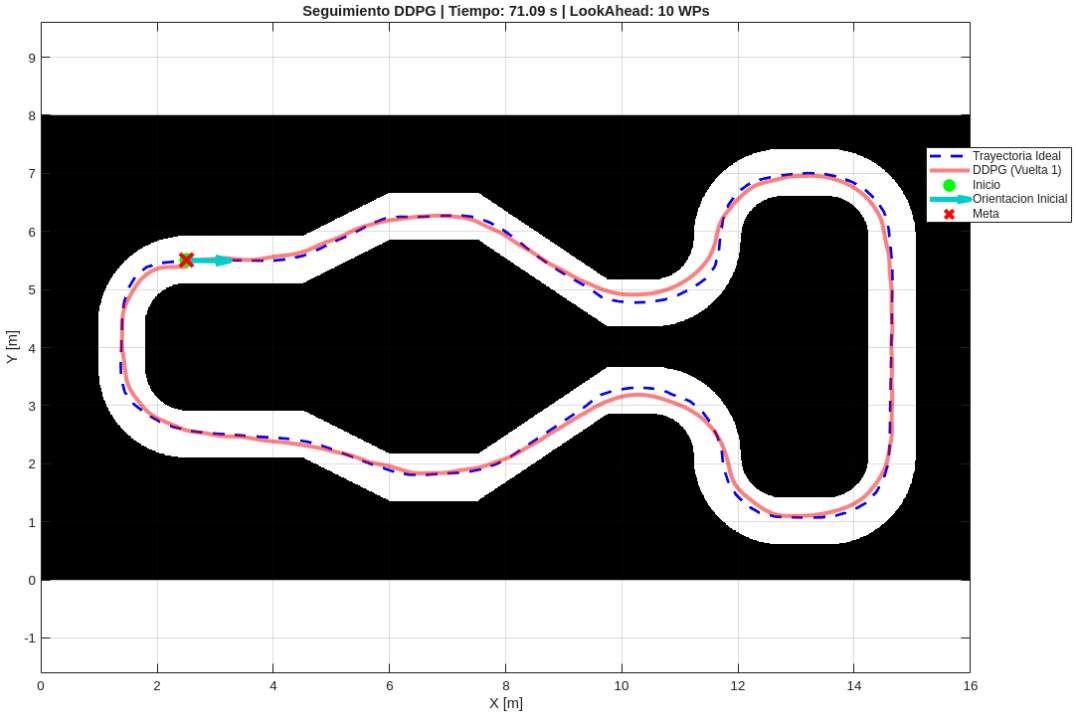


Figura 6.5: Vuelta del Circuito 2 con Agente DDPG.

```
================================================================
Tiempo de vuelta       : 71.09 s (0.504 m/s)
----------------------------------------------------------------
Mean Lateral Error       [m]    : 0.0566 (5.7 cm)
Maximum Lateral Error    [m]    : 0.1854 (18.5 cm)
Error Lateral (RMSE)     [m]    : 0.0695 (7.0 cm)
Error Lateral (STD)      [m]    : 0.0404 (4.0 cm) [Consistencia]
Steering Oscillation RMS [deg/s]: 22.0332
Lateral Jerk RMS         [m/s³] : 0.8562
Longitudinal Jerk RMS    [m/s³] : 3.6736
Control Effort RMS       [-]    : 0.6366
================================================================
```

Figura 6.6: Métricas DDPG circuito 2.

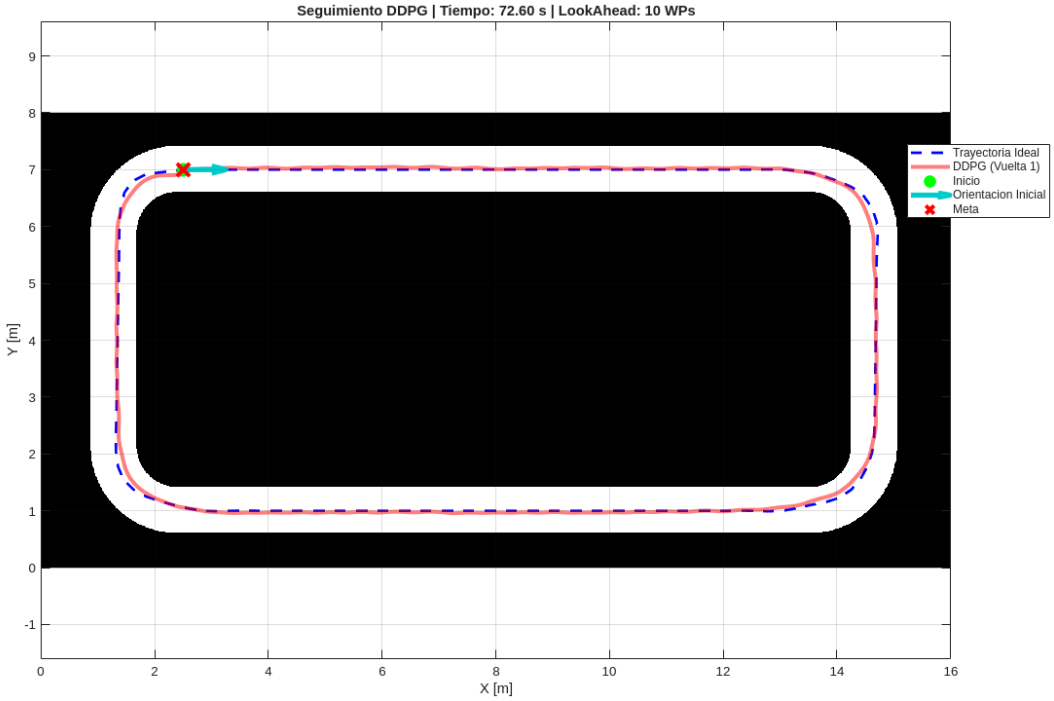


Figura 6.7: Vuelta del Circuito 3 con Agente DDPG.

```
================================================================
Tiempo de vuelta       : 72.60 s (0.499 m/s)
----------------------------------------------------------------
Mean Lateral Error       [m]    : 0.0327 (3.3 cm)
Maximum Lateral Error    [m]    : 0.1219 (12.2 cm)
Error Lateral (RMSE)     [m]    : 0.0405 (4.0 cm)
Error Lateral (STD)      [m]    : 0.0238 (2.4 cm) [Consistencia]
Steering Oscillation RMS [deg/s]: 19.7424
Lateral Jerk RMS         [m/s³] : 0.8907
Longitudinal Jerk RMS    [m/s³] : 0.0002
Control Effort RMS       [-]    : 0.6130
================================================================
```

Figura 6.8: Métricas DDPG Circuito 3.

En ambos circuitos (Figura 6.6 y Figura 6.8) se puede observar que la velocidad media es de 0.5 m/s, indicando que el robot ha ido a su máxima velocidad permitida durante toda la vuelta. Además, se observa que la desviación típica del error lateral del circuito 2 es de 4 centímetros, mientras que en el circuito 3 se obtiene una desviación inferior de 2.4 centímetros debido a la geometría rectangular de la trayectoria. Por último, al analizar el circuito 3, el error lateral máximo registrado se sitúa en torno a los 12 centímetros. Este valor confirma nuevamente el éxito de la política aprendida: a pesar del cambio en el trazado, el agente DDPG replica su estrategia de optimización. De forma autónoma, el vehículo recorta las curvas acercándose al límite propuesto de 15 centímetros, encontrando que el límite para dicho circuito es de 12 centímetros.

6.1.2 Resultados Pure Pursuit

En cuanto a los resultados del algoritmo *Pure Pursuit*, y con el objetivo de poder compararlos directamente con el agente DDPG y los demás controladores, se ha establecido una velocidad lineal deseada de 0.5 m/s y una velocidad angular comprendida entre [-1.0, 1.0] rad/s. Además, tal como se ha explicado en su implementación, se ha definido una distancia de *lookahead* de 0.5 metros, lo que significa que el robot proyecta siempre su objetivo a esa distancia sobre la trayectoria. Se ha comprobado que este valor es el idóneo para seguir la ruta con precisión y evitar oscilaciones. El resultado visual de este seguimiento sobre la trayectoria central del circuito se muestra en la Figura 6.9.

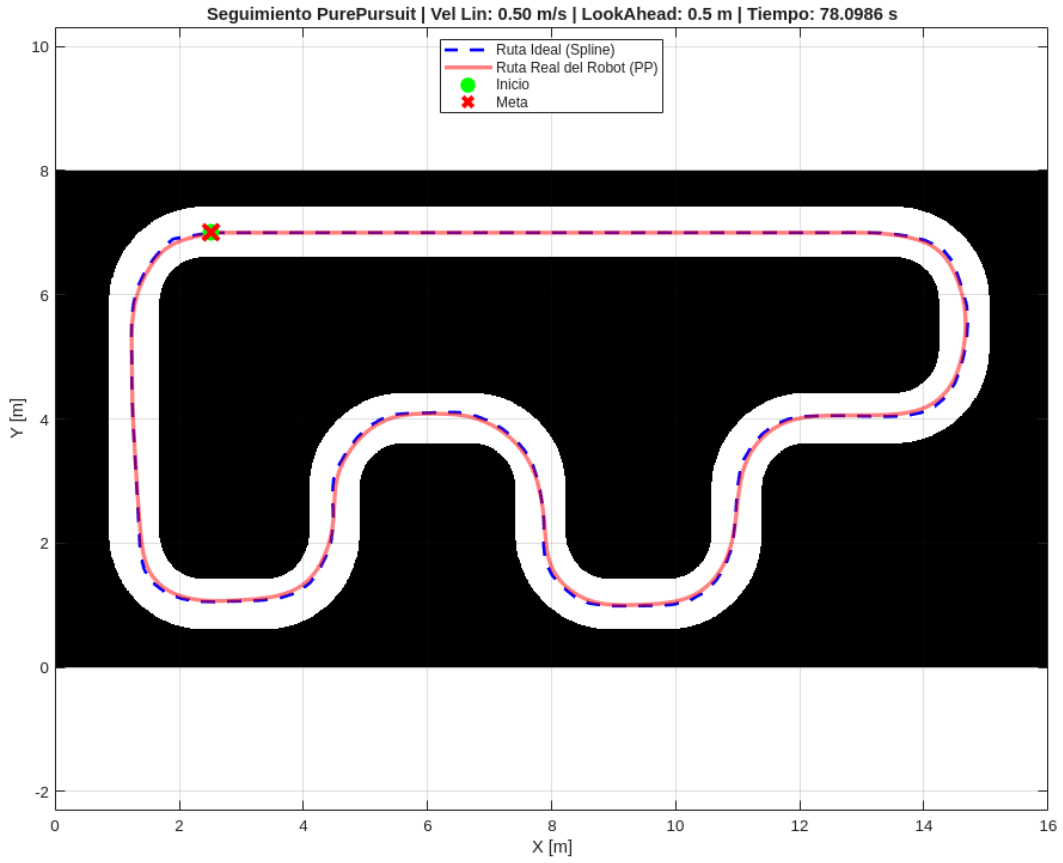


Figura 6.9: Vuelta Trayectoria Central Pure Pursuit Circuito 1.

```
================================================================
Tiempo de vuelta       : 78.10 s (0.499 m/s)
----------------------------------------------------------------
Mean Lateral Error       [m]    : 0.0204 (2.0 cm)
Maximum Lateral Error    [m]    : 0.0726 (7.3 cm)
Error Lateral (RMSE)     [m]    : 0.0267 (2.7 cm)
Error Lateral (STD)      [m]    : 0.0172 (1.7 cm) [Consistencia]
Steering Oscillation RMS [deg/s]: 15.7668
Lateral Jerk RMS         [m/s³] : 0.1538
Longitudinal Jerk RMS    [m/s³] : 0.0000
Control Effort RMS       [-]    : 0.5759
================================================================
```

Figura 6.10: Métricas Pure Pursuit Trayectoria Central Circuito 1.

Atendiendo al informe de métricas (Figura 6.10), *Pure Pursuit* ha completado la vuelta en un tiempo de 78.1 segundos, destacando especialmente su elevada consistencia debido a una desviación típica de apenas 1.7 centímetros, un error lateral medio de 2 centímetros y un error lateral máximo muy contenido de 7.3 centímetros.

Por otro lado, se ha realizado una segunda prueba con este controlador para verificar si la trayectoria óptima presenta mejoras en cuanto a rendimiento con respecto a la trayectoria central. Dichos resultados se pueden observar en la Figura (6.11) y (6.12).

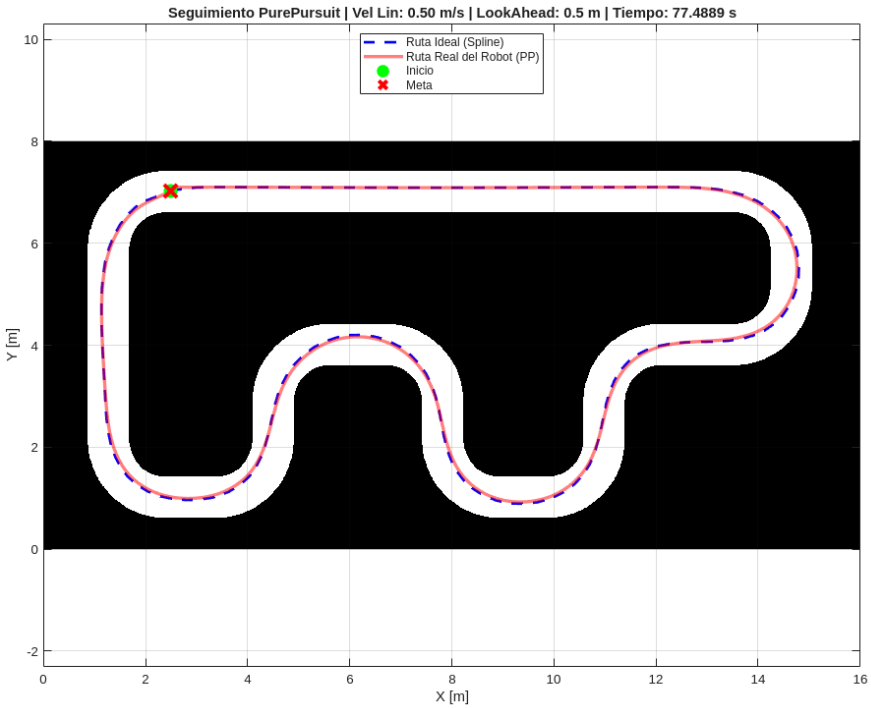


Figura 6.11: Vuelta Trayectoria Óptima Pure Pursuit Circuito 1.

```
================================================================
Tiempo de vuelta       : 77.49 s (0.498 m/s)
----------------------------------------------------------------
Mean Lateral Error       [m]    : 0.0203 (2.0 cm)
Maximum Lateral Error    [m]    : 0.0728 (7.3 cm)
Error Lateral (RMSE)     [m]    : 0.0253 (2.5 cm)
Error Lateral (STD)      [m]    : 0.0150 (1.5 cm) [Consistencia]
Steering Oscillation RMS [deg/s]: 12.9890
Lateral Jerk RMS         [m/s³] : 0.0900
Longitudinal Jerk RMS    [m/s³] : 0.0000
Control Effort RMS       [-]    : 0.5541
================================================================
```

Figura 6.12: Métricas Pure Pursuit Trayectoria Óptima Circuito 1.

Según el nuevo informe de métricas (Figura 6.12), *Pure Pursuit* ha completado la vuelta sobre la trayectoria ideal en un tiempo de 77.49 segundos, lo que supone una mejora de 0.6 segundos. Esto corrobora la lógica del problema: al mantener una velocidad lineal constante, adaptar la trazada para recortar las curvas permite al robot reducir la distancia total recorrida y, en consecuencia, disminuir la duración final de la vuelta.

Por último, también se ha desplegado el control *Pure Pursuit* en los demás circuitos, tanto para seguir la trayectoria central (Figura 6.13 y Figura 6.14) como para seguir la óptima (Figura 6.15 y Figura 6.16). Estos resultados se analizarán más adelante en el apartado de comparación de algoritmos.

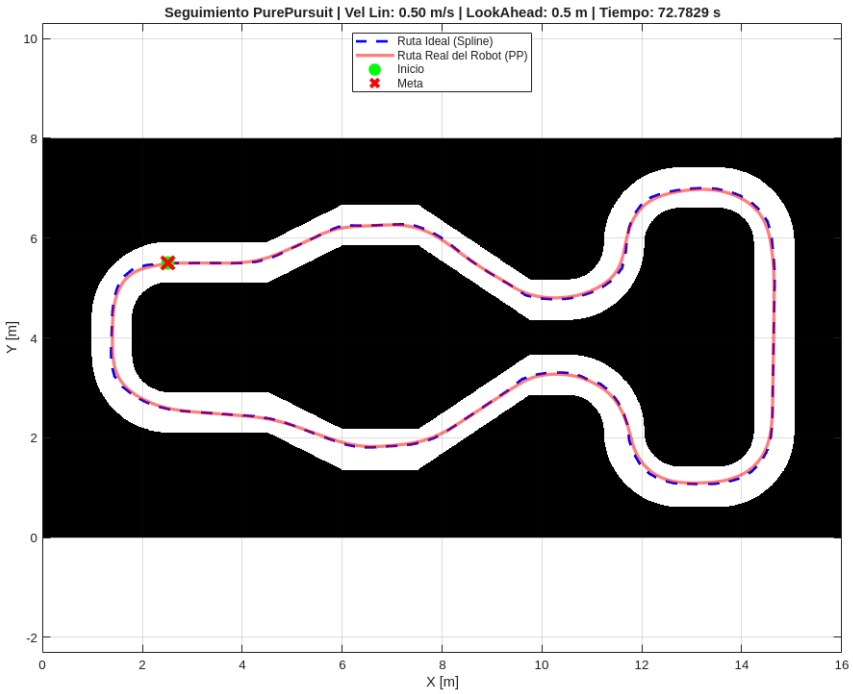


Figura 6.13: Vuelta Trayectoria Central Pure Pursuit Circuito 2.

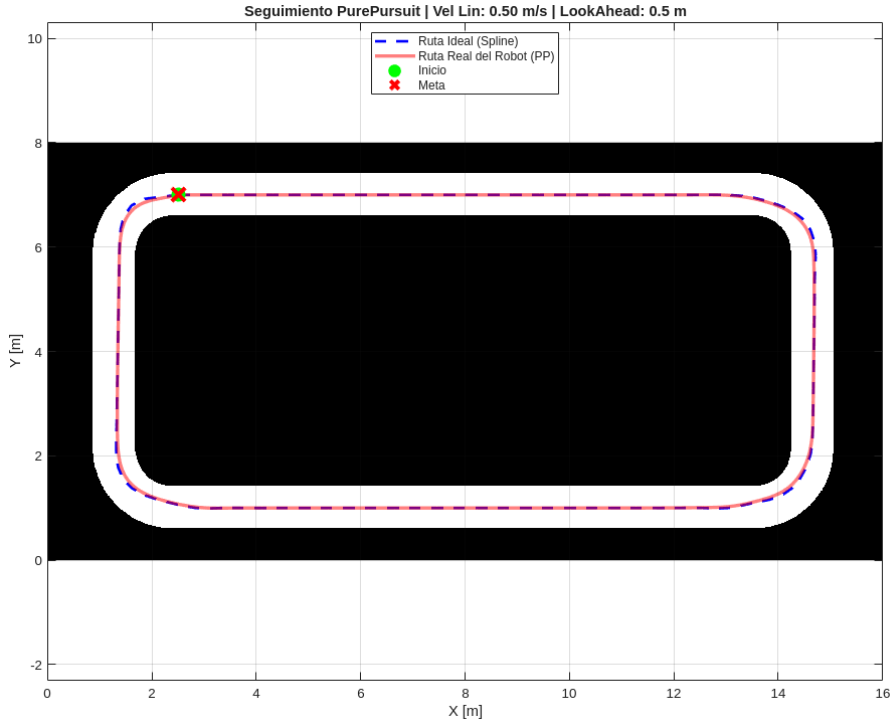


Figura 6.14: Vuelta Trayectoria Central Pure Pursuit Circuito 3.

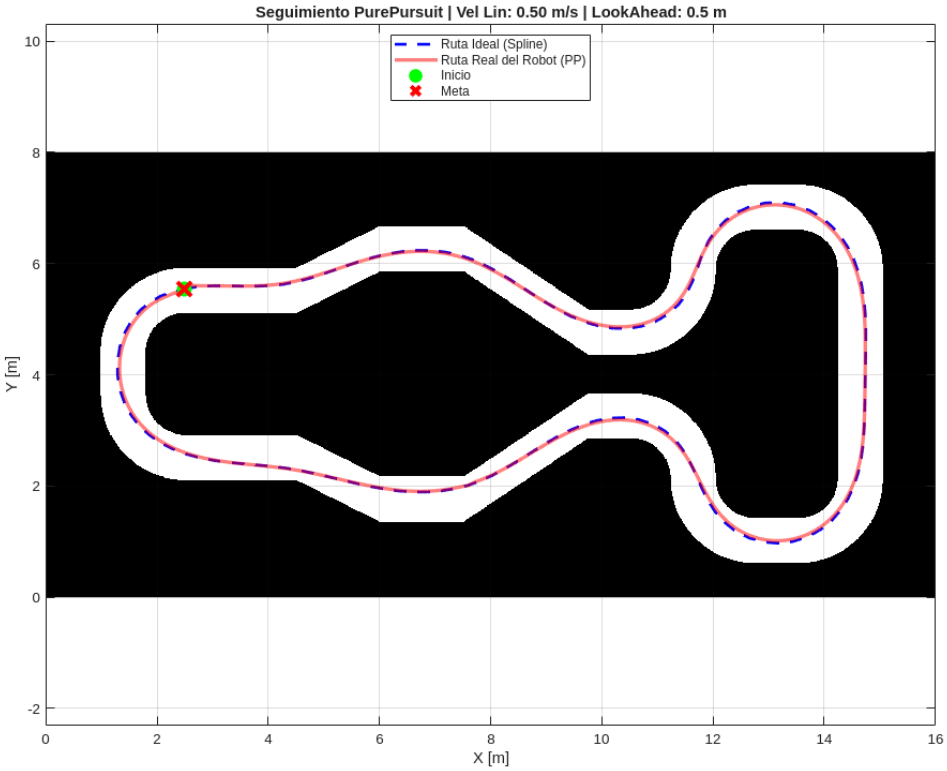


Figura 6.15: Vuelta Trayectoria Óptima Pure Pursuit Circuito 2.



Figura 6.16: Vuelta Trayectoria Óptima Pure Pursuit Circuito 3.

6.1.3 Resultados LQR

Para los resultados de LQR se ha procedido de la misma forma que antes: se define la velocidad lineal máxima a 0.5 m/s y la velocidad angular limitada entre [-1.0, 1.0] rad/s. Los únicos parámetros que se pueden modificar de LQR son los de las matrices, que ya se han explicado y fijado en el apartado de la implementación. Los resultados tras realizar una vuelta por la trayectoria central del circuito 1 con LQR se observan en la Figura 6.17.

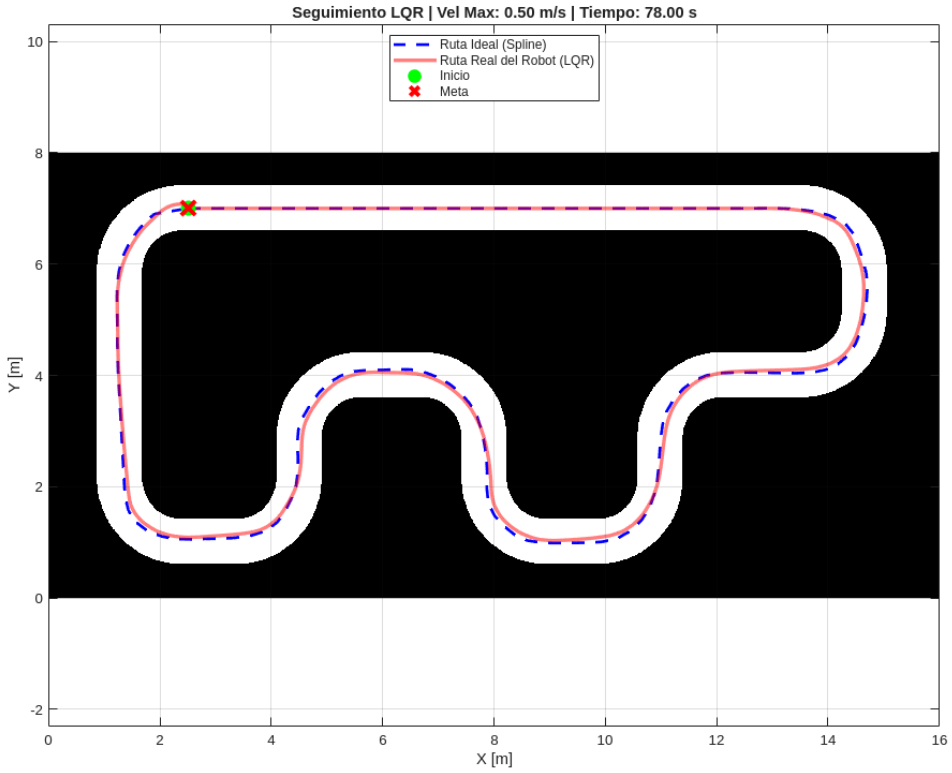


Figura 6.17: Vuelta Trayectoria Central LQR Circuito 1.

```
================================================================
Tiempo de vuelta       : 78.00 s (0.499 m/s)
----------------------------------------------------------------
Mean Lateral Error       [m]    : 0.0350 (3.5 cm)
Maximum Lateral Error    [m]    : 0.1042 (10.4 cm)
Error Lateral (RMSE)     [m]    : 0.0448 (4.5 cm)
Error Lateral (STD)      [m]    : 0.0280 (2.8 cm) [Consistencia]
Steering Oscillation RMS [deg/s]  : 20.9929
Lateral Jerk RMS         [m/s³] : 1.3705
Longitudinal Jerk RMS    [m/s³] : 0.0000
Control Effort RMS       [-]    : 0.6253
================================================================
```

Figura 6.18: Métricas Trayectoria Central LQR Circuito 1.

Según el informe de métricas del LQR (Figura 6.18), los datos extraídos muestran que el robot ha completado el trazado en 78 segundos, tiempo apenas inferior al de *Pure Pursuit*. Además, obtiene una desviación típica mínima de 2.8 centímetros y un error lateral medio de 3.5 centímetros, lo que demuestra que es un controlador muy seguro para la navegación. Sin

embargo, presenta un *lateral jerk* bastante más elevado que los algoritmos anteriores, ya que al no tener un horizonte de predicción actúa de forma más reactiva.

Al igual que se ha hecho con el algoritmo *Pure Pursuit*, se ha llevado a cabo una nueva simulación para evaluar el comportamiento del LQR al seguir la trayectoria óptima (Figura 6.19). Esta comparativa directa frente a la trazada central busca corroborar si el suavizado de la curvatura aporta ventajas operativas adicionales, como una reducción en el tiempo total.

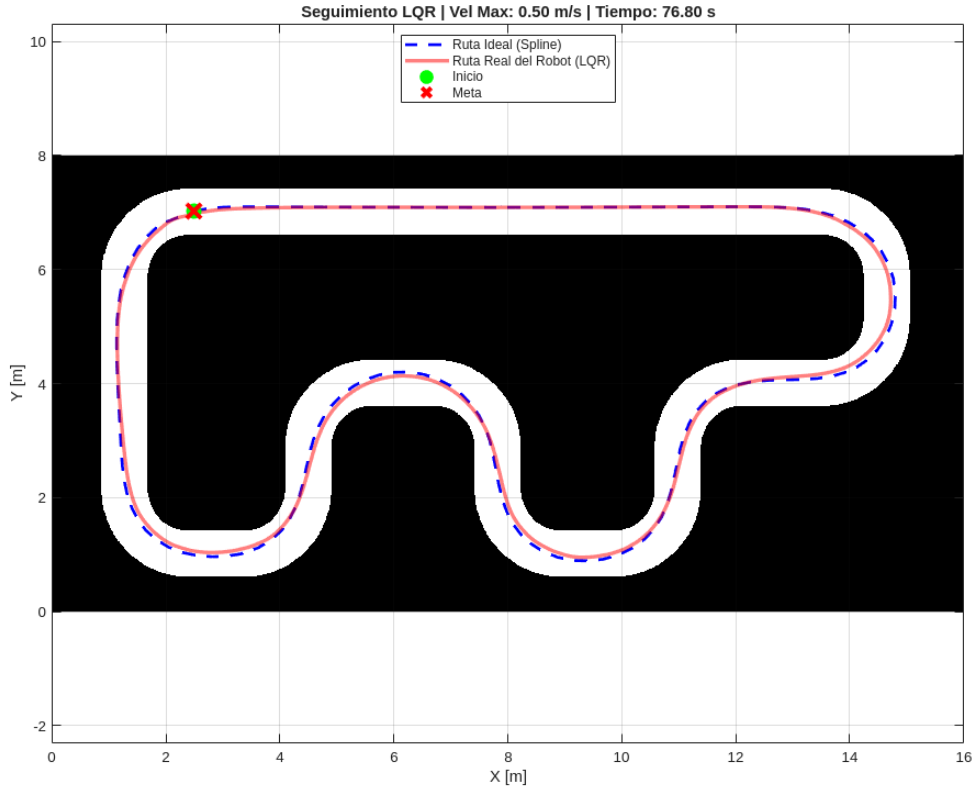


Figura 6.19: Vuelta Trayectoria Óptima LQR Circuito 1.

```
================================================================
Tiempo de vuelta       : 76.80 s (0.500 m/s)
----------------------------------------------------------------
Mean Lateral Error       [m]    : 0.0371 (3.7 cm)
Maximum Lateral Error    [m]    : 0.0892 (8.9 cm)
Error Lateral (RMSE)     [m]    : 0.0459 (4.6 cm)
Error Lateral (STD)      [m]    : 0.0269 (2.7 cm) [Consistencia]
Steering Oscillation RMS [deg/s]  : 18.2009
Lateral Jerk RMS         [m/s³] : 1.3250
Longitudinal Jerk RMS    [m/s³] : 0.0000
Control Effort RMS       [-]    : 0.5969
================================================================
```

Figura 6.20: Métricas Trayectoria Óptima LQR Circuito 1.

Las métricas obtenidas al seguir la trayectoria óptima (Figura 6.20) demuestran un impacto muy positivo en el desempeño del LQR. El tiempo total de la vuelta ha disminuido a 76.8 segundos, reduciendo el tiempo de la trayectoria central en 1.2 segundos, lo que indica nuevamente que la trayectoria óptima supone una mejora significativa con respecto a la central en términos de tiempo por vuelta.

Por último, también se ha desplegado el control LQR en los demás circuitos, tanto para seguir la trayectoria central (Figura 6.21 y Figura 6.22) como para seguir la trayectoria óptima (Figura 6.23 y Figura 6.24).

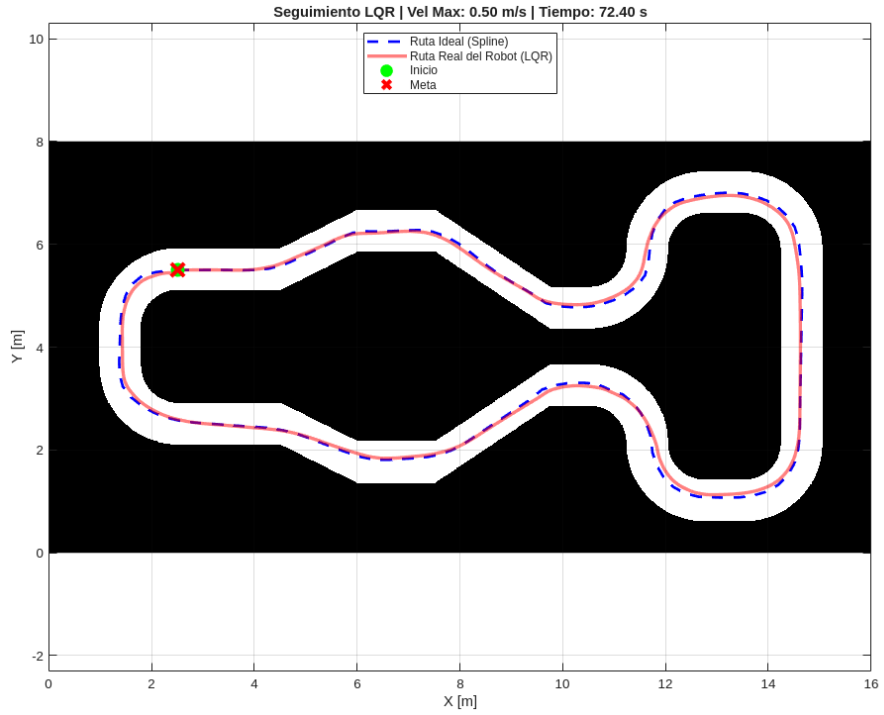


Figura 6.21: Vuelta Trayectoria Central LQR Circuito 2.

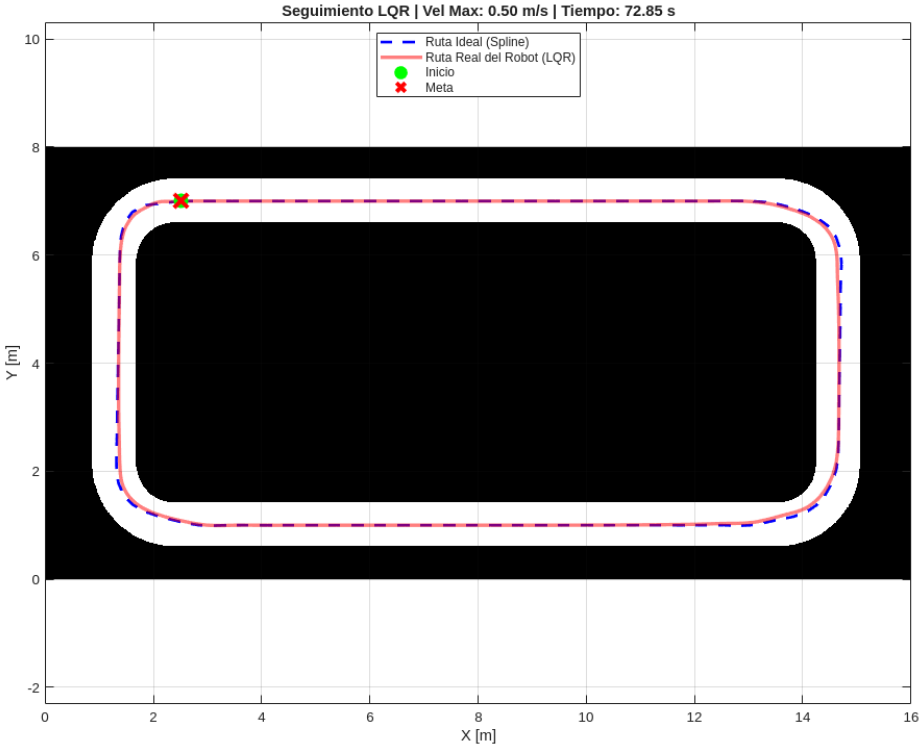


Figura 6.22: Vuelta Trayectoria Central LQR Circuito 3.

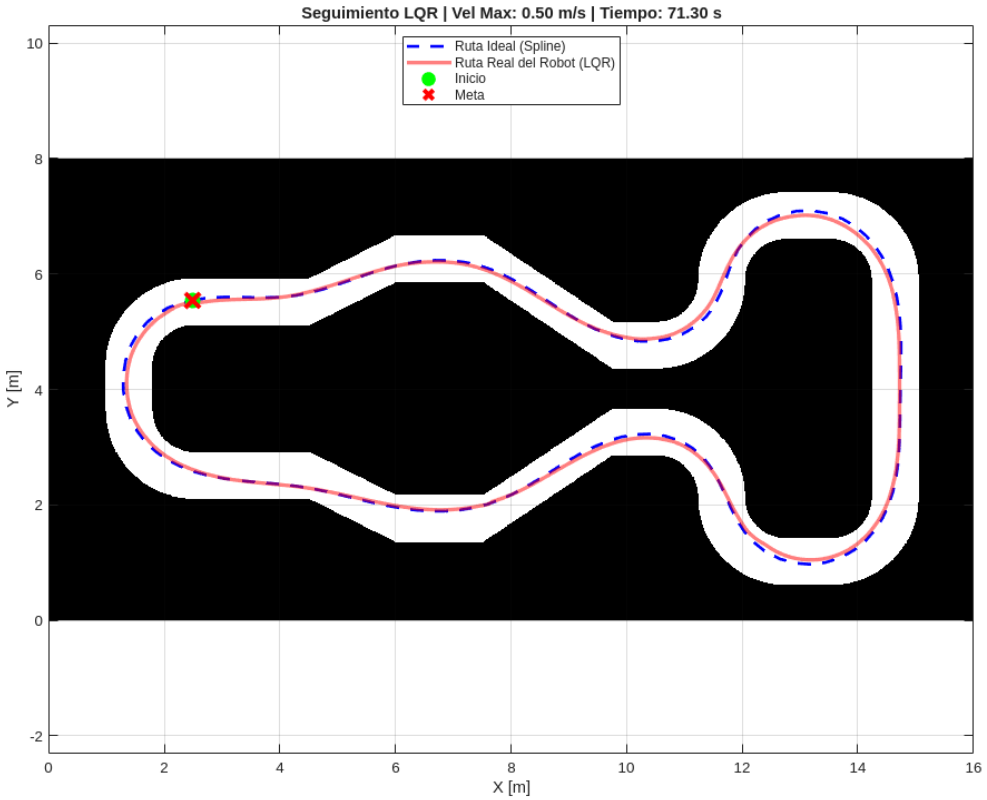


Figura 6.23: Vuelta Trayectoria Óptima LQR Circuito 2.

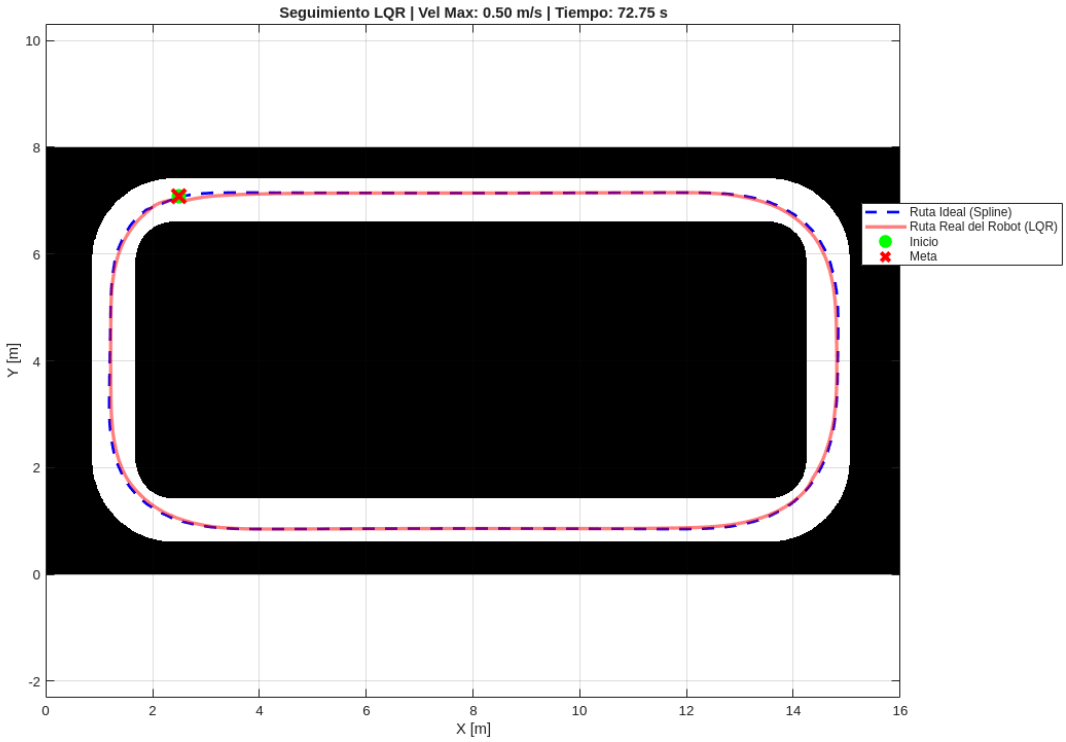


Figura 6.24: Vuelta Trayectoria Óptima LQR Circuito 3.

6.1.4 Resultados MPC

Por último, para los resultados del controlador MPC se ha procedido de la misma forma que en los casos anteriores: se ha fijado la velocidad lineal máxima en 0.5 m/s y la velocidad angular se ha limitado al rango de [-1.0, 1.0] rad/s. El horizonte de predicción, específico del MPC, se ha fijado en un valor de 10, lo que le permite al robot evaluar 10 instantes hacia el futuro (tal como se explicó en su implementación). Los resultados obtenidos tras realizar una vuelta por la trayectoria central del circuito 1 se observan en la Figura 6.25.

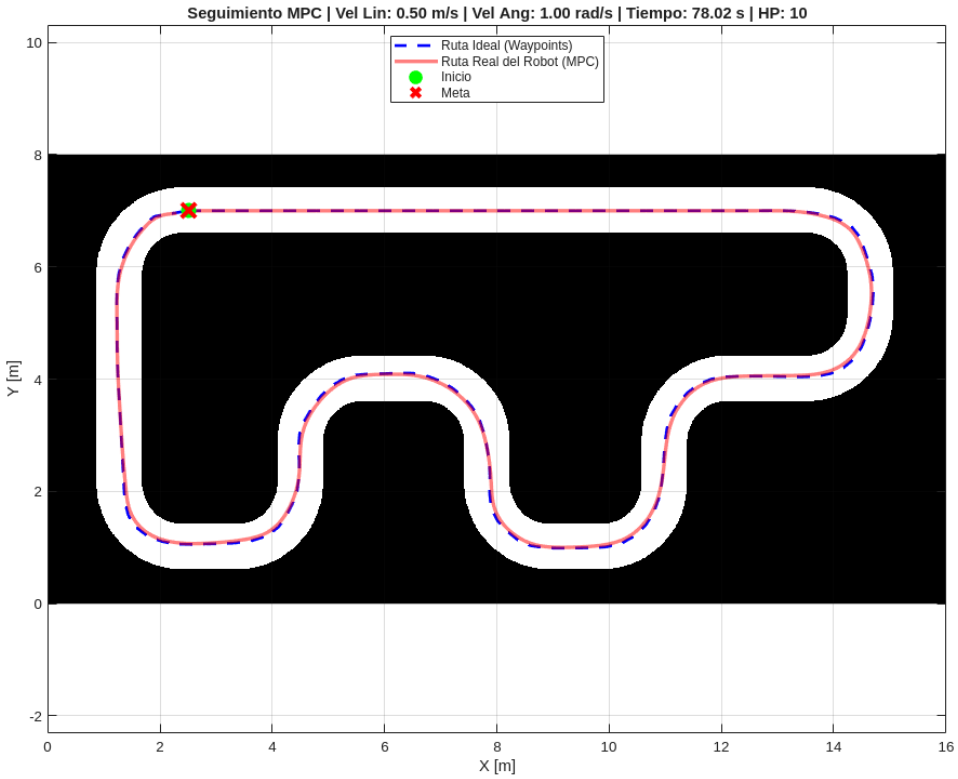


Figura 6.25: Vuelta Trayectoria Central MPC Circuito 1.

```
================================================================
Tiempo de vuelta       : 78.02 s (0.499 m/s)
----------------------------------------------------------------
Mean Lateral Error       [m]    : 0.0185 (1.9 cm)
Maximum Lateral Error    [m]    : 0.0589 (5.9 cm)
Error Lateral (RMSE)     [m]    : 0.0241 (2.4 cm)
Error Lateral (STD)      [m]    : 0.0154 (1.5 cm) [Consistencia]
Steering Oscillation RMS [deg/s]  : 16.6724
Lateral Jerk RMS         [m/s³] : 0.3965
Longitudinal Jerk RMS    [m/s³] : 0.0000
Control Effort RMS       [-]    : 0.5839
================================================================
```

Figura 6.26: Métricas Trayectoria Central MPC Circuito 1.

Según el informe de métricas del MPC (Figura 6.26), los datos extraídos muestran que el robot ha completado el trazado central en 78.02 segundos, tiempo similar a los controladores previos. Además, obtiene la mejor desviación típica con un valor de 1.5 centímetros, y un error

lateral máximo de 5.9 centímetros, lo que demuestra que es un controlador muy preciso y estable para la navegación.

Al igual que se hizo con los algoritmos anteriores, se ha llevado a cabo una nueva simulación para evaluar el comportamiento del MPC al seguir la trayectoria óptima (Figura 6.27).

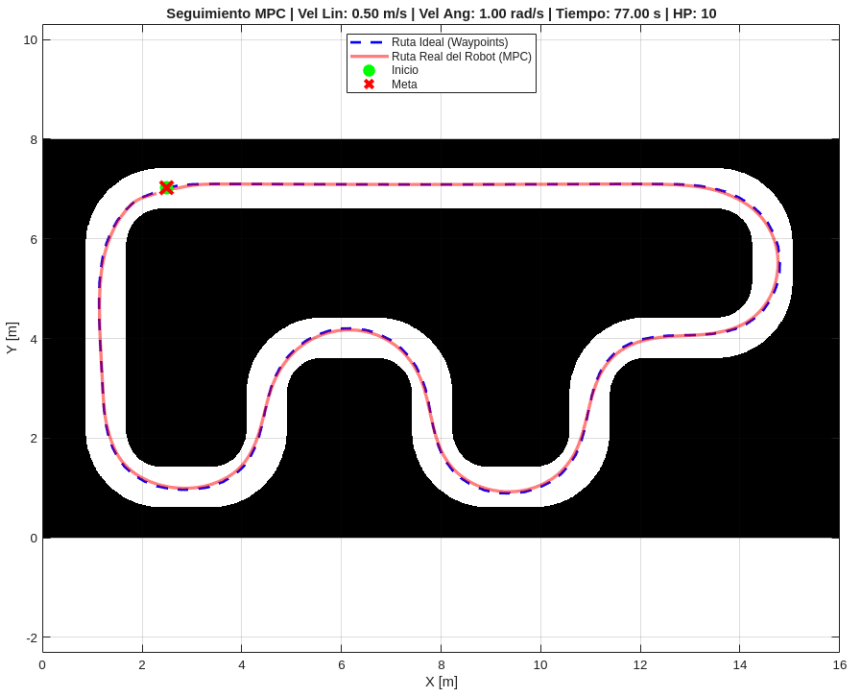


Figura 6.27: Vuelta Trayectoria Óptima MPC Circuito 1.

```
================================================================
Tiempo de vuelta       : 77.00 s (0.500 m/s)
----------------------------------------------------------------
Mean Lateral Error       [m]    : 0.0179 (1.8 cm)
Maximum Lateral Error    [m]    : 0.0454 (4.5 cm)
Error Lateral (RMSE)     [m]    : 0.0218 (2.2 cm)
Error Lateral (STD)      [m]    : 0.0125 (1.3 cm) [Consistencia]
Steering Oscillation RMS [deg/s]  : 14.1156
Lateral Jerk RMS         [m/s³] : 0.3567
Longitudinal Jerk RMS    [m/s³] : 0.0000
Control Effort RMS       [-]    : 0.5627
================================================================
```

Figura 6.28: Métricas Trayectoria Óptima MPC Circuito 1.

Los resultados obtenidos al seguir la trayectoria ideal (Figura 6.28) vuelven a demostrar que la optimización de la trazada mejora el rendimiento global del sistema, independientemente del controlador empleado. En esta última prueba, MPC ha conseguido completar el circuito en 77 segundos, reduciendo su tiempo original en un segundo. Sin embargo, el aspecto más destacable de esta prueba es el nivel de precisión alcanzado: ha logrado minimizar la desviación típica del error lateral a tan solo 1.3 centímetros, y acotar el error lateral medio y máximo a 1.8 y 4.5 centímetros, respectivamente. Ambas métricas suponen los mejores registros de todas las

pruebas sobre el circuito 1, demostrando que MPC posee una capacidad de seguimiento milimétrica y una consistencia insuperable.

Por último, al igual que se ha realizado antes, se ha desplegado el control MPC en los demás circuitos, tanto para seguir las trayectorias centrales (Figura 6.29 y Figura 6.30) como para seguir las trayectorias óptimas (Figura 6.31 y Figura 6.32).

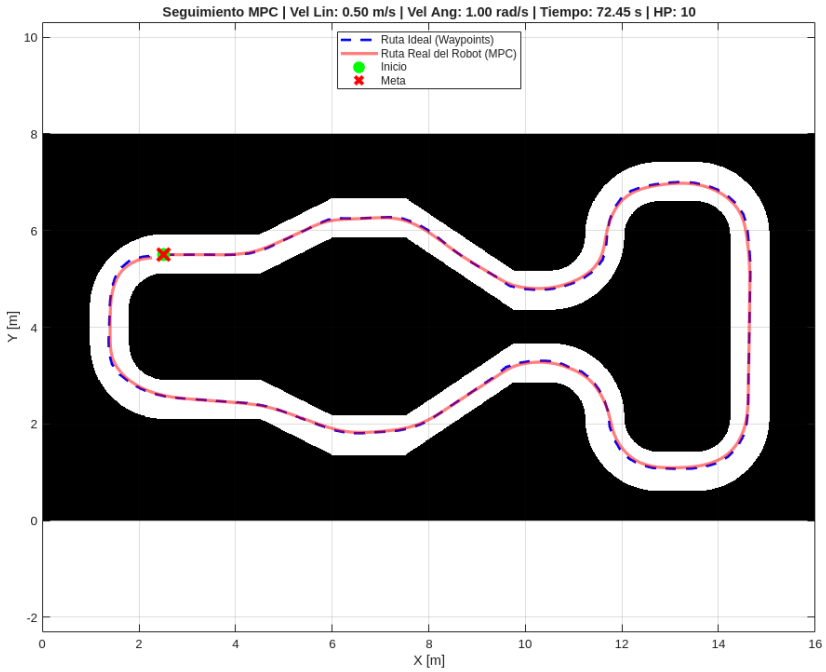


Figura 6.29: Vuelta Trayectoria Central MPC Circuito 2.

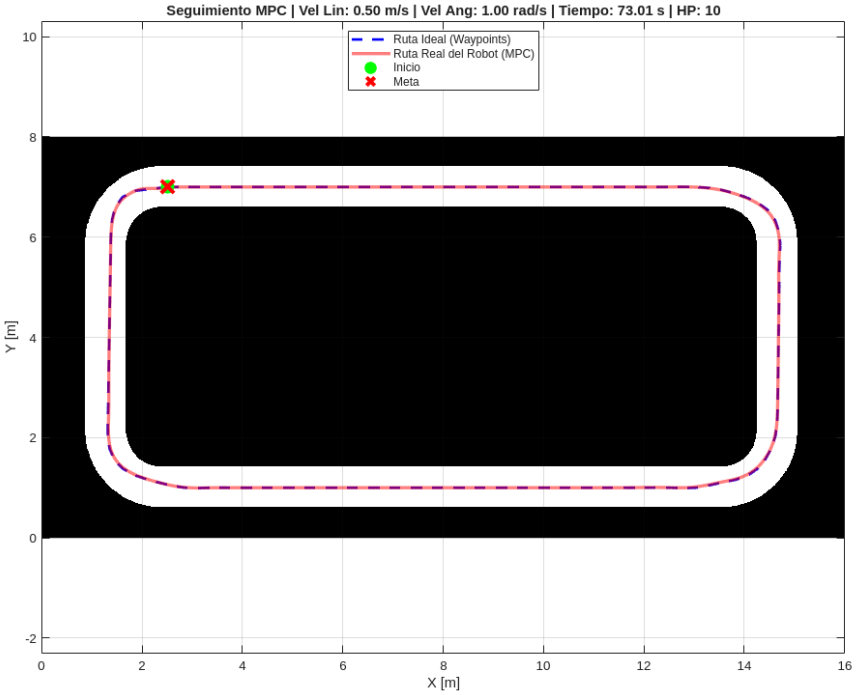


Figura 6.30: Vuelta Trayectoria Central MPC Circuito 3.

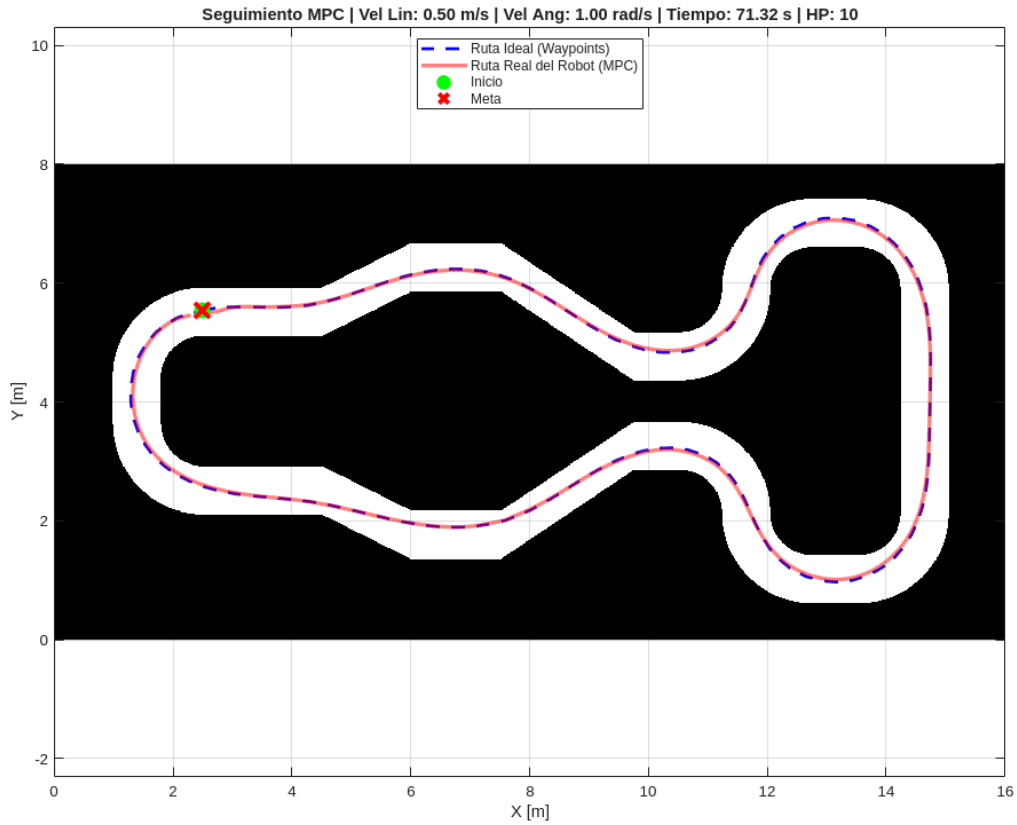


Figura 6.31: Vuelta Trayectoria Óptima MPC Circuito 2.

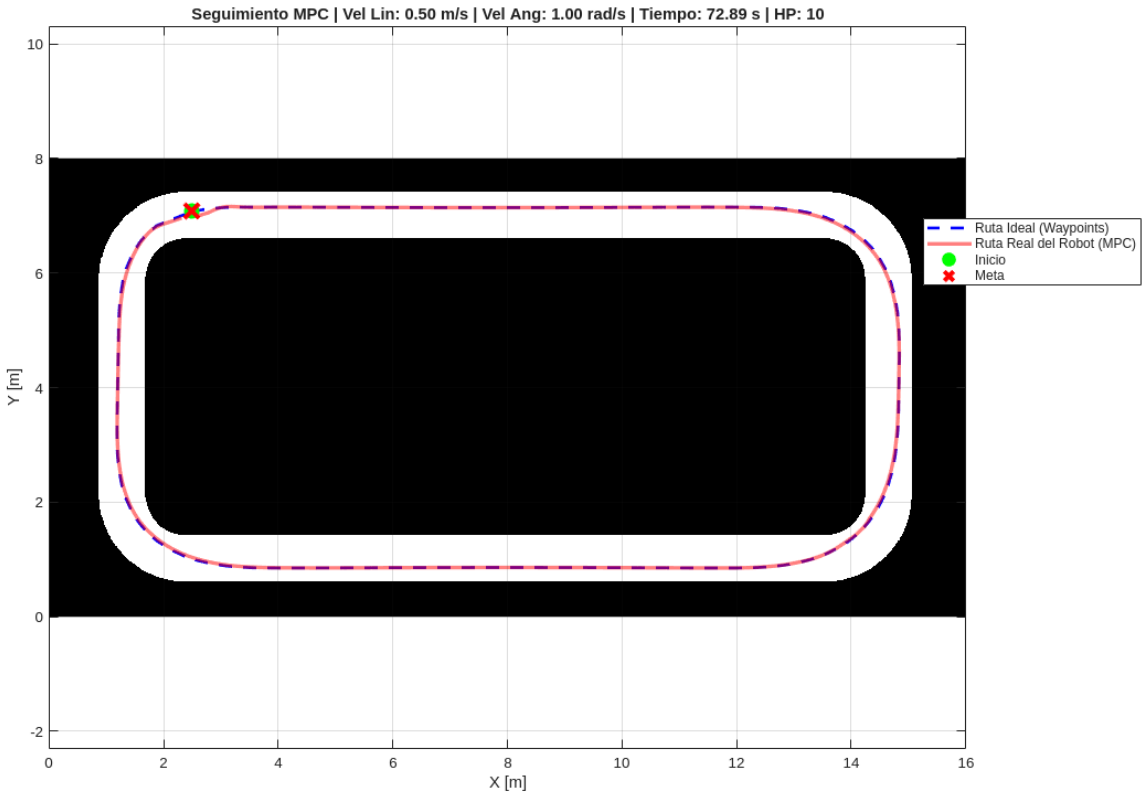


Figura 6.32: Vuelta Trayectoria Óptima MPC Circuito 3.

6.1.5 Comparación de algoritmos

Para facilitar la comparación del rendimiento entre los cuatro controladores, el análisis se abordará en dos fases. En primer lugar, se evaluarán los algoritmos clásicos operando sobre la trayectoria central, comparando su desempeño con la ruta empírica descubierta por el agente DDPG. Las métricas obtenidas en esta primera comparativa para los circuitos 1, 2 y 3 se resumen en las Tablas 6.1, 6.2 y 6.3, respectivamente. Cabe señalar que la métrica de Sacudida Longitudinal no figura en las mismas, dado que su valor es cero para la totalidad de los algoritmos al operar con una velocidad lineal constante**.** La única excepción es el agente DDPG, que presenta una ligera variación longitudinal únicamente en el momento del arranque, siendo su valor insignificante para el análisis global del recorrido.

	Tiempo (s)	**MLE (cm)**	**MAX (cm)**	**RMSE (cm)**	**STD (cm)**	**SO (deg/s)**	**LATJ (m/s^3)**	**CE (-)**
DDPG	77.30	5.3	15	6.4	3.6	22.15	0.86	0.63
PP	78.10	2.0	7.3	2.7	1.7	15.76	0.15	0.57
LQR	78	3.5	10.4	4.5	2.8	20.9	1.37	0.62
MPC	78.02	1.9	5.9	2.4	1.5	16.67	0.39	0.58

Tabla 6.1: Comparativa de rendimiento de los algoritmos de control en circuito 1.

	Tiempo (s)	**MLE (cm)**	**MAX (cm)**	**RMSE (cm)**	**STD (cm)**	**SO (deg/s)**	**LATJ (m/s^3)**	**CE (-)**
DDPG	71.09	5.7	18.5	7	4	22.03	0.85	0.63
PP	72.78	2.2	6.7	2.7	1.6	16.08	0.17	0.57
LQR	72.40	3.9	10.2	4.6	2.5	22.11	1.48	0.63
MPC	72.45	2.0	6.5	2.4	1.4	16.85	0.39	0.58

Tabla 6.2: Comparativa de rendimiento de los algoritmos de control en circuito 2.

	Tiempo (s)	**MLE (cm)**	**MAX (cm)**	**RMSE (cm)**	**STD (cm)**	**SO (deg/s)**	**LATJ (m/s^3)**	**CE (-)**
DDPG	72.60	3.3	12.2	4	2.4	19.74	0.89	0.61
PP	72.88	1	7.2	1.8	1.4	10.55	0.12	0.53
LQR	72.85	1.6	8.1	2.5	2	13.89	0.86	0.56
MPC	73.01	0.5	2.6	0.7	0.6	11.59	0.23	0.54

Tabla 6.3: Comparativa de rendimiento de los algoritmos de control en circuito 3.

A partir de los datos recopilados en los tres circuitos, se extraen las siguientes conclusiones sobre el desempeño de cada estrategia de control en función del criterio de diseño que se decida priorizar.

En primer lugar, dado que el objetivo fundamental de este proyecto es precisamente maximizar la velocidad y minimizar el tiempo por vuelta, el agente DDPG se consolida como el

método más rápido en términos generales, liderando todos los circuitos con tiempos de 77.30, 71.09 y 72.60 segundos para los circuitos 1, 2 y 3, respectivamente. Este resultado indica que DDPG ha aprendido a sacrificar el seguimiento estricto del centro del carril para encontrar su propia trazada óptima recortando los vértices de las curvas, manteniendo así una mayor inercia.

Por otro lado, cuando se evalúa la precisión geométrica y el requisito de seguridad exige un seguimiento milimétrico, la excelencia vuelve a recaer sobre MPC. Aunque el algoritmo *Pure Pursuit* presenta métricas muy buenas, MPC logra superarlo consistentemente en todos los circuitos, registrando los valores más bajos de Error Medio Lateral (MLE), Error Cuadrático Medio (RMSE) y Desviación Típica (STD) en el circuito 3 con 0.5, 0.7 y 0.6 centímetros, respectivamente. Esta precisión demuestra la superioridad analítica del modelo predictivo frente al geométrico, ya que MPC logra esta exactitud rodando más rápido que *Pure Pursuit*. Por el contrario, tal como se ha explicado anteriormente, DDPG presenta los mayores márgenes de error lateral debido a que se ha entrenado para buscar el límite del circuito, con el objetivo final de obtener la trayectoria óptima de forma automática.

En cuanto a la estabilidad dinámica y el confort de marcha de la plataforma, el análisis de las métricas establece una división clara. *Pure Pursuit* destaca por ofrecer la conducción más fluida de todas, registrando los valores mínimos de Oscilación de la Dirección (SO) y un *Lateral Jerk* (LATJ) excepcionalmente bajo, que no supera los 0.17 m/s^3. A este le sigue de cerca MPC, manteniendo una estabilidad excelente. Por el contrario, tanto el LQR como el DDPG muestran un comportamiento más nervioso. LQR penaliza mucho en las aceleraciones laterales, registrando picos de *jerk* cercanos a 1.5 m/s^3 que se traducirían en "volantazos" bruscos. Esto es debido a que actúa de forma reactiva, ya que, al no poseer un horizonte de predicción, el controlador responde únicamente al error actual del sistema en cada instante, sin capacidad de anticipar la trayectoria futura. Asimismo, la elevada oscilación de la dirección del DDPG (alcanzando los 22.15 deg/s) refleja que su rápida navegación se logra a base de correcciones constantes, sacrificando la suavidad operativa.

Finalmente, el análisis del Esfuerzo de Control (CE) muestra la eficiencia energética de cada algoritmo y su impacto en el desgaste de los actuadores del robot. Las estrategias anticipativas y suaves, como *Pure Pursuit* y MPC, exigen la menor cantidad de energía combinada al sistema, manteniendo su índice entre 0.53 y 0.58 a lo largo de las pruebas. Por el contrario, la agresividad del LQR y las constantes "microcorrecciones" del agente DDPG se traducen en un mayor requerimiento de actuación, elevando el esfuerzo de control por encima de 0.61. No obstante, hay que remarcar que este comportamiento del DDPG es consecuencia del diseño de su función de recompensa. Aunque esta incluye un término para evitar oscilaciones severas, no penaliza la magnitud absoluta del esfuerzo motor. Al tener como objetivo prioritario maximizar la velocidad, la red neuronal recurre a estas "microcorrecciones" que apuran los límites de la conducción, elevando el esfuerzo de control general. Por este motivo, aunque el agente de RL carezca del cuidado mecánico que los controladores clásicos garantizan, cumple de forma competitiva con el objetivo de minimizar el tiempo para el que fue entrenado.

En segundo lugar, para profundizar en el análisis, resulta de gran utilidad comparar la trayectoria generada autónomamente por el agente DDPG con las trayectorias óptimas precalculadas en los apartados anteriores para los algoritmos de control clásicos. Dado que en las evaluaciones iniciales el algoritmo de aprendizaje por refuerzo partía con cierta ventaja al deducir atajos (recortar curvas), esta nueva comparativa permite evaluar de forma justa si la ruta descubierta por la red neuronal DDPG es realmente superior, en términos de tiempo por vuelta, frente a las optimizadas que minimizan la curvatura, explicadas en apartados anteriores. Dichos resultados al completar los circuitos 1, 2 y 3 sobre la trayectoria óptima se resumen en las Tablas 6.4, 6.5 y 6.6, respectivamente.

	Tiempo (s)	**MLE (cm)**	**MAX (cm)**	**RMSE (cm)**	**STD (cm)**	**SO (deg/s)**	**LATJ (m/s^3)**	**CE (-)**
DDPG	77.30	5.3	15	6.4	3.6	22.15	0.86	0.63
PP	77.49	2.0	7.3	2.5	1.5	12.98	0.09	0.55
LQR	76.80	3.7	8.9	4.6	2.7	18.2	1.32	0.59
MPC	77	1.8	4.5	2.2	1.3	14.11	0.35	0.56

Tabla 6.4: Comparativa de rendimiento de los algoritmos de control en circuito 1 sobre trazada óptima

	Tiempo (s)	**MLE (cm)**	**MAX (cm)**	**RMSE (cm)**	**STD (cm)**	**SO (deg/s)**	**LATJ (m/s^3)**	**CE (-)**
DDPG	71.09	5.7	18.5	7	4	22.03	0.85	0.63
PP	71.69	2.1	4.9	2.5	1.3	12.77	0.09	0.55
LQR	71.30	3.7	9.6	4.4	2.4	17.66	1.33	0.59
MPC	71.32	1.8	6.3	2.2	1.2	14.03	0.38	0.56

Tabla 6.5: Comparativa de rendimiento de los algoritmos de control en circuito 2 sobre trazada óptima

	Tiempo (s)	**MLE (cm)**	**MAX (cm)**	**RMSE (cm)**	**STD (cm)**	**SO (deg/s)**	**LATJ (m/s^3)**	**CE (-)**
DDPG	72.60	3.3	12.2	4	2.4	19.74	0.89	0.61
PP	72.69	1	7.4	1.5	1.1	6.77	0.07	0.52
LQR	72.75	1.9	8.6	2.6	1.8	10.86	0.77	0.54
MPC	72.89	0.9	8.6	1.4	1	9.74	0.30	0.53

Tabla 6.6: Comparativa de rendimiento de los algoritmos de control en circuito 3 sobre trazada óptima

Al evaluar los tiempos en estas nuevas tablas, se hace evidente el impacto de la trayectoria optimizada. Al dotar a los controladores clásicos de una trazada que optimiza la geometría del recorrido bajo la lógica “exterior-interior-exterior”, la brecha temporal respecto al agente DDPG se reduce significativamente, dando lugar a una comparativa mucho más ajustada. En el Circuito 1, el LQR (76.80 s) y el MPC (77.00 s) logran superar por muy pocas décimas al agente DDPG (77.30 s). Sin embargo, tanto en el Circuito 2 como en el Circuito 3, el agente

DDPG sigue manteniendo el liderazgo, obteniendo una ventaja temporal sobres los demás controladores de 0.21 y 0.09 segundos, respectivamente. Estos resultados demuestran que DDPG se mantiene como un competidor excelente, demostrando su rapidez y su gran capacidad de generalización en cualquier tipo de trazado.

Respecto al resto de variables de precisión geométrica, estabilidad dinámica y esfuerzo de control, el comportamiento general de los algoritmos clásicos mantiene una línea muy similar a la observada en las pruebas con la trayectoria central. MPC y *Pure Pursuit* continúan destacando por su equilibrio entre precisión y fluidez. Por un lado, MPC sigue registrando los menores errores medios laterales (MLE), bajando hasta los 0.9 centímetros en el Circuito 3; mientras que, por otro, *Pure Pursuit* ofrece la conducción más suave con valores mínimos de *Lateral Jerk* (LATJ) y oscilación (SO). Por su parte, LQR vuelve a demostrar su naturaleza más agresiva, penalizando en las transiciones curvas con picos de *Lateral jerk* más elevados debido a su carácter reactivo.

Como conclusión general para todo este apartado, al analizar lado a lado ambos escenarios (trayectoria central y trayectoria óptima), se ha demostrado que dotar a los algoritmos clásicos de la trayectoria óptima supone un gran beneficio. Estos mejoraron significativamente sus tiempos de vuelta, logrando neutralizar la ventaja temporal inicial que tenía el agente DDPG al deducir atajos de forma empírica.

Bajo estas condiciones, es evidente que no existe un "algoritmo definitivo", sino soluciones idóneas que dependen del objetivo principal a conseguir. Por un lado, el DDPG ha demostrado ser una de las opciones más competitivas para minimizar el tiempo de vuelta, sobresaliendo especialmente por su capacidad única de deducir la trazada óptima por su cuenta a través de la experiencia directa, lo que demuestra el enorme potencial del aprendizaje por refuerzo. Sin embargo, esta aparente superioridad solo se sostiene bajo las condiciones de la simulación, ya que estos resultados pertenecen a un entorno totalmente idealizado. Al carecer de fricciones con el suelo, ruido en la odometría o desgaste mecánico, el agente puede ejecutar su agresiva política de control sin sufrir consecuencias. Como se analizará a continuación, al trasladar este comportamiento al mundo físico, estas dinámicas no modeladas penalizarán bastante más al DDPG.

Por otro lado, enfoques geométricos como el *Pure Pursuit* se mantienen como la opción ideal para sistemas limitados computacionalmente que exigen un seguimiento fiable y un mínimo desgaste mecánico; mientras que el MPC se consolida indiscutiblemente como el algoritmo más balanceado del estudio, justificando su estatus como el estándar de la industria.

6.2 Resultados en robot real

En este apartado se presentan los resultados obtenidos con el robot *Turtlebot 4* físico tras seguir una trayectoria de *waypoints*, obtenida ya sea mediante teleoperación y *waypoints* cada 0.5 metros, o mediante los códigos ArUco. En este caso, se ha probado hacer la trayectoria con la primera alternativa siguiendo la trazada de la Figura 6.33, que a posterior fue suavizada por *Splines* (tal como se observa en la Figura 6.34).



Figura 6.33: Circuito real.

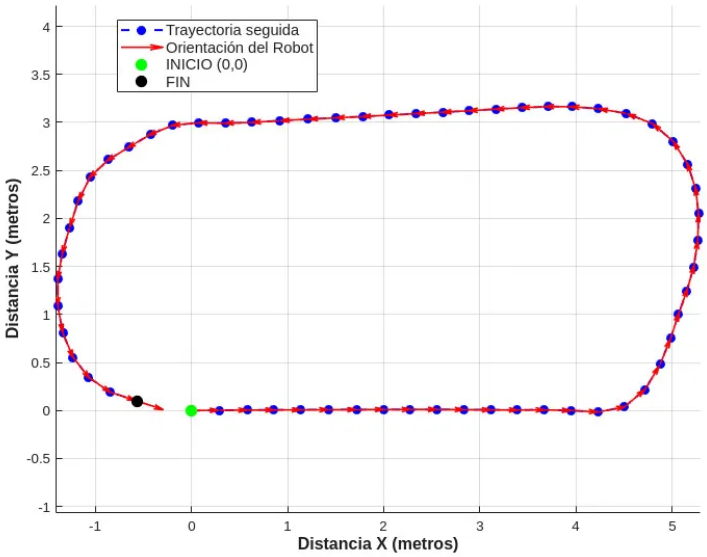


Figura 6.34: Trayectoria del circuito real.

Una vez definida la trayectoria a seguir, simplemente se ejecutan los mismos códigos implementados para el simulador. Únicamente debemos asegurarnos de conectarnos correctamente a los canales de comunicación del robot (como *robot8/odom* para el robot 8), ajustar la velocidad máxima permitida a 0.3 m/s (que es la velocidad lineal máxima del *Turtlebot 4* físico), y pasar el archivo de *waypoints* del nuevo circuito a seguir. A continuación, se muestran las figuras para los 4 métodos de control, que posteriormente se comparan en la Tabla 6.7.

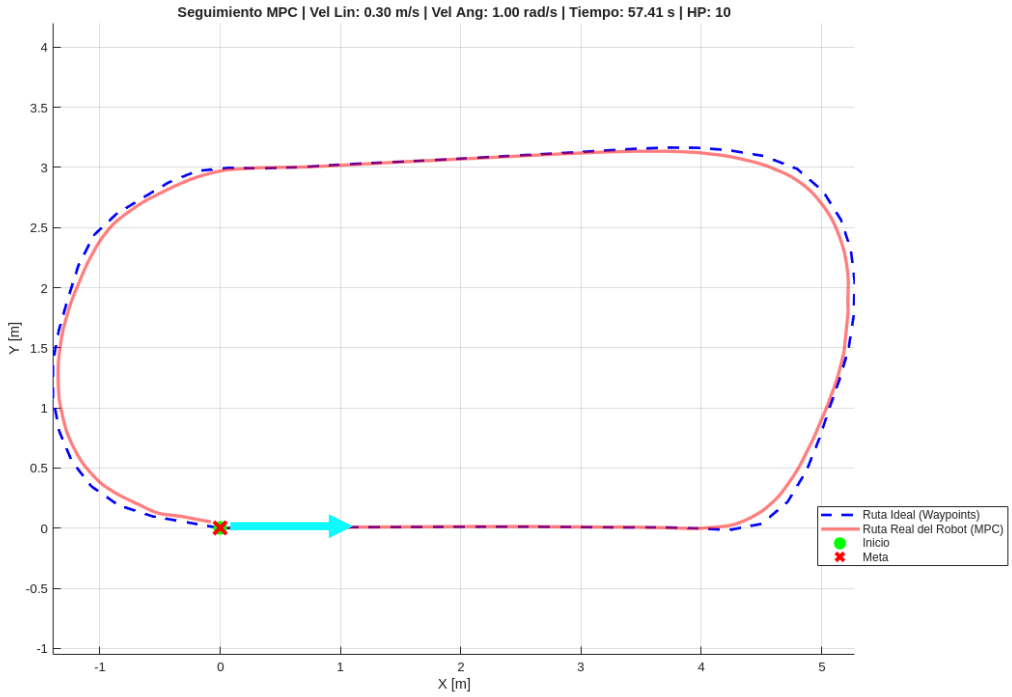


Figura 6.35: Vuelta MPC con circuito real.

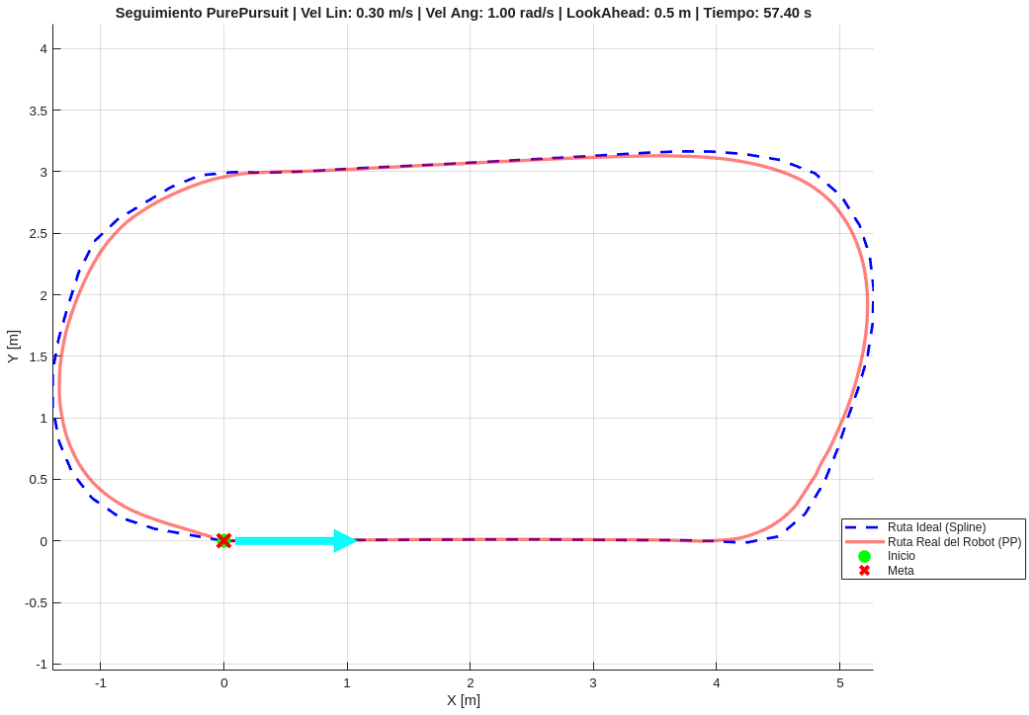


Figura 6.36: Vuelta Pure Pursuit con circuito real.

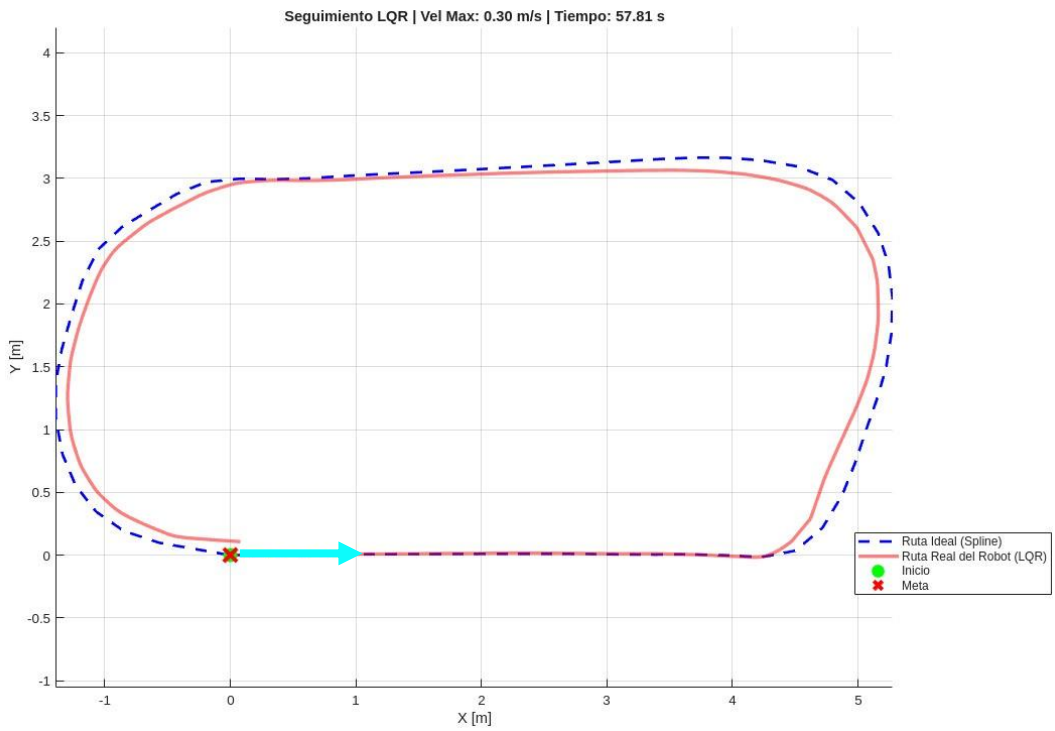


Figura 6.37: Vuelta LQR con circuito real.

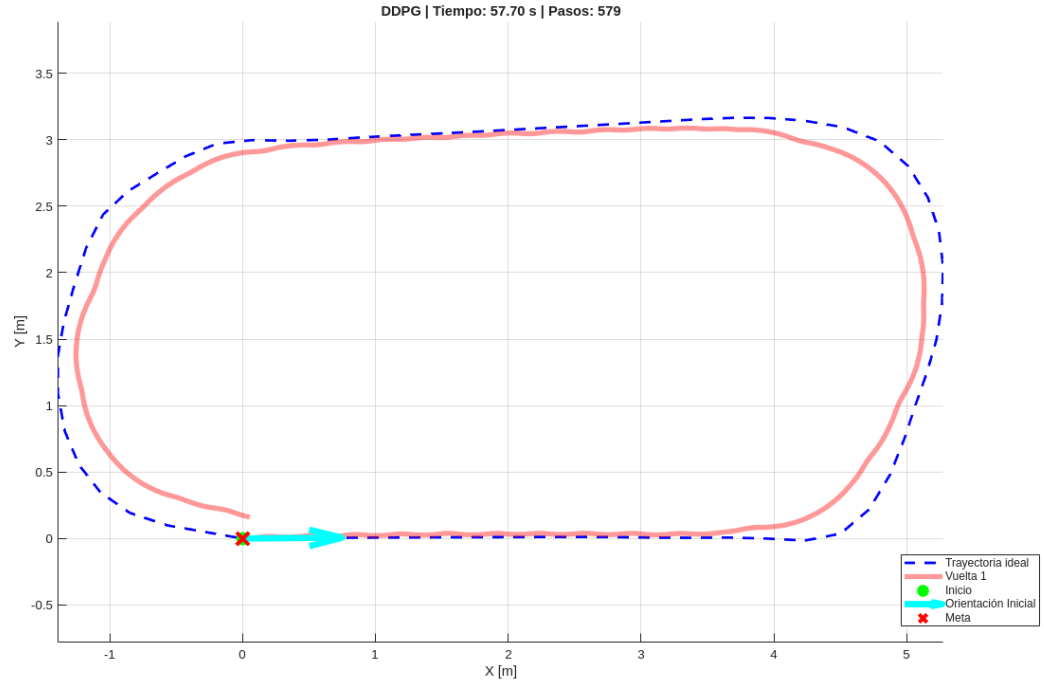


Figura 6.38: Vuelta DDPG con circuito real.

	Tiempo (s)	MLE (cm)	MAX (cm)	RMSE (cm)	STD (cm)	SO (deg/s)	LATJ (m/s^3)	CE (-)
DDPG	57.70	10	24	12.3	7.2	23.44	0.94	0.58
PP	57.40	3.7	10	4.9	3.2	7.41	0.11	0.34
LQR	57.81	6.8	18	8.4	4.9	11.86	0.30	0.38
MPC	57.41	3	8.6	3.9	2.5	8.9	0.16	0.35

Tabla 6.7: Comparativa de rendimiento de los algoritmos de control en el circuito real.

El análisis de las métricas obtenidas tras el despliegue de los cuatro controladores sobre el robot físico demuestra diferencias importantes en el comportamiento dinámico, la precisión geométrica y la carga sobre los actuadores. A partir de los resultados consolidados en la Tabla 6.7 y las trazadas registradas, se extraen las siguientes conclusiones:

1. **MPC.** Nuevamente se consolida como el algoritmo más preciso de la comparativa, justificando ser el estándar de referencia. Este logra el menor error cuadrático medio (RMSE de 3.9 cm), así como el menor error lateral medio (MLE de 3 cm) y máximo (8.6 cm). Al observar su comportamiento, se aprecia cómo su horizonte de predicción le permite anticiparse a las variaciones inmediatas de la trazada, minimizando la desviación sin introducir oscilaciones abruptas. Además, logra este excelente seguimiento con una alta eficiencia energética (CE de 0.35), demostrando un equilibrio muy bueno entre precisión geométrica y un tiempo de vuelta muy competitivo (57.41 s).
2. **Pure Pursuit.** Este algoritmo vuelve a sorprender al ofrecer el comportamiento más eficiente y suave mecánicamente de la prueba. Su naturaleza geométrica le permite guiar al robot con el menor esfuerzo de control de toda la comparativa (CE de 0.34) y la menor tasa de aceleraciones laterales (*lateral jerk* de 0.11 m/s^3). Esto se traduce en la vida real en una conducción sumamente fluida y continua, sin estrés para los motores físicos. El uso del *lookahead* hace que el robot tienda a "recortar" ligeramente el vértice de las curvas, por lo que recorre menos distancia espacial y obtiene el mejor tiempo de vuelta de la tabla (57.40 s). Todo esto manteniendo una precisión geométrica altamente robusta, con un error máximo contenido de 10 centímetros y un RMSE de tan solo 4.9.
3. **LQR.** Si bien el controlador LQR logra mantener al robot dentro de la pista, su despliegue físico evidencia las limitaciones de su naturaleza reactiva. Al carecer del horizonte de predicción o "visión de futuro" que poseen algoritmos como MPC o *Pure Pursuit*, el sistema se ve obligado a "perseguir" la trayectoria. Para proteger a los motores físicos, se aprovechó la flexibilidad analítica de este controlador incrementando el peso de la velocidad angular en la matriz de penalización R (pasando de 0.5 a 12). Este ajuste logró estabilizar el esfuerzo de control (CE de 0.38), pero al suavizar las reacciones del robot, aumentó significativamente su error geométrico (RMSE de 8.4 cm y un error

lateral máximo de 18 cm). A consecuencia de estas correcciones tardías, acaba registrando el peor tiempo de la comparativa (57.81 s).

4. **DDPG.** El agente entrenado mediante RL presenta unos resultados bastante positivos. Por un lado, demuestra ser competitivo en velocidad pura, logrando un tiempo de 57.70 segundos, superando al LQR. Sin embargo, su conducción evidencia claramente el impacto del *sim-to-real gap* (la brecha entre simulación y realidad). El modelo virtual donde el agente fue entrenado carece de la fricción exacta de la pista real, el ruido de los sensores y las latencias de procesamiento. Ante estas perturbaciones del mundo físico, la red neuronal sobrerreacciona de forma continua. Este "zigzagueo" se refleja matemáticamente en unos valores de oscilación (SO de 23.44 deg/s) y tirones laterales (LATJ de 0.94 m/s^3) que triplican a los de los algoritmos clásicos. Finalmente, aunque su error cuadrático medio y desviación máxima son los más elevados, este comportamiento obedece a la naturaleza intrínseca del agente RL. Al priorizar la maximización de la velocidad, la red neuronal no intenta calcar la línea geométrica de referencia, sino que busca de forma activa la trayectoria más óptima y rápida a través de la pista.

A modo de conclusión para este apartado, la diferencia de rendimiento del agente DDPG frente a los controladores clásicos pone de manifiesto uno de los mayores desafíos en el campo de la robótica y el RL: la ya mencionada brecha entre simulación y realidad (*sim-to-real gap*). Durante la fase de entrenamiento, la red neuronal optimizó su política de control interactuando miles de veces con un entorno virtual formado por ecuaciones cinemáticas totalmente ideales. Sin embargo, al desplegarse sobre el *hardware* físico del *Turtlebot 4*, el agente se enfrenta de repente a dinámicas complejas que no estaban presentes en su entorno de aprendizaje. Factores como el ruido en la estimación de la odometría, la fricción asimétrica de las ruedas con el suelo, la inercia del chasis y, sobre todo, la latencia computacional entre el cálculo de la acción y su ejecución real en los motores, alteran por completo la respuesta esperada por la red.

Mientras que algoritmos analíticos como el MPC o el Pure Pursuit presentan cierta robustez ante estas perturbaciones físicas, la política del DDPG sufre al estar "sobreajustada" (*overfitted*) a un mundo perfecto o ideal. Al percibir que el robot no reacciona exactamente como en la simulación, la red neuronal comienza a emitir comandos excesivos de corrección, lo que acaba penalizando tanto la precisión como la fluidez del recorrido. Pese a ello, las métricas físicas demuestran que el agente DDPG con el robot real sigue siendo una alternativa muy viable, logrando posicionarse como la tercera mejor opción y superando el rendimiento de controladores clásicos como el LQR.

Capítulo 7

7. Conclusión y líneas futuras

7.1 Conclusiones

En este Trabajo Fin de Grado se ha desarrollado y validado un marco completo para la planificación y el control de trayectorias en vehículos autónomos, aplicado a un robot de tracción diferencial *TurtleBot 4 Lite*. Un pilar fundamental de este proyecto ha sido el estudio de la planificación de las trayectorias de referencia. Se ha demostrado que estas pueden generarse de manera efectiva mediante teleoperación, posicionando *waypoints* a distancias regulares, o bien mediante el uso de marcadores ArUco, un método que aporta una adaptabilidad y flexibilidad ante cambios en el entorno. Asimismo, el proceso de obtención de la trayectoria de carreras optimizada (*racing line*) permitió refinar el comportamiento de los controladores analíticos tradicionales (Pure Pursuit, MPC y LQR), logrando reducir con éxito el tiempo de vuelta respecto a las trazadas convencionales.

En el plano del aprendizaje por refuerzo, el entrenamiento del agente DDPG supuso un enorme desafío de diseño. Por un lado, encontrar el equilibrio exacto en la función de recompensa, entre las bonificaciones por mantener la inercia y las penalizaciones por salidas o giros bruscos, resultó ser una tarea muy compleja. Por otro lado, la estabilización del aprendizaje exigió un ajuste minucioso de los hiperparámetros de las redes neuronales, tales como la tasa de aprendizaje (*learning rate*) o la capacidad de la memoria de experiencia (*replay buffer*). La combinación de ambos retos requirió múltiples iteraciones empíricas hasta lograr que el modelo convergiera hacia el comportamiento deseado.

Una vez ajustada la función de recompensa y los hiperparámetros, el proceso de entrenamiento presentó nuevos obstáculos. En una primera instancia, se intentó entrenar al agente exponiéndolo de forma directa al circuito completo, una estrategia que resultó ineficiente debido a la excesiva complejidad del espacio de estados. Para intentar solucionar esto, se implementó un enfoque de *Curriculum Learning* dividiendo el trazado en sectores progresivos; sin embargo, esta metodología introdujo el problema de sobreajuste (*overfitting*), provocando que el agente sea incapaz de generalizar la navegación. La solución definitiva se alcanzó mediante la implementación de *Task Randomization* aplicada a dichos sectores, alternando aleatoriamente las condiciones de inicio y las configuraciones de los tramos. Esta aleatoriedad rompió el sobreajuste y permitió consolidar una política de control verdaderamente robusta y convergente en simulación.

La fase de evaluación en el simulador MVSim demostró que, bajo condiciones virtuales perfectas, no existe un algoritmo superior en términos absolutos, sino que la mejor solución

dependerá de lo que se intente priorizar. El agente DDPG logró los tiempos por vuelta más rápidos al aproximarse de forma autónoma al límite teórico de 15 centímetros. Este comportamiento confirma que la red neuronal fue capaz de inferir la "*racing line"* matemática por sí sola, sacrificando el centro del carril para recortar las curvas con la máxima eficiencia. Por su parte, *Pure Pursuit* destacó por su suavidad operativa, ideal para sistemas con recursos escasos, mientras que el algoritmo MPC se consolidó como la opción más equilibrada en simulación.

El traspaso de estos sistemas al robot físico puso de manifiesto el conocido desafío del *sim-to-real gap*. Las limitaciones prácticas de la realidad, como el deslizamiento físico con el suelo o el desgaste mecánico, alteraron el rendimiento de los algoritmos, debiendo incluso reajustar sus parámetros, como ocurrió en el caso del LQR. Además, como era previsible debido a las diferencias con la realidad, la ventaja competitiva que mostraba el DDPG en el entorno virtual disminuyó frente a estas dinámicas no modeladas.

En definitiva, a pesar de los desafíos que supone el salto al mundo físico, el balance global del proyecto alcanza unos resultados muy positivos. El agente DDPG ha sido capaz de descubrir de forma totalmente autónoma la trayectoria dinámica óptima sin depender de directrices geométricas previas, quedando demostrado el enorme potencial del aprendizaje por refuerzo. Este hito le ha permitido consagrarse en la simulación como el algoritmo más rápido de la comparativa, logrando superar el rendimiento de métodos clásicos ampliamente consolidados. Con la culminación de este trabajo, se consolida un agente muy competitivo que posiciona a la inteligencia artificial como una herramienta robusta y de vanguardia para la navegación robótica autónoma de carreras.

7.2 Líneas futuras

Como continuación natural de los desarrollos alcanzados en este Trabajo Fin de Grado, se abren diversas vías de investigación para escalar y perfeccionar el sistema. Desde el punto de vista de la infraestructura de software, resulta de gran interés plantear la migración de todo el marco de entrenamiento y control (actualmente desarrollado en MATLAB) hacia entornos basados en Python. Esta integración más nativa con el ecosistema de ROS2 facilitaría la transición hacia simuladores mucho más realistas orientados a la conducción autónoma compleja, como CARLA. Asimismo, esta evolución permitiría desacoplar la arquitectura del *TurtleBot 4 Lite*, abriendo la puerta a testear estas mismas políticas de navegación en modelos de vehículos con diferentes configuraciones cinemáticas, como la dirección de tipo Ackermann.

Desde el punto de vista del RL, las limitaciones temporales del proyecto restringieron la experimentación profunda al agente DDPG. Una línea futura consistiría en implementar y evaluar otros algoritmos de vanguardia para espacios de acción continua que presentan rendimientos similares, como es el caso de SAC (Soft Actor-Critic) o TD3. Estos algoritmos podrían ofrecer una exploración del entorno diferente y generar políticas de control más estables frente

a las incertidumbres del mundo físico. En paralelo, y con el objetivo de mitigar de forma definitiva el *sim-to-real gap*, convendría incorporar técnicas de *Domain Randomization* durante las fases de entrenamiento virtual, alterando de forma aleatoria la masa del robot o los coeficientes de fricción del suelo, para forzar a las redes neuronales a aprender comportamientos mucho más robustos.

Finalmente, desde la perspectiva de la navegación autónoma pura, el sistema actual opera sobre trayectorias estáticas precalculadas y en escenarios despejados, dependiendo en gran medida de la odometría, lo que a largo plazo origina problemas de deriva acumulativa (*drift*). La evolución lógica del proyecto requiere la integración de algoritmos de localización y mapeo simultáneo (SLAM) en tiempo real, combinados con sensores adicionales como escáneres LiDAR. Esto, sumado a la introducción de obstáculos dinámicos en los circuitos de prueba, supondría el último gran paso para dotar al vehículo de una autonomía completa, reactiva y capaz de replanificar su ruta ante imprevistos, alineando el proyecto con las exigencias operativas reales del sector automotriz.

Apéndice A. Presupuesto

Este apéndice contiene un presupuesto estimado para la ejecución del proyecto, que se compone de las siguientes partidas:

- Costes asociados a los materiales de software y hardware implicados en el proyecto.
- Costes de personal para cada una de las principales tareas en las que se ha dividido el proyecto.
- Coste total del proyecto, incluyendo los impuestos españoles aplicables (IVA 21%).

A.1. Costes de software y hardware

Respecto a los recursos utilizados, el proyecto se apoya en un ordenador de sobremesa para el estudio teórico, el desarrollo del código y la redacción de la memoria, así como en un robot *TurtleBot 4 Lite* para la realización de las pruebas reales. En lo referente a las herramientas de *software*, la selección se ha basado mayormente en soluciones de código abierto y gratuitas (*open-source*), constituyendo la licencia de uso de MATLAB el único gasto imputable en esta categoría.

Concepto	**Precio**
Hardware	
Ordenador sobremesa	1.000 €
Robot TurtleBot 4 Lite	1.700 €
Software	
Licencia académica MATLAB	300 €
Ubuntu OS	0 €
Robot Operating System (ROS)	0 €
MVSim	0 €
Total	**3.000 €**

Tabla A.1: Presupuesto de recursos hardware y software.

A.2. Costes de personal

Se ha considerado una media de 4 horas de trabajo diario, 22 días laborables al mes y un salario de 10 €/h. El precio por cada etapa del proyecto se refleja en la siguiente tabla:

Actividad	**Duración (días)**	**Precio**
Investigación y revisión de literatura	**7**	280 €
Análisis del problema y preparación del entorno	7	280 €
Planificación de las trayectorias	14	560 €
Desarrollo de los algoritmos de control clásicos	14	560 €
Diseño, entrenamiento y ajuste del agente RL	30	1.200 €
Pruebas en el simulador MVSim	14	560 €
Pruebas en el TurtleBot 4 real	14	560 €
Redacción del TFG	30	1.200 €
Total		**5.200 €**

Tabla A.2: Costes de personal.

A.3. Coste final del proyecto

Sumando todos los costes anteriores y aplicando el IVA (21 %), se obtiene el coste final del proyecto:

Concepto	**Precio**
Costes hardware y software	3.000 €
Costes de personal	5.200 €
Subtotal	8.200 €
IVA (21%)	1.722 €
Total	**9.922 €**

Tabla A.3: Coste total del proyecto.

Bibliografía

[1] MathWorks, "Reinforcement Learning Toolbox," 2026. [Online]. Available: https://es.mathworks.com/products/reinforcement-learning.html.

[2] M. Segovia, "Un 16% de los nuevos coches en España puede conducir de forma autónoma.," *El Independiente,* 20 11 2025.

[3] C. Barbero Ruiz, "Comparative analysis of MPC controllers applied to Autonomous," Alcalá de Henares, 2021.

[4] The MathWorks Inc., "Pure Pursuit Controller," 2026. [Online]. Available: https://www.mathworks.com/help/nav/ug/pure-pursuit-controller.html.

[5] N. A. A. lah, "Controlador híbrido basado en MPC y “splines” para maniobras de cambio de carril en vehículos autónomos," Alcalá de Henares, 2022.

[6] E. Fernández Camacho and C. Bordons Alba, "Control predictivo: Pasado, presente y futuro," *Revista Iberoamericana de Automática e Informática Industrial,* vol. 1, no. 3, 2004.

[7] AutomaticAddison, "automaticaddison.com," 13 Diciembre 2020. [Online]. Available: https://automaticaddison.com/linear-quadratic-regulator-lqr-with-python-code-example/.

[8] AutomaticAddison, "automaticaddison.com," 10 Diciembre 2020. [Online]. Available: https://automaticaddison.com/how-to-derive-the-state-space-model-for-a-mobile-robot/.

[9] S. Bittanti, A. J. Laub and J. C. Willems, The Riccati Equation, 2012: Springer Science & Business.

[10] MathWorks, "Introducción a Reinforcement Learning," [Online]. Available: mathworks.com/discovery/reinforcement-learning.html.

[11] S. L. Brunton and J. N. Kutz, Data-Driven Science and Engineering (Machine Learning, Dynamical Systems, and Control), Segunda ed., Cambridge University Press, 2022.

[12] C. Dorhoi, "Navegación autónoma mediante técnicas de reinforcement learning aplicadas al robot Turtlebot," Alcalá de Henares, 2025.

[13] S. IBRAHIM, "Comprehensive Overview of Reward Engineering and Shaping in Advancing Reinforcement Learning Applications," Moscow, 2024.

[14] W. Van Heeswijk, "Towards Data Science. Deep Deterministic Policy Gradients (DDPG) Explained," 5 Abril 2023. [Online]. Available: https://towardsdatascience.com/deep-deterministic-policy-gradients-explained-4643c1f71b2e/.

[15] A. Heilmeier, A. Wischnewski, L. Hermansdorfer, J. Betz, M. Lienkamp and B. Lohmann, "Minimum curvature trajectory planning and control for an autonomous race car," *Vehicle System Dynamics,* 2020.

[16] M. E. Tipping, M. A. Hatton and R. Herbrich, "Racing Line Optimization". Estados Unidos Patent US 7,892,078 B2, 2011.

[17] J. Wallace, B. Mitchell, E. Diaz and L. Bircumshaw, "The Racing Line," 2019. [Online]. Available: https://physicsofformula1.wordpress.com/the-racing-line/.

- Contenido de la asignatura Sistemas Control para Robots

Universidad de Alcalá

2026