

# PROYECTO CONTROL Y PROGRAMACIÓN DE ROBOTS

---

Control y programación de  
un robot diferencial usando el  
algoritmo “Pure Pursuit”

---

Autores:  
Francisco Javier Román Cortés  
Rohan Keith Laycock  
Juan de Dios Herrera Hurtado

# ÍNDICE

## 1.- INTRODUCCIÓN

- 1.1.- PLANIFICADOR
- 1.2.- INTERPOLADOR
- 1.3.- CONTROL PURE PURSUIT
- 1.4.- FUNCIONAMIENTO GLOBAL DEL PROGRAMA

## 2.- ARCHIVOS

## 3.- PLANIFICADOR

- 3.1.- FUNCIONAMIENTO
- 3.2.- MODIFICACIONES

## 4.- INTERPOLADOR

- 4.1.- INTERPOLADOR: BLOQUE FUNCIONAL DENTRO DEL NODO PLANIFICADOR
- 4.2.- REPLANIFICADOR: BLOQUE FUNCIONAL DENTRO DEL NODO PLANIFICADOR
  - 4.2.1.- PERCEPCIÓN
  - 4.2.2.- DETECCIÓN
  - 4.2.3.- TRANSFORMACIONES
  - 4.2.4.- EVITACIÓN
  - 4.2.5.- HIPÓTESIS DE PARTIDA REPLANIFICADOR
  - 4.2.6.- DIMENSIONES DEL ROBOT
  - 4.2.7.- RANGO LÁSER
  - 4.2.8.- DIMENSIONES DEL OBSTÁCULO
  - 4.2.9.- MODO 1
  - 4.2.10.- MODO 2 Y 3
  - 4.2.11.- GESTIÓN DE LA INFORMACIÓN DE LOS WAYPOINTS LOCALES
  - 4.2.12.- FUNCIONES DESARROLLADAS Y EMPLEADAS
  - 4.2.13.- PARÁMETROS DEL PLANIFICADOR Y DEL REPLANIFICADOR

## **5.- ALGORITMO PURE PURSUIT PARA SEGUIMIENTO DE TRAYECTORIAS**

- 5.1.- INTRODUCCIÓN
- 5.2.- DESARROLLO
- 5.3.- ALGORITMO DE SEGUIMIENTO DE WAYPOINTS
- 5.4.- NODO “PURE\_PURSUIT.PY”
- 5.5.- ANÁLISIS DEL PARÁMETRO “L”
- 5.6.- INFLUENCIA DE LA VELOCIDAD DEL ROBOT EN EL CONTROL
- 5.7.- CONTROL ADAPTATIVO DE “L” EN EL REPLANIFICADOR
- 5.8.- DISEÑO ERRÓNEO DEL PURE PURSUIT ¿UNA VARIACIÓN MEJOR?
- 5.9.- LIMITACIÓN DEL PURE PURSUIT EN EL FINAL DE LA TRAYECTORIA SEGÚN “L”
- 5.10.- EXPERIMENTOS CON DIFERENTES L Y OTROS PARÁMETROS
- 5.11.- COMPARACIÓN PURE PURSUIT – PSEUDO PID

## **6.- EXPERIMENTOS EN VARIOS MAPAS**

- 6.1.- MAPA ALTERNATIVO 1
  - 6.1.1.- TEST 1
  - 6.1.2.- TEST 2
- 6.2.- MAPA ALTERNATIVO 2
  - 6.2.1.- TEST 1
  - 6.2.2.- TEST 2

## **7.- RESULTADOS GRÁFICOS DEL CONTROLADOR Y PLANIFICADOR**

- 7.1.- EXPERIMENTO SIN OBSTÁCULO
- 7.2.- EXPERIMENTO CON OBSTÁCULO

## **8.- DIFICULTADES PRESENTADAS A LO LARGO DEL PROYECTO**

- 8.1.- ROS
- 8.2.- GESTIONAR TIPOS DE MENSAJES
- 8.3.- TRANSFORMACIONES
- 8.4.- VISUALIZACIONES
  - 8.4.1.- PRIMERAS FASES DE IMPLEMENTACIÓN DEL PLANIFICADOR E INTERPOLADOR
  - 8.4.2.- FASES DE DESARROLLO DEL REPLANIFICADOR

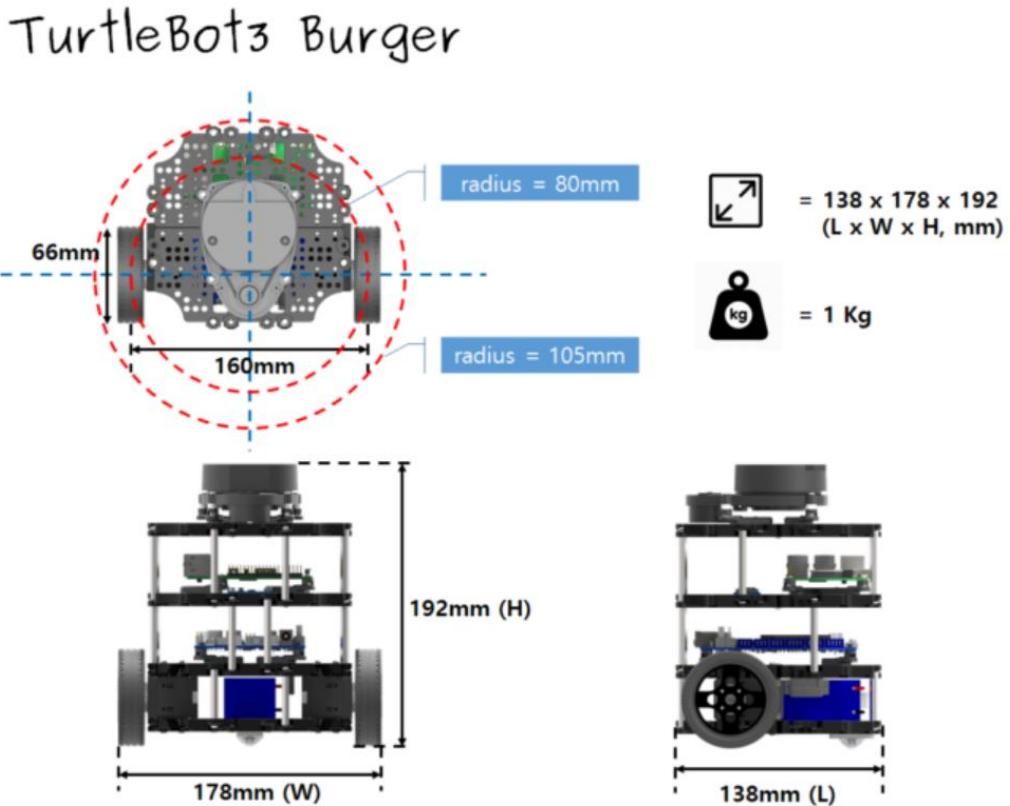
## **9.- CONCLUSIÓN**

## **10.- REFERENCIAS**

## 1.- Introducción

El proyecto a desarrollar consiste en realizar el control de un robot diferencial mediante el método de control de persecución pura (“pure pursuit”).

El modelo seleccionado para llevar a cabo el proyecto es el “TurtleBot3” en su versión “Burger”. Ahora se muestran imágenes del mismo:



*Figura 1.1: Dimensiones y forma del modelo TurtleBot3 Burger*

El proyecto se puede dividir en tres bloques fundamentales, cada uno implementa una funcionalidad necesaria para que el robot realice la trayectoria correctamente. A continuación, se explican los diferentes bloques.

**1.1.- Planificador:** implementa el algoritmo “Lazy Theta\*”. Este bloque se encarga de, dado un punto origen (donde se encuentra el robot), un punto destino, el mapa de costes del mapa junto con un archivo .pgm que describe el mapa en escala de grises y un archivo .yaml que configura diferentes parámetros del planificador, suministrar una serie de “waypoints” que conducirán al robot hasta el punto destino, evitando la colisión con cualquier obstáculo que se encuentre en el mapa.

**1.2.- Interpolador:** el planificador suministra puntos para llegar a la trayectoria, pero a veces estos puntos pueden estar muy distanciados los unos de los otros como sucede por ejemplo en un tramo recto, donde tendremos un “waypoint” al principio de la recta y otro al final de la recta pudiendo ser esta recta muy larga.

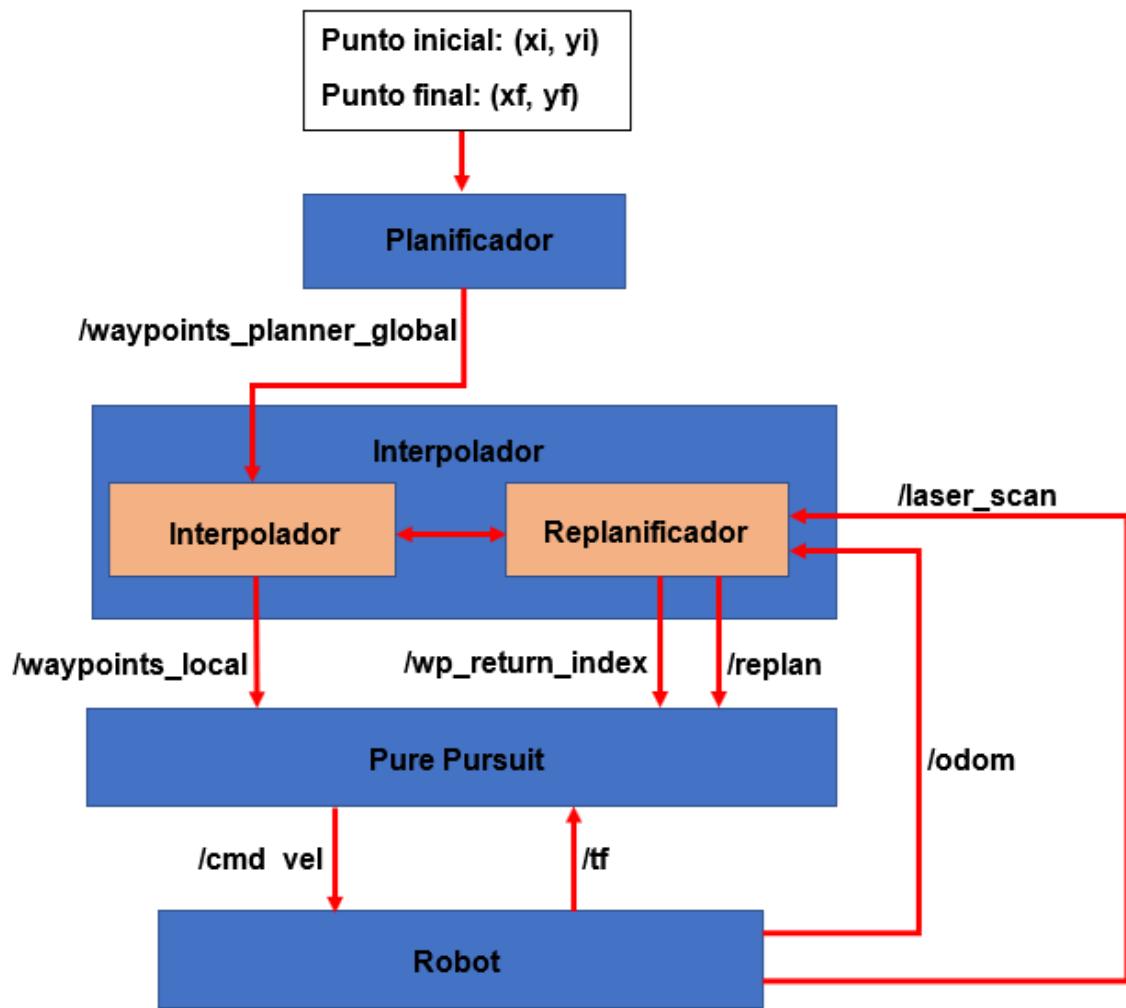
Este bloque realiza interpolaciones lineales o cúbicas (“cubic spline”), cuyo objetivo es crear puntos intermedios entre los distintos “waypoints” recibidos desde el planificador, aportando más datos al robot sobre la trayectoria. Es por ello que este bloque recibe los “waypoints” procedentes del planificador.

Por último, este bloque también implementa una funcionalidad de replanificación que otorga al robot la capacidad de detectar y evitar un obstáculo que no estuviese contemplado en el mapa inicialmente, para ello, este bloque recibe también la información procedente del lidar de nuestro robot.

**1.3.- Control Pure Pursuit:** llegamos al bloque que se encarga de que nuestro robot siga la trayectoria correctamente. Teniendo los “waypoints” del planificador, lo puntos intermedios creados por el interpolador y suponiendo velocidad lineal constante (en nuestro caso es también este bloque el que publica esta información), actúa sobre la velocidad angular del robot de manera que este siga a la trayectoria de manera correcta.

Adicionalmente, en este bloque se hará una comparación entre este tipo de control y un controlador tipo PID (proporcional + integral + derivativo)

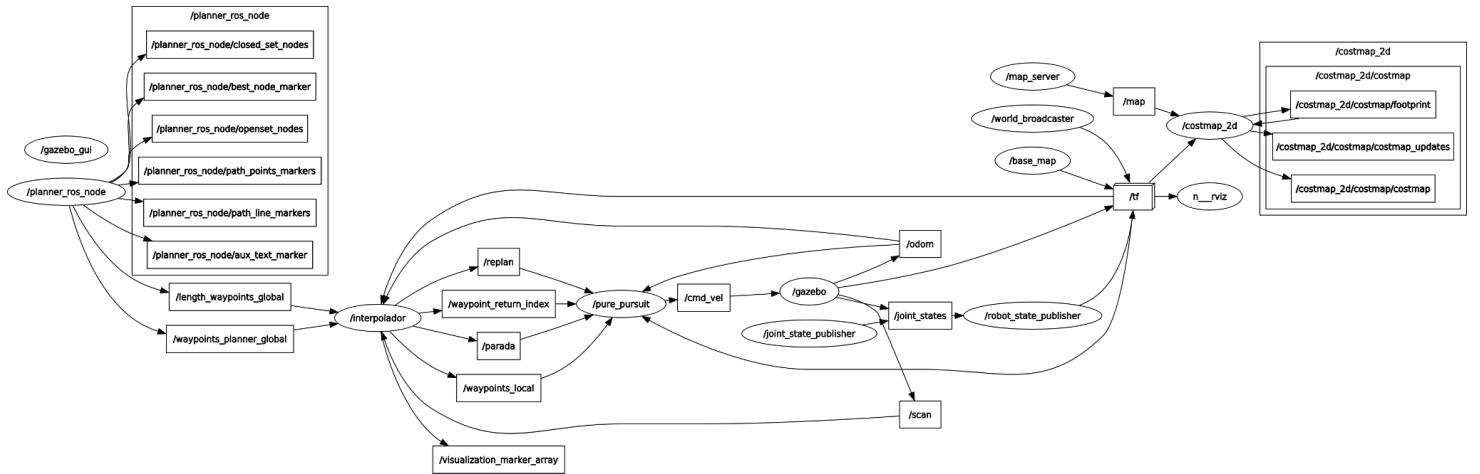
El esquema global de los bloques anteriormente comentados es el siguiente:



*Figura 1.2: Esquema global del proyecto*

Descripción:

- **/waypoints\_planner\_global**: waypoints globales que envía el planificador al interpolador.
- **/waypoints\_local**: waypoints locales que envía el interpolador al control.
- **/wp\_return\_index**: índice “i” que indica en qué waypoint local (wp\_local\_msg [i]) vuelve el robot a seguir a la trayectoria original.
- **/replan**: señal booleana que se activa si se entra en modo replanificación y que se le pasa al control.
- **/cmd\_vel**: tópico que envía el control al robot con el fin de indicarle las velocidades necesarias para que siga la trayectoria correctamente.
- **/tf**: paquete que permite realizar transformaciones de coordenadas de un sistema de referencia a otro. Se usa para transformar los waypoints locales al sistema de referencia solidario con del robot.
- **/odom**: tópico en el que publica el robot su posición actual con respecto al origen del mapa y que recibe el replanificador.
- **/laser\_scan**: tópico donde se recoge la información percibida por el lidar y que se pasa al replanificador.



**Figura 1.3:** Esquema de nodos y topics del sistema completo

#### 1.4.- Funcionamiento global del programa:

- En un experimento, se le indicará un punto inicial ( $x_i, y_i$ ) y un punto final ( $x_f, y_f$ ).
- Al inicio del programa se genera la trayectoria inicial y esta lo almacena el controlador.
- Primero, el nodo planificador calcula los waypoints globales de la trayectoria según el algoritmo implementado, y se lo envía al nodo interpolador. Después el nodo interpolador interpola a partir de los waypoints globales y genera un conjunto de waypoints locales que se envían al nodo del controlador pure pursuit.
- Finalmente, dentro del controlador pure pursuit, existe un control de waypoints donde se suministra waypoint por waypoint según un criterio implementado. En cada instante existirá una referencia de un waypoint y se aplica un control mediante el algoritmo pure pursuit.
- Mientras el robot recorre la trayectoria inicial entra en un modo replanificar, donde escanea constantemente su entorno, para detectar si un obstáculo se interpone sobre la trayectoria inicial, y evitarlo. Después de evitar el obstáculo, el robot retoma la trayectoria inicial, y sigue en este modo de detección hasta que alcance el punto final.

## 2.- Archivos del proyecto

Se ha creado un paquete personalizado de ROS llamado “control”.

Nodos:

- interpolador.py
- pure\_pursuit.py
- grafica.py

Estos vienen incluidos en el CMakeLists.txt

Launch:

El archivo launch llama a varios paquetes, como:

- tf
- diff\_drive
- Heuristic\_path\_planners
- map2gazebo

También se ha creado un archivo que contiene la configuración (valores) de parámetros de los nodos anteriormente mencionados:

- parametros.yaml

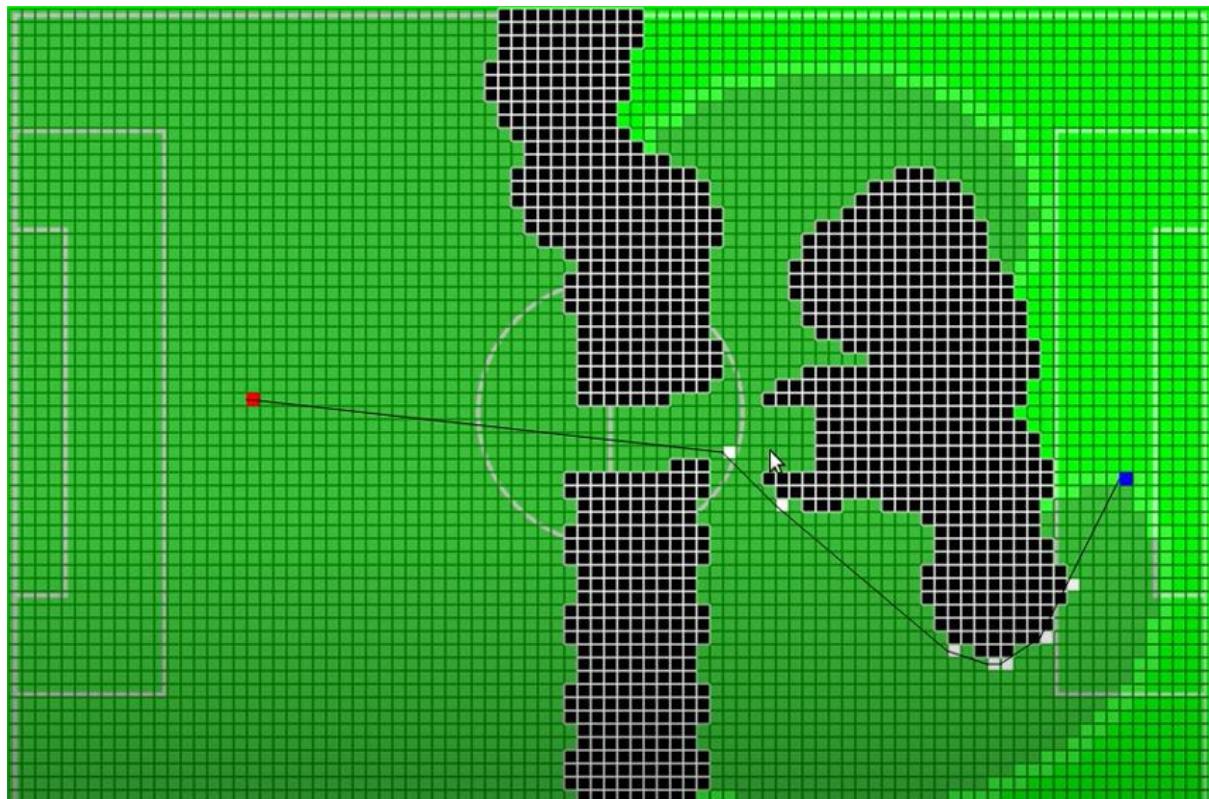
### 3.- Planificador

Como ya se ha comentado, el planificador se encarga de crear una trayectoria que permita a nuestro robot alcanzar el punto deseado evitando la colisión con las paredes y obstáculos que se encuentren en un mapa.

Este bloque no se ha desarrollado, sino que se ha implementado el algoritmo ya desarrollado genéricamente, y se ha adaptado para poder usarlo en nuestro proyecto. El paquete permite implementar 5 tipos de planificadores, pero nosotros usaremos el algoritmo “Lazy Theta \*”. El enlace al paquete es el siguiente:

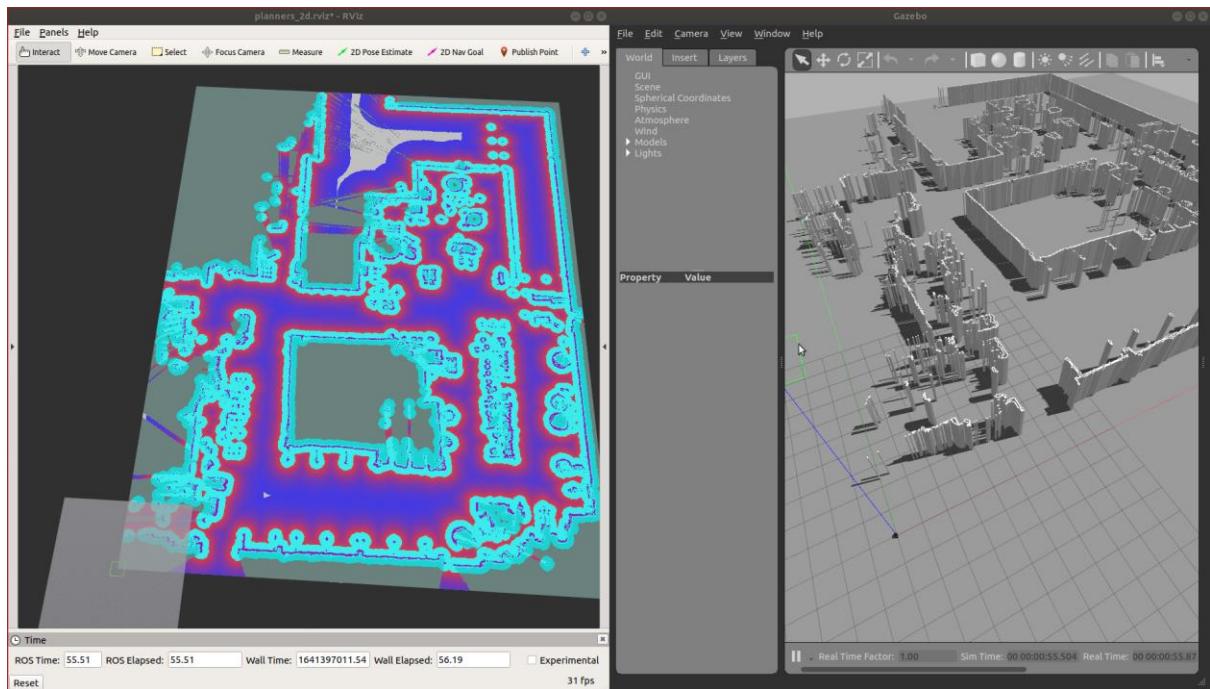
[https://github.com/robotics-upo/Heuristic\\_path\\_planners](https://github.com/robotics-upo/Heuristic_path_planners)

#### **3.1.- Funcionamiento:**



*Figura 3.1: Ejemplo de generación de trayectoria del algoritmo Lazy Theta \**

- A diferencia del algoritmo A\* que solo usa la información del mapa de costes, es un algoritmo de búsqueda de trayectorias en cualquier ángulo existente, es decir, usa tanto las cuadrículas del mapa de costes como direcciones.



**Figura 3.2:** Visualización del mapa de costes en RViz correspondiente al mapa disponible con el que trabaja el planificador

- El algoritmo se basa en realizar una sucesión de trayectorias rectas hasta lograr alcanzar el punto destino. Es en los sitios en los que el robot debe realizar alguna trayectoria curva, donde el planificador cambia la pendiente de la recta hasta que nuevamente haga falta modificar dicha pendiente.
- A la hora de configurar el comportamiento del planificador, principalmente en cuanto al margen de separación que garantiza en la trayectoria respecto de los obstáculos conocidos, se dispone de un archivo “.yaml”. Para garantizar que la trayectoria generada no implicaba acercarse peligrosamente a paredes u obstáculos se ha delimitado el “footprint” o “bounding box del robot” a un cuadrado en el plano de 0.5 m de lado, prácticamente el doble que las dimensiones reales del robot, así como se necesitó ajustar el valor del parámetro “inflation\_radius”. Con ambos cambios, se logran unas trayectorias generadas más seguras para nuestro robot Turtlebot3.

```

Abrir ▾ 例題
costmap:
    global_frame: "map"
    robot_base_frame: "world"
    map_topic: /map

    subscribe_to_updates: false
    always_send_full_costmap: true
    rolling_window: false

    transform_tolerance: 1.0
    resolution: 0.05
#Hay que cambiar footprint PARA TURTLEBOT3 y además, ajustar inflation_radius (MUY IMPORTANTE)
    robot_radius: 0.20
    footprint: [[0.5,0.5], [0.5,-0.5], [-0.5, -0.5], [-0.5, 0.5]]
    footprint_padding: 0.05
    inflation_layer:
        cost_scaling_factor: 1.2
        enabled: true
        inflate_unknown: false
        inflation_radius: 3.0

    static_layer:
        unknown_cost_value: -1
        lethal_cost_threshold: 100

plugins:
    - {name: static_layer, type: "costmap_2d::StaticLayer"}
    - {name: inflation_layer, type: 'costmap_2d::InflationLayer'}

```

**Figura 3.3:** Parámetros de configuración del planificador para modificar las características de las trayectorias que genera

- Es importante mencionar que el planificador sólo almacena como “waypoints” aquellos puntos donde se produce un cambio de pendiente, debido a esta forma de trabajar, si por ejemplo tenemos una trayectoria recta muy larga, es decir, no se producen cambios en la pendiente y es constante, tan solo tendremos un waypoint al comienzo de la recta y otro al final, pero ninguno intermedio a pesar de que el robot tenga que recorrer mucha distancia del mapa.

Para poder usar este paquete, se han realizado ciertas modificaciones.

### 3.2.- Modificaciones:

- Se publica en tres topics adicionales que son:

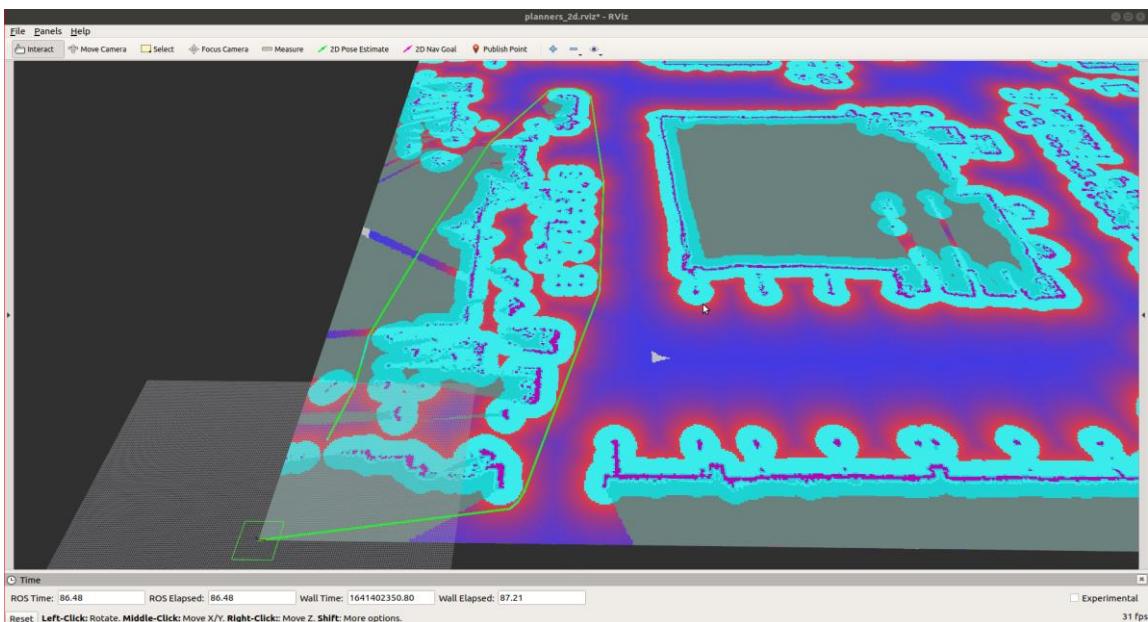
- Un primer topic para publicar los “waypoints” en formato PoseStamped (este topic publica todos los puntos generados al principio del programa, no los va dando poco a poco conforme se avanza en la trayectoria).
- Otro topic que publica el número de “waypoints” creados por el planificador.
- En un principio se creó un tercer topic en el que se publicaban los “waypoints” en formato nav\_msgs, pero el controlador PID que se estaba usando al principio, no admitía dicho formato para funcionar correctamente, por lo que se recurrió al primer topic ya comentado.

- Un problema que presentaba este paquete, era que el vector original de “waypoints” se almacenaba al revés, es decir, la primera componente del vector era el punto destino, y la última era el punto de partida del robot. Es por eso que se tuvo que crear otro vector con los puntos ordenados. Este vector es el que se publica en el primer topic.

Las modificaciones del código se encuentran en el archivo Heuristic\_path\_planners/src/ROS/planner\_ros\_node.cpp entre las líneas 244 y 315.

Se muestra ahora un ejemplo de generación de trayectoria con este planificador tal que:

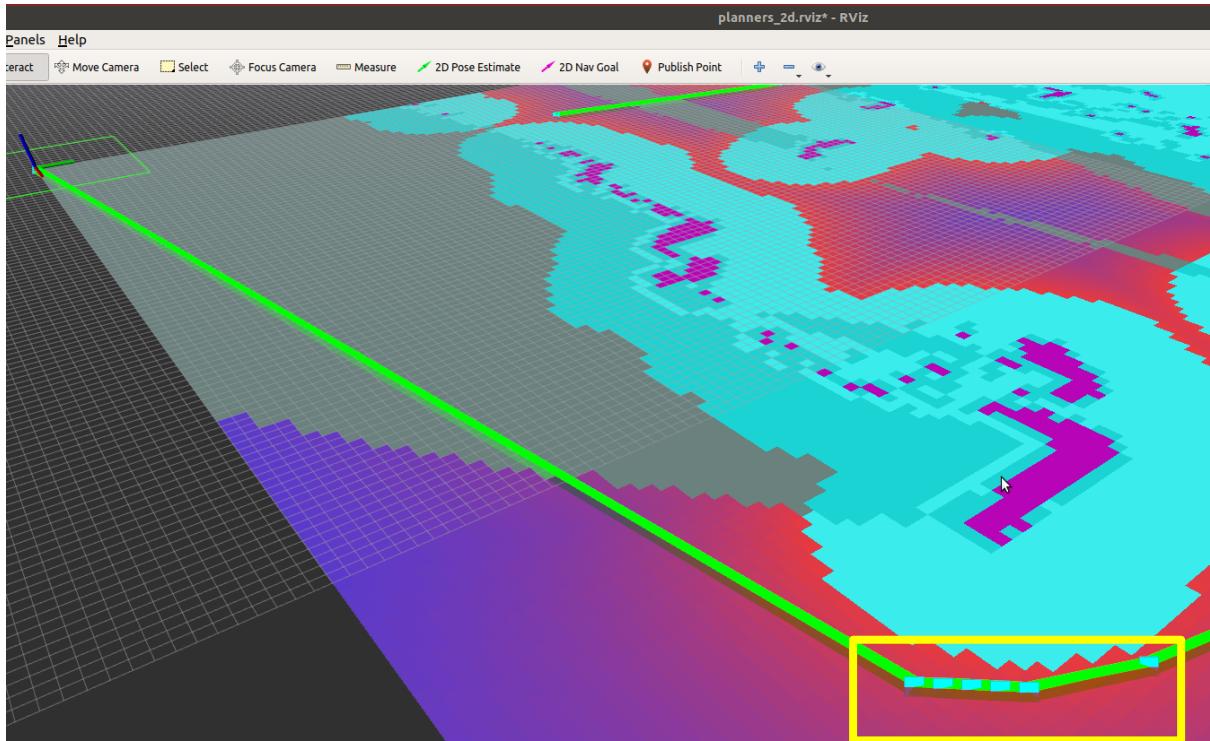
- Punto origen:  $x = 0.0, y = 0.0, z = 0.0$  [m]
- Punto destino:  $x = 1.0, y = 3.0, z = 0.0$  [m]



**Figura 3.4:** Generación de la trayectoria solicitada por parte del planificador

Hay que recalcar que, aunque se visualice una trayectoria continua, este planificador sólo proporciona puntos discretos (con sus coordenadas publicándose en un topic) en los puntos de cambio de pendiente.

Se puede ver en la figura a continuación:



**Figura 3.5:** Detalle de la primera curva (cambio de pendiente)

Vemos cómo sólo los puntos visualizados en celeste (dentro del recuadro amarillo) serían aquellos cuya información se publica y, por tanto, los únicos accesibles para trabajar con ellos a la hora de plantear la trayectoria generada.

Esto plantea la necesidad de realizar interpolación entre waypoints para disponer de una trayectoria más homogénea compuesta por los waypoints obtenidos del bloque interpolador a partir de estos escasos waypoints que nos proporciona el planificador.

## 4.- Interpolador

El nodo interpolador, es un bloque intermedio entre el planificador y el controlador pure pursuit. Se compone de dos bloques funcionales: el interpolador del planificador y el replanificador. Se han definido waypoints globales como los puntos calculados por el planificador, y los waypoints locales los puntos interpolados a partir de los globales. El objetivo principal de este nodo, tanto en el modo planificador como replanificador, es enviar waypoints locales al controlador pure pursuit. Estos se envían por el mismo tópico /waypoints\_local. También publica otros dos tópicos, que son de interés en el modo replanificador, estas son la señal /replan, una señal booleana que indica si está en el modo replanificador, y /return\_index, el índice de retorno de la trayectoria inicial del vector de waypoints locales.

### **4.1.- Interpolador: bloque funcional dentro del nodo interpolador**

En ambos modos, a partir de un conjunto de puntos se interpola y se genera un conjunto mucho mayor de puntos (waypoints locales). Este cálculo se realiza en la función `interpolar()`.

#### Función Interpolar:

Entradas:

- waypoints\_x, waypoints\_y : listas de la componente “x” e “y” de los waypoints globales.
- Publicar: un parámetro booleano que si está activo (a ‘1’) se publicará por el tópico los waypoints locales calculados, en caso contrario no los publica.
- wp\_x, wp\_y: listas de la componente “x” e “y” de los waypoints locales.

Para interpolar, es necesario como mínimo dos waypoints globales. Se ha implementado un interpolador que interpola una línea recta con 2 puntos o una línea curva con 3 puntos si es posible.

Existen algunos casos restrictivos que hace que no sea posible la interpolación con tres puntos. Dados tres puntos P1(x1,y1), P2(x2,y2) y P3(x3,y3), la condición necesaria para obtener una función  $y=f(x)$  se debe cumplir que  $x_1 < x_2 < x_3$  ó  $x_1 > x_2 > x_3$ . Es decir, debe existir una monotonía creciente o decreciente, en caso contrario, será imposible obtener una función  $y=f(x)$ .

Análogamente, para obtener una función  $x=f(y)$  debe existir una monotonía creciente o decreciente sobre la variable “y”, se debe cumplir que  $y_1 < y_2 < y_3$  ó  $y_1 > y_2 > y_3$ .

Finalmente, la interpolación en sí se realiza utilizando la función `CubicSpline` de la librería de numpy.

La función interpolar maximiza la interpolación con tres puntos, y siempre lo hará cuando sea posible, para ello se comprueba primero las condiciones tanto para interpolar en “x” como en “y”. En caso de que no cumpliese ninguna de las condiciones, se recurre a una interpolación con 2 puntos.

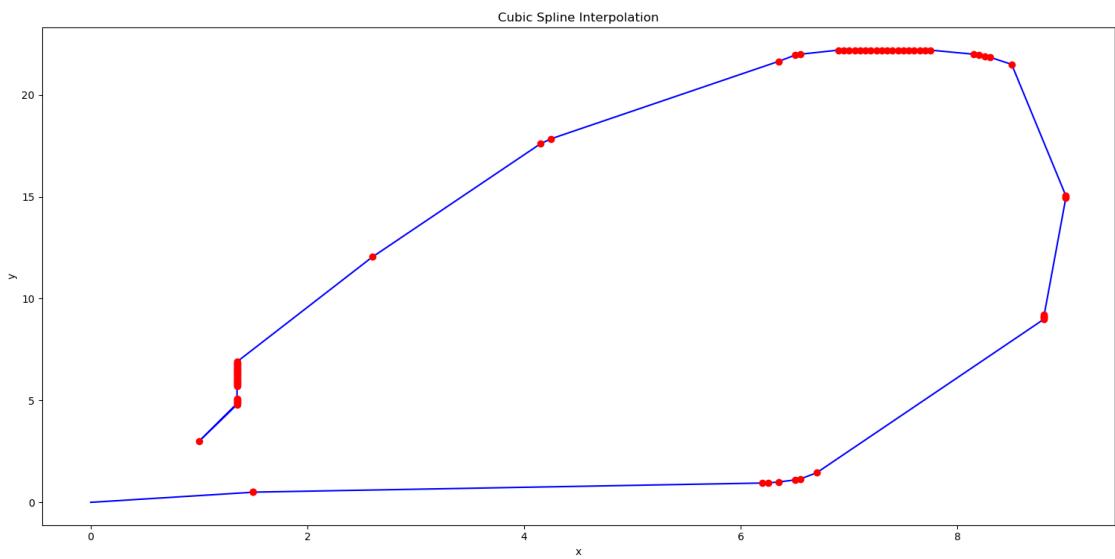
Esto se ha impuesto así porque experimentalmente se ha observado que el seguimiento de la trayectoria del robot mejora en el caso de que la curva sea suave. Se ha conseguido que sea suave por partes, de manera que *el cambio de pendiente en las discontinuidades de los tramos sea el cambio más pequeño posible*. El cambio de pendiente en dos rectas a trozos puede ser mucho mayor.

Queremos evitar cambios bruscos en la referencia al controlador, ya que es pedirle mayor accionamiento en los motores (reduciendo su vida útil) y además, es posible causar que el sistema se vuelva inestable.

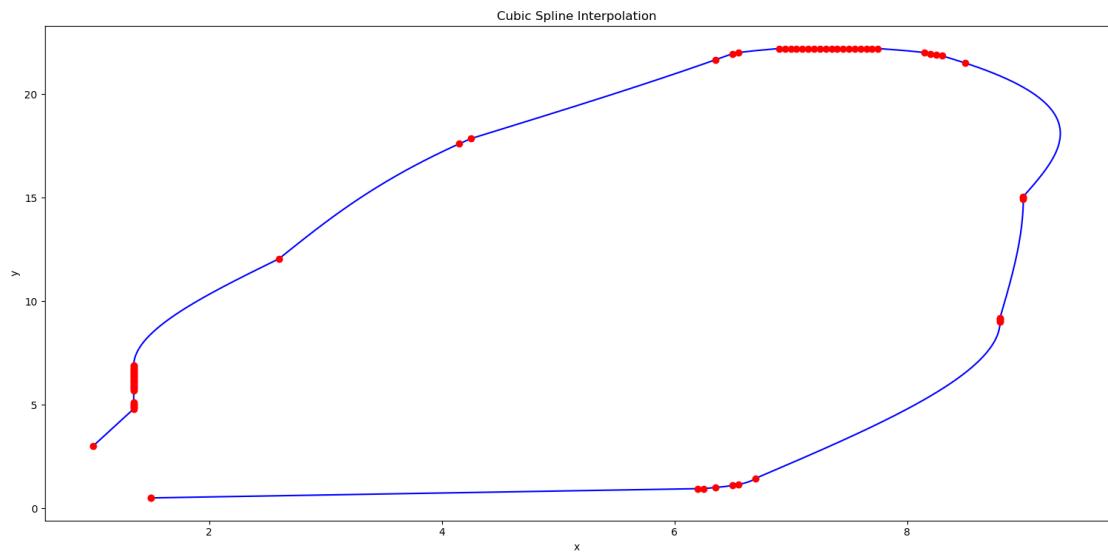
A pesar de maximizar la interpolación con 3 puntos, existe una limitación a esta función, y es que no se elige el conjunto óptimo de puntos para minimizar el cambio en la pendiente. Por ejemplo, dado un rango de puntos monótonos crecientes o decrecientes, la interpolación de tres puntos siempre se hará con los primeros tres que cumpla la condición, aunque podría ser mejor el segundo, tercero y cuarto.

La mejor forma de minimizar el cambio en la pendiente habría sido interpolar con más puntos, pero debido a los pocos casos en los que interpola con tres puntos, y del buen funcionamiento que ya se ha obtenido, decidimos no expandir a más puntos, aunque solo requeriría extender código ya desarrollado.

La razón por la que no interpole con tres puntos en muchos de los casos se debe a la forma en los que el planificador calcula los waypoints globales debido al propio algoritmo de planificación anteriormente comentado. Estos waypoints globales están muy concentrados en lugares donde hay esquinas y en tramos rectos sólo se proporciona los dos puntos que los une (pudiendo quedar así tramos “vacíos” de waypoints entre dos waypoints globales muy alejados).



**Figura 4.1:** Resultados de la interpolación para el caso de interpolación entre parejas de waypoints (en rojo los waypoints globales dados por el planificador y en azul la unión continua del conjunto de waypoints locales obtenidos al interpolar). (Eje X: Distancia en m / Eje Y: Distancia en m)



**Figura 4.2:** Resultados de la interpolación para el caso de interpolación entre 3 waypoints globales (en rojo los waypoints globales dados por el planificador y en azul la unión continua del conjunto de waypoints locales obtenidos al interpolar). (Eje X: Distancia en m / Eje Y: Distancia en m)

Podemos comprobar haciendo “rostopic echo /waypoints\_local” como se genera un vector con los waypoints locales interpolados además ordenados de principio a fin, describiendo la trayectoria completa.

```
---  
header:  
  seq: 2393  
  stamp:  
    secs: 97  
    nsecs: 997000000  
  frame_id: "base_link"  
pose:  
  position:  
    x: 2.7177284136  
    y: 12.5801940933  
    z: 0.0  
  orientation:  
    x: 0.0  
    y: 0.0  
    z: 0.628028239824  
    w: 0.778190548634  
---  
header:  
  seq: 2394  
  stamp:  
    secs: 97  
    nsecs: 997000000  
  frame_id: "base_link"  
pose:  
  position:  
    x: 2.71414471163  
    y: 12.564127611  
    z: 0.0  
  orientation:  
    x: 0.0  
    y: 0.0  
    z: 0.628031578008  
    w: 0.778187854585  
---  
header:  
  seq: 2395  
  stamp:  
    secs: 97  
    nsecs: 997000000  
  frame_id: "base_link"  
pose:  
  position:  
    x: 2.71056250278  
    y: 12.5480611288  
    z: 0.0  
  orientation:  
    x: 0.0  
    y: 0.0  
    z: 0.628034880507  
    w: 0.778185189313  
---  
header:  
  seq: 2978  
  stamp:  
    secs: 98  
    nsecs: 49000000  
  frame_id: "base_link"  
pose:  
  position:  
    x: 1.00642203335  
    y: 3.03302756813  
    z: 0.0  
  orientation:  
    x: 0.0  
    y: 0.0  
    z: 0.585262515186  
    w: 0.810843874194  
---  
header:  
  seq: 2979  
  stamp:  
    secs: 98  
    nsecs: 49000000  
  frame_id: "base_link"  
pose:  
  position:  
    x: 1.00321102412  
    y: 3.01651380642  
    z: 0.0  
  orientation:  
    x: 0.0  
    y: 0.0  
    z: 0.584987882881  
    w: 0.811042031514  
---  
header:  
  seq: 2980  
  stamp:  
    secs: 98  
    nsecs: 49000000  
  frame_id: "base_link"  
pose:  
  position:  
    x: 1.0000000149  
    y: 3.0000000447  
    z: 0.0  
  orientation:  
    x: 0.0  
    y: 0.0  
    z: 0.584710284664  
    w: 0.811242185176  
---
```

**Figura 4.3:** Fragmentos extraídos del topic /waypoints\_local, que publica todos los waypoints interpolados en formato PoseStamped, desde el principio de la trayectoria hasta su final

## **4.2.- Replanificador: bloque funcional dentro del nodo interpolador**

Una vez generado los waypoints locales del planificador, se ejecuta la función **SenseAndAvoid**, donde se escanea el entorno constantemente para comprobar si existe algún obstáculo que obstruye la trayectoria inicial. El programa sigue en esta función, en modo detección, hasta que el robot alcance el punto final.

### 4.2.1.- Percepción

Para percibir el entorno se ha utilizado un sensor LIDAR (360 Laser Distance Sensor LDS-01) que venía integrado con el modelo del turtlebot3, es un sensor que permite determinar la distancia desde un emisor láser a un objeto o superficie utilizando un haz láser pulsado [1]. De esta manera logramos un barrido 2D del entorno en todas sus direcciones, pudiendo obtener la distancia de cualquier obstáculo existente al rango que permite detectar el sensor (3.5 m como máximo). Sin embargo, aunque el lidar ofrezca medidas en todas las direcciones, puede ser útil usar sólo un cierto rango de detección para ciertas aplicaciones, como veremos a la hora de implementar la tarea de replanificación entorno a un obstáculo, donde se usan únicamente barridos en cierto rango de ángulos convenientes.

Al incluir el archivo urdf del modelo del robot, que contiene todas las características físicas del mismo, se facilita a Gazebo/ROS la comunicación de los datos adquiridos por los sensores del robot. Por tanto, al suscribirnos al tópico /laser\_scan se obtienen las medidas del LIDAR 360 disponible. De este mensaje accedemos al vector ranges[], que en nuestro caso es un vector de tamaño 360, es decir se obtienen las medidas en todas las direcciones del robot, y de una medida a otra se produce un incremento de un grado, resultando tener una relación intuitiva entre el índice del vector y el ángulo, que se aprovecha en su uso. Este sensor tiene una limitación de un alcance máximo de 3.5 metros. Una medida superior a esta se corresponde con una medida inf (infinito). Esto último es importante, ya que ha sido necesario tratar esta consideración del infinito a la hora de manejar la información procedente el LIDAR para implementar el algoritmo del replanificador.



Ambas transformaciones se realizan en la función **`laser_scan_vector_to_world_coordinates()`**. Esta función devuelve `obstaculos_x [ ]` y `obstaculos_y [ ]`, listas de las componentes x e y de las coordenadas de los obstáculos referidas al mundo. Luego iterando se comprueba si alguna coordenada del “`laser_scan`” coincide con alguno de los waypoints que componen la trayectoria original.

#### 4.2.4.- Evitación

Si se detecta un punto que obstruye la trayectoria inicial, se llama a la función **`replanificar()`**, esta se encarga de evitar el obstáculo, divergiendo de la trayectoria inicial, rodeando el obstáculo y replanificando una nueva trayectoria hasta poder retomar la inicial. Para ello, se ha desarrollado un algoritmo general constituido por tres etapas que denominamos modos.

La idea general del replanificador es construir una trayectoria lineal a trozos sobre la marcha. En cada iteración recorre un tramo, generada a partir de dos waypoints globales, y cuando finaliza el tramo genera otro tramo, hasta volver a la trayectoria inicial.

A la función replanificar, le pasamos como entradas un vector de waypoints reducidos, eliminando todos los waypoints anteriores al punto del obstáculo. De esta manera, reducimos complicaciones en el momento del robot de retomar la trayectoria. Teniendo todos los waypoints, podría intentar retomar la trayectoria en zona ya recorrida. Además, este problema tiene otra implementación que garantiza la seguridad de que eso ocurra, siendo ésta que el robot en modo replanificar, hace escaneos frontales y laterales, restringiendo el rango, que obliga al robot avanzar hacia delante y con giros reducidos (respecto al rango máximo de 360º).

Aunque la razón principal por no escanear por detrás (en el modo 3) fue porque observamos que una vez replanificado la nueva trayectoria y recorrida. Cuando vuelve a la trayectoria inicial y vuelve a entrar en modo detección, detectaba un punto del obstáculo ya evitado, obstruyendo el tramo de la trayectoria no recorrida, y es evidente que esto no es de interés.

#### 4.2.5.- Hipótesis de partida del replanificador

Existen tres variables que influyen en el problema de evitación de obstáculos, y según sus valores determinarán si se puede evitar o no el obstáculo. Las restricciones son geométricas. Con ellas se define un rango de funcionamiento y acotamos el problema.

- Dimensiones del robot
- Rango del láser frontal, parametrizado con `ang_ini` y `ang_fin`
- Dimensiones del obstáculo

#### 4.2.6.- Dimensiones del robot

Las dimensiones del robot son fijas, se introduce un margen respecto de las dimensiones reales del robot, definiendo un “bounding box” que lo delimita de 0.5 x 0.5 (m), esto se vio en el apartado del planificador.

#### 4.2.7.- Rango láser

Reducimos la apertura del escaneo láser a una apertura frontal (para el modo 1), esta apertura está parametrizado con ang\_ini y ang\_fin, y unos valores adecuados son  $45^\circ$  y  $315^\circ$  respectivamente, resultando en una apertura frontal de  $90^\circ$ .

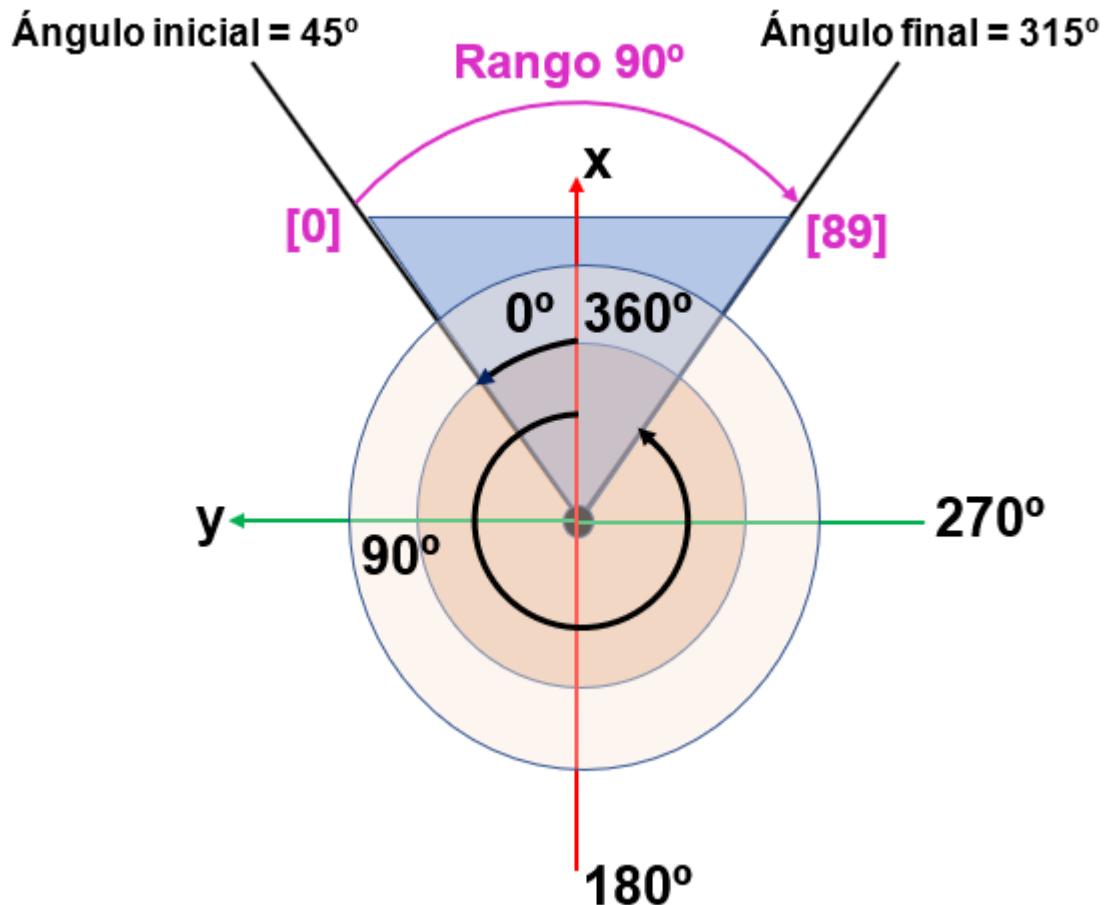


Figura 4.5: Adaptación del rango láser reducido

No conviene que el rango sea el máximo (los  $360^\circ$ ) ya que el siguiente waypoint se elige dentro de este rango, y este ángulo representa cuánto se desviará respecto de la trayectoria inicial. Si el punto estuviera muy lejos del obstáculo, teniendo en cuenta el rango de escaneo posteriormente, el robot podría perder el obstáculo y la trayectoria y divergir completamente, imposibilitando su retorno. Tampoco conviene que el rango sea muy estrecho, ya que necesita una apertura suficiente para rodear el obstáculo. Un término medio entre dichos extremos es una apertura total de  $90^\circ$ .

#### 4.2.8.- Dimensiones del obstáculo

La dimensión de interés es la distancia entre el punto que se interpone sobre la trayectoria y el borde del obstáculo (en el caso de un cubo es la esquina). El valor de esta distancia ( $b$ ) determina si se podrá rodear el obstáculo o no, según los valores de la distancia al obstáculo ( $d$ ) y el rango de apertura ( $2^{\circ}\alpha$ ).

Se ha encontrado una expresión matemática general mediante trigonometría que relaciona las distancias  $b$ ,  $d$ , y el ángulo  $\alpha$ , y las dimensiones del robot.

Para que el robot pueda rodear el obstáculo, se ha impuesto la condición de que el espacio adyacente al obstáculo, definido por la distancia  $e$ , debe ser mayor que la distancia delimitada por el bounding box, que representa la dimensión del robot.

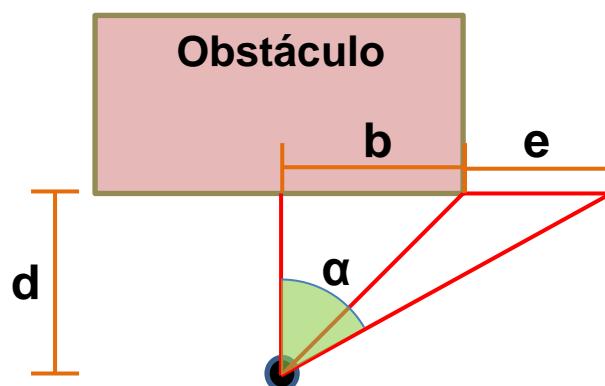


Figura 4.6: Esquema evitación de obstáculo

Aplicando geometría obtenemos el valor de “ $e$ ”:

$$\tan \alpha = \frac{b + e}{d}$$

$$d \cdot \tan \alpha = b + e$$

$$e = d \cdot \tan \alpha - b$$

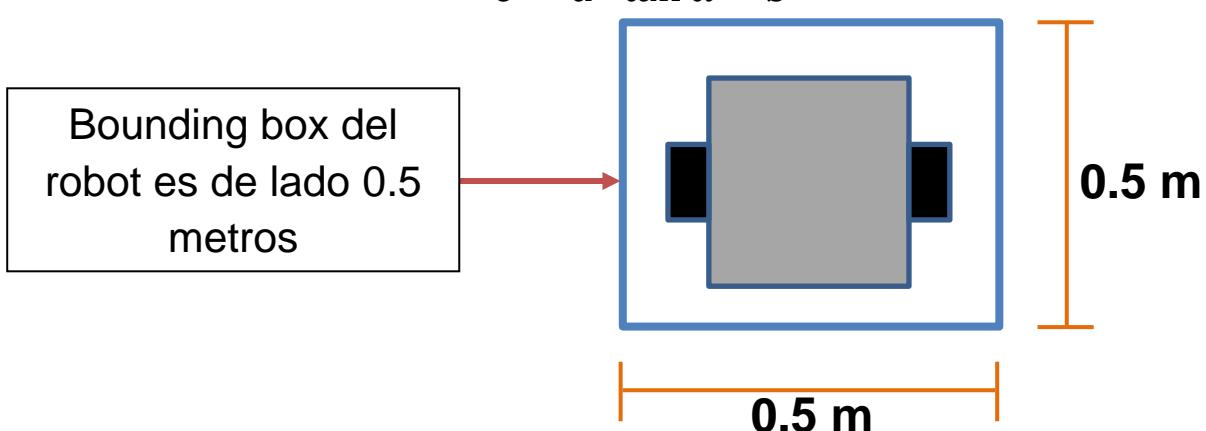


Figura 4.7: Bounding box del TurtleBot3 Burger

Debido al valor que se ha establecido para el bounding box (0.5 metros), imponemos que “e” sea mayor a dicho valor:

$$e > 0.5 \text{ metros}$$

$$e = d \cdot \tan \alpha - b$$

$d \cdot \tan \alpha - b > 0.5 \text{ metros} \rightarrow \text{condición de evitación de obstáculo}$

Aplicado a nuestro caso particular “ $\alpha$ ” es  $45^\circ$ , luego pasamos a sustituir:

$$d \cdot \tan 45^\circ - b > 0.5 \text{ metros}$$

$$d \cdot 1 - b > 0.5 \text{ metros}$$

$$d - b > 0.5 \text{ metros}$$

Y también tenemos que “b” son 0.5 metros, así que volvemos a sustituir:

$$d - 0.5 > 0.5 \text{ metros}$$

$$d > 1 \text{ metro}$$

A la hora de implementar esta condición en el código, se ha añadido una cierta tolerancia resultando:

$$d = 1.3 \text{ metros}$$

#### 4.2.9.- Modo 1

A partir de la odometría, se toma el primer waypoint (waypoint global 1) como la posición actual del robot en el instante en el que se detectó el obstáculo. El obstáculo se detecta a una distancia “d” desde el robot.

Teniendo un vector de las medidas del láser frontal, se busca *el rango continuo de medidas máximas*, que se corresponde con *el espacio con mayor apertura*, y en este rango, buscamos el punto medio. Esta medida nos indica el ángulo de giro del tramo nuevo respecto a la trayectoria, y por tanto lo que el robot girará para situarse sobre él.

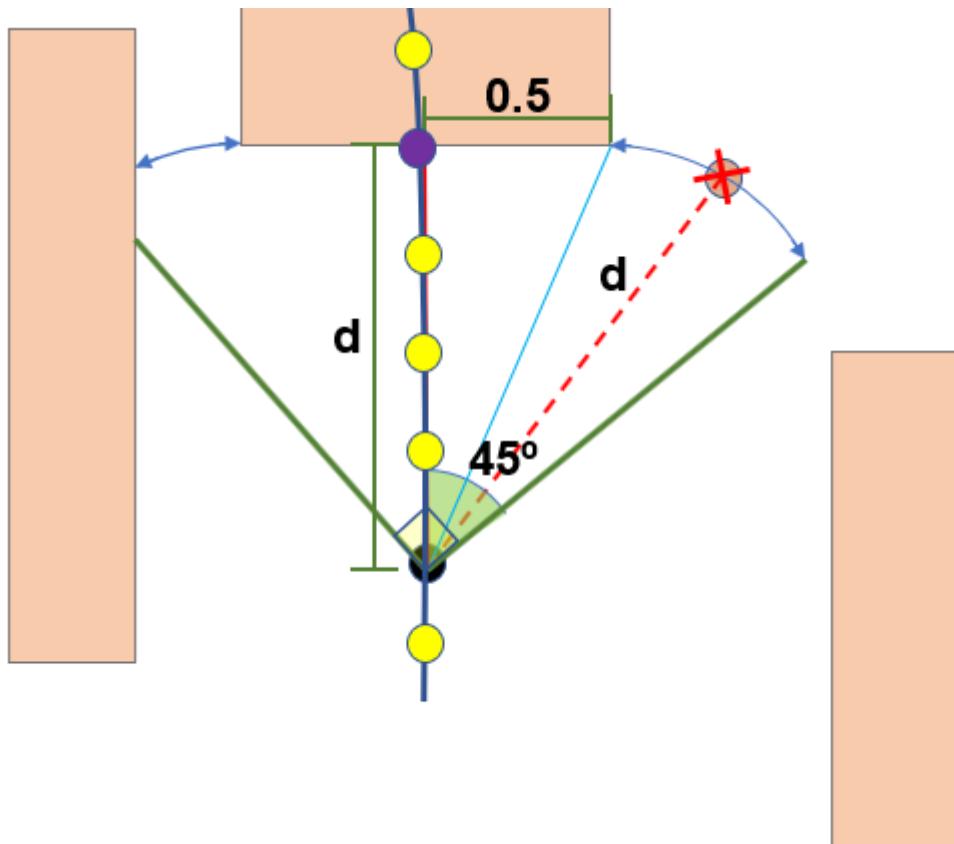
Las medidas del láser frontal fue necesario ordenarlo, porque presenta una discontinuidad en el eje x, y esto imposibilita encontrar un rango continuo de valores máximos. Para ello se empleó la función **ordenar\_vector\_scan\_frontal()**. Una vez ordenado el vector, se busca el punto medio del rango continuo de mayor longitud de elementos de valor máximo, con la función **encuentra\_max()**, esta devuelve el índice del punto buscado referido al vector ordenado, después se convierte el índice referido al vector ordenado al índice referido al vector de medidas totales, de  $360^\circ$ . Este índice equivale al ángulo en sentido antihorario sobre el semieje x positivo del sistema de coordenadas del robot, por tanto, para transformar la medida del láser de

coordenadas a polares a cartesianas, es necesario tener este ángulo. Se convierte mediante la función `indice_a_angulo()`.

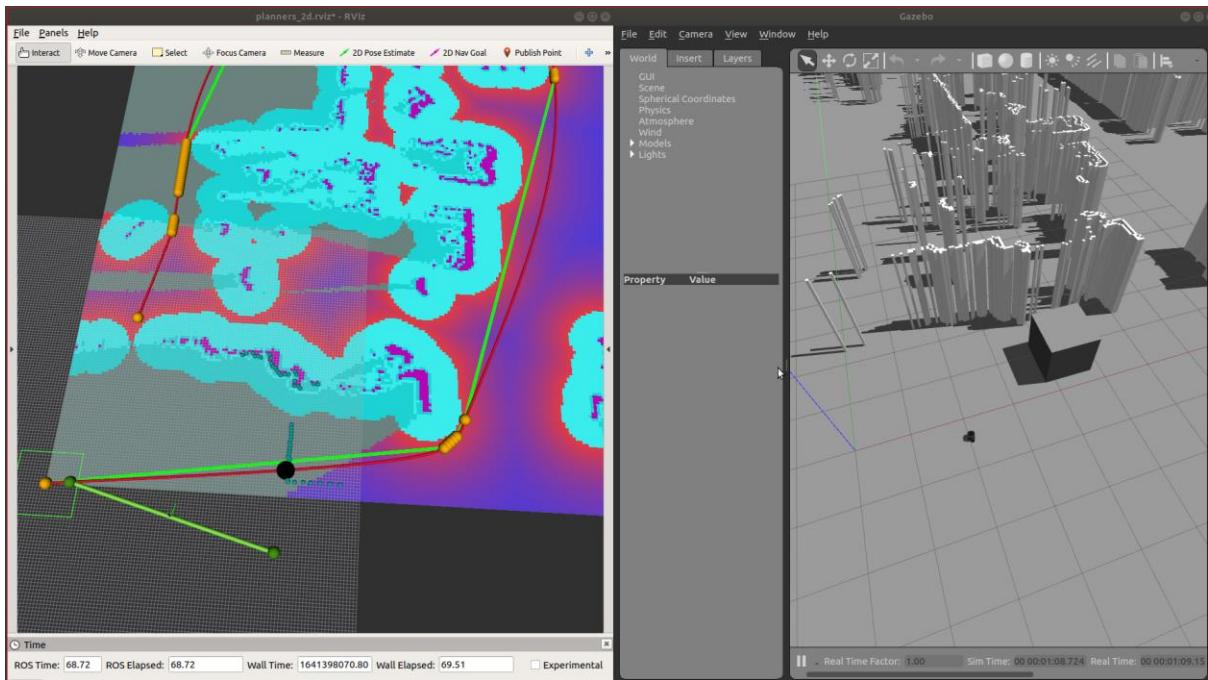
Con la distancia “d” y el ángulo de giro, tenemos una coordenada polar referida al robot. Empleando la función `laser_scan_point_to_world_coordinates()`, se transforma en un punto referido al mundo, y este se corresponde con el waypoint global 2. Teniendo 2 waypoints globales, se interpolan para generar waypoints locales, con la función `interpolar()`, y se publican por el tópico `/waypoints_local`.

Cuando alcanza el waypoint global 2, que se determina con la odometría, se pasa al modo 2.

Este criterio se verificó experimentalmente como el más general, ya que es un criterio que funciona en todos los casos dentro del rango de funcionamiento, independiente de si el obstáculo está cerca o lejos. *Garantiza un alejamiento suficiente del obstáculo para poder evitarlo y rodearlo, pero no diverge demasiado como para perder la trayectoria inicial.*



*Figura 4.8: Esquema del funcionamiento del modo 1*



**Figura 4.9:** Ejemplo de robot realizando el Modo 1 del Replanificador en RViz/Gazebo

#### 4.2.10.- Modos 2 y 3

En el modo 2, se toma el waypoint global 1 como el waypoint global 2 anterior, así se mantiene la continuidad en los tramos lineales a trozos.

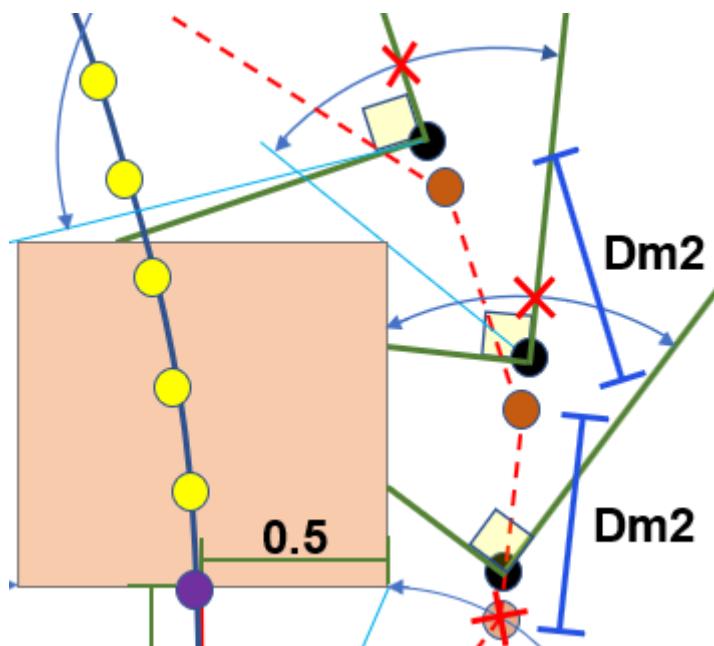
Para el waypoint global 2, según el posicionamiento del robot respecto del obstáculo y la trayectoria inicial, existen dos opciones: seguirá en el modo 2 rodeando el obstáculo o retomará la trayectoria inicial entrando al modo 3. En el modo 2, según si ha evitado el obstáculo por la izquierda o derecha (respecto de la dirección de avance del robot), hará un escaneo lateral hacia la derecha o hacia la izquierda, respectivamente.

Al igual en el modo 1, se busca el punto medio en el rango con mayor apertura. Difiere del modo 1 debido al rango de escaneo del scan laser. De aquí se obtiene el ángulo de giro. Existe un parámetro **longitud\_tramos\_modo\_2**, que fija la longitud de los tramos en el modo 2.

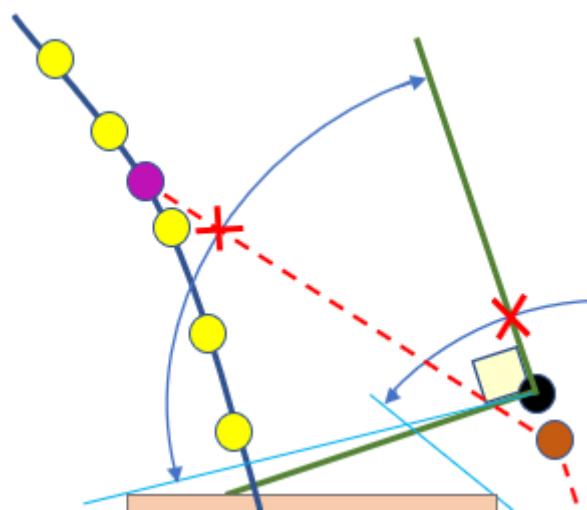
Finalmente, como en el modo 1, empleando la función **laser\_scan\_point\_to\_world\_coordinates()**, con la coordenada polar; el ángulo y la longitud fijada, se transforma en un punto referido al mundo y este se corresponde con el waypoint global 2. Si no entra en el modo 3, con los 2 waypoints globales, se interpolan para generar waypoints locales, con la función **interpolar()**, y se publican por el tópico **/waypoints\_local**. Cuando alcanza el waypoint global 2, se itera otra vez este proceso.

Para comprobar si puede entrar en el modo 3, se interpola el segmento que une el waypoint 1 global actual y el punto medio del rango con mayor apertura y se comprueba si el segmento corta con la trayectoria inicial (viendo si coincide con algún waypoint de la trayectoria original con cierta tolerancia). Si existe un corte, se retomará la trayectoria, siendo el punto de corte el waypoint local de retorno de la trayectoria inicial. Este índice es de vital importancia para que el pure pursuit pueda volver a la trayectoria inicial y se publica por el tópico `/waypoint_return_index`.

En caso contrario, seguirá en el modo 2 hasta poder entrar al modo 3. Es decir, seguirá rodeando al obstáculo hasta que pueda volver a la trayectoria inicial.

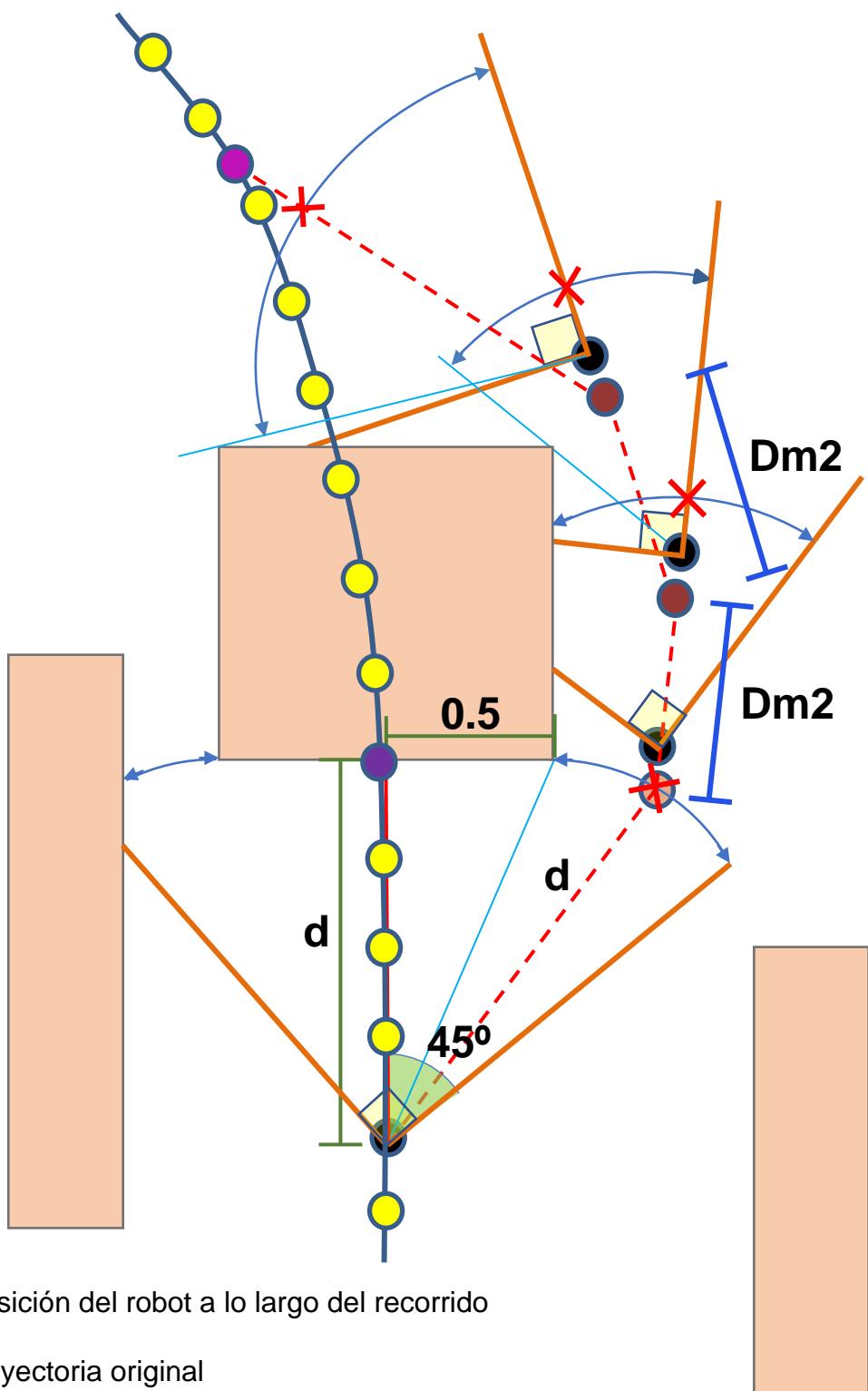


*Figura 4.10: Esquema del funcionamiento del modo 2*



*Figura 4.11: Esquema del funcionamiento del modo 3*

*Figura 4.12: Esquema completo del funcionamiento de la replanificación*



■ Posición del robot a lo largo del recorrido

■ Trayectoria original

■ Waypoints de la trayectoria original

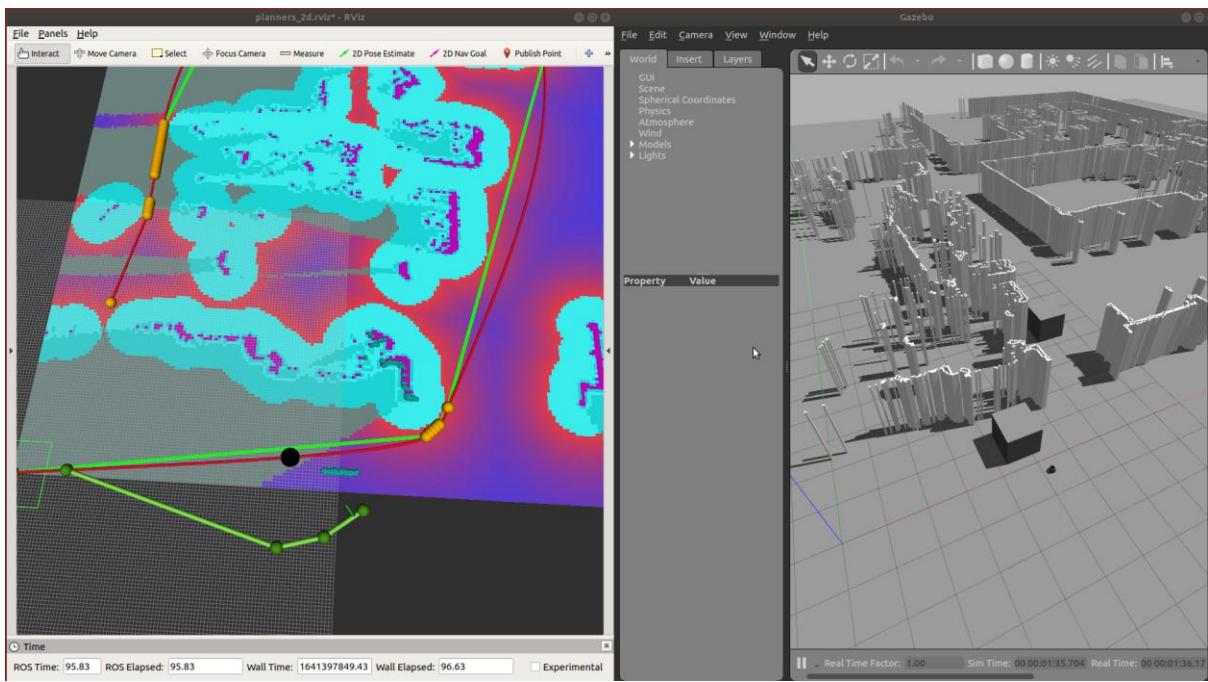
■ Trayectoria replanificada (línea discontinua)

■ Waypoints de la replanificación

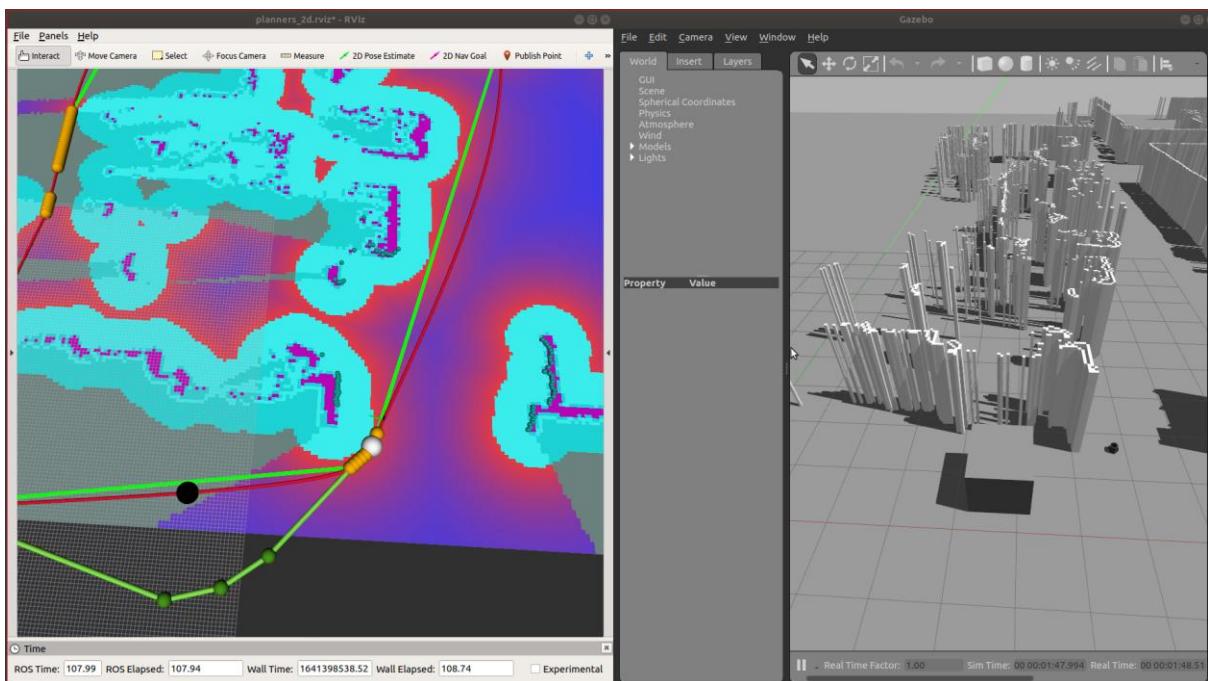
■ Punto de corte de la trayectoria original con el obstáculo

■ Punto de retorno a la trayectoria original

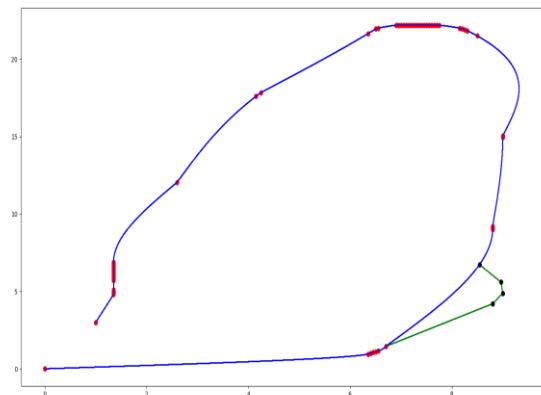
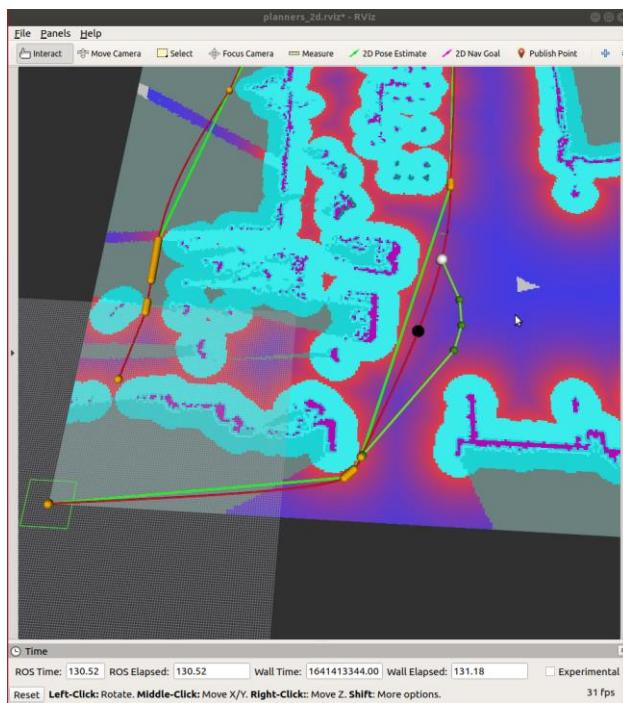
Dm2: distancia en modo 2 (los waypoints de la replanificación están equiespaciados)



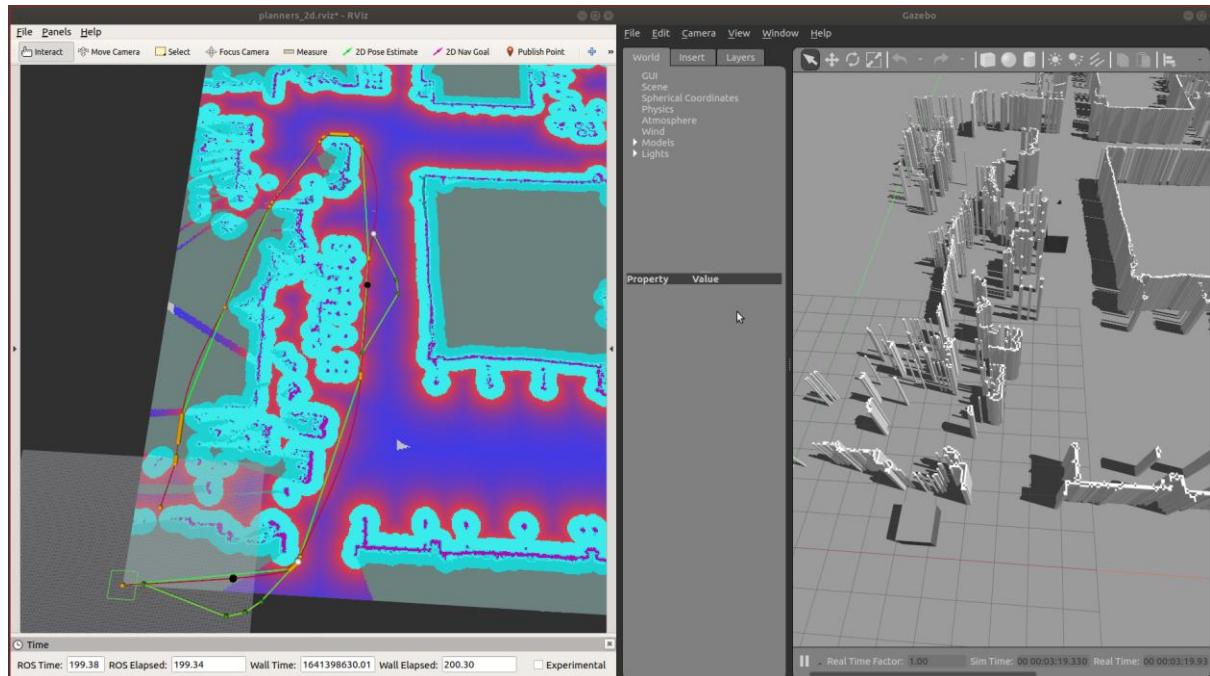
**Figura 4.13:** Ejemplo de robot realizando el Modo 2 del Replanificador en RViz/Gazebo



**Figura 4.14:** Ejemplo de robot realizando el Modo 3 del Replanificador en RViz/Gazebo, en blanco el punto de reinserción a la trayectoria original tras superar el obstáculo



**Figura 4.15:** Comparativa de gráfica (Eje X: Distancia en m / Eje Y: Distancia en m) generada en la tarea de replanificación con nodo propio (grafica.py) frente a visualización de RViz



**Figura 4.16:** Ejemplo del robot realizando la trayectoria tras haber evitado dos obstáculos que se interponen en ella

#### 4.2.11.- Gestión de la información de los waypoints locales

La función `interpolar()`, recibe los waypoints globales e interpola y genera waypoints locales. Estas se publican por el tópico `/waypoints_local` tanto en la planificación como la replanificación. Por simplicidad se utilizó el mismo medio de comunicación y también para no acoplar el replanificador con la función `interpolar()`.

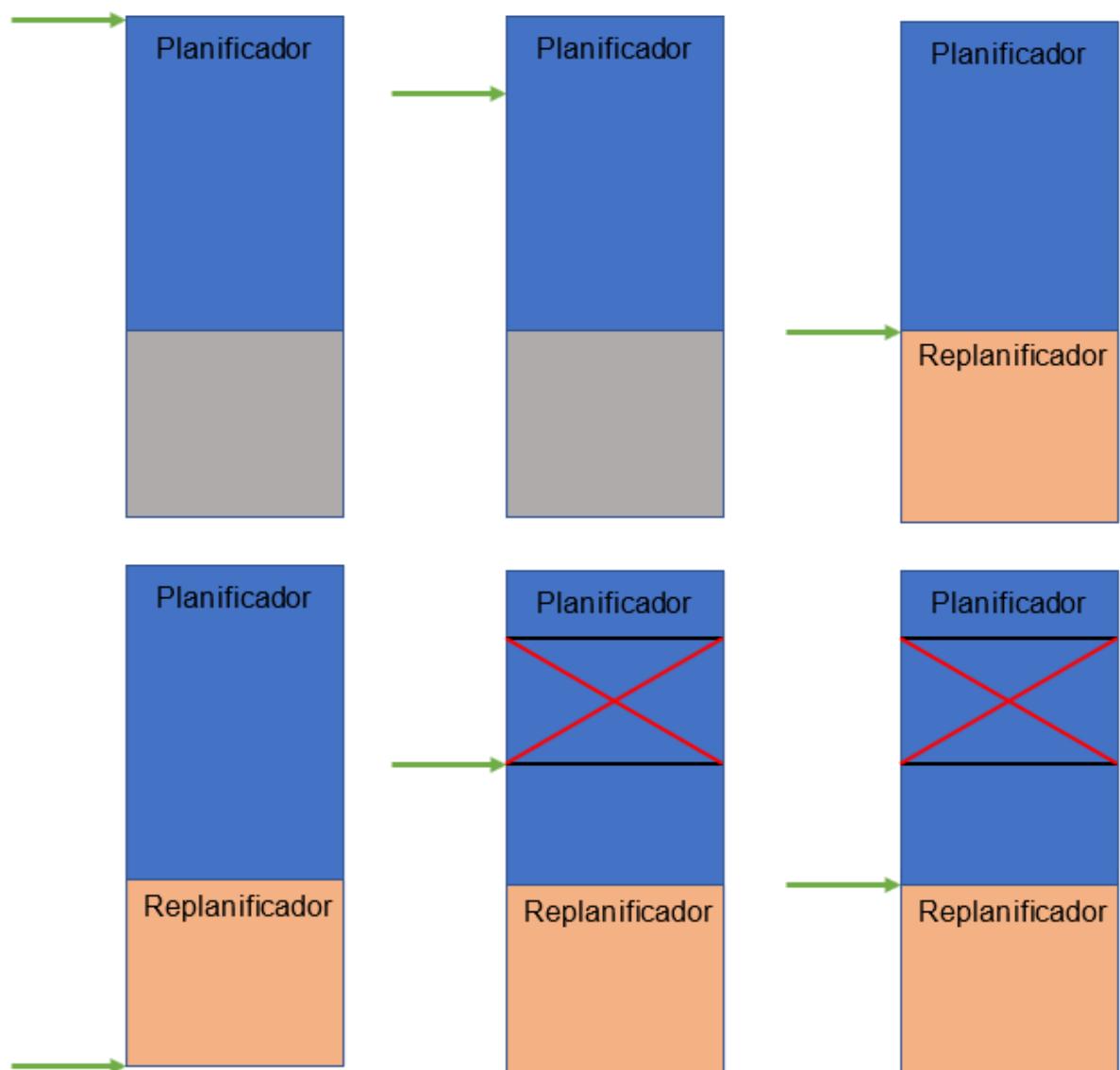
Una condición importante para que funcione el replanificador, es que se dé un tiempo (menor a 5 segundos) para que el interpolador genere los waypoints locales a partir de los waypoints globales. Estas se envían al pure pursuit, y el pure pursuit almacena la longitud de este vector. Una vez este dato esté almacenado, se puede obstruir la trayectoria, entrar en modo replanificador y generar nuevos waypoints locales, modificando la longitud inicial del vector. Si detecta un obstáculo en la inicialización, se incluirá waypoints del replanificador dentro de la longitud de la trayectoria inicial, resultando en un mal funcionamiento del programa.

Cuando entra en el modo replanificador y se llama a la función `interpolar()` que genera nuevos waypoints locales replanificados, el pure pursuit lo almacena en el mismo vector de waypoints locales, y por supuesto, al final del vector.

Pure pursuit, aplicará el control para los waypoints locales adicionales que se le envíen hasta que llegue al final, y hasta que la señal `replan` se desactive.

En el modo replanificar se determina el índice de retorno de la trayectoria inicial de los waypoints locales, y este índice se envía al pure pursuit, para volver a la trayectoria.

Se muestra ahora un esquema de lo que se acaba de comentar:



**Figura 4.17:** Estados por los que pasa el vector de waypoints al entrar en modo replanificación

#### 4.2.12.- Funciones desarrolladas y empleadas

Nombre	Entradas	Salidas	Función
interpolar	Waypoints globales	Waypoints locales	Interpola el conjunto de waypoints globales y genera un conjunto de waypoints locales
SenseAndAvoid	Waypoints locales de la trayectoria inicial	-----	Detecta si algún obstáculo se interpone sobre la trayectoria. En caso afirmativo, llama a la función replanificar, pasandole los waypoints locales desde el punto obstruido hasta el final. Eliminamos los puntos anteriores para que cuando finalice la replanificación, no retome la trayectoria ya recorrida.
replanificar	Waypoints locales reducidos de la trayectoria inicial.  Distancia al obstáculo en el instante de detección.	-----	Ejecuta el algoritmo desarrollado de los tres modos, hasta que retoma la trayectoria inicial.
indice_a_angulo	Índice correspondiente al vector ranges[ ] obtenidas a partir de /laser_scan.	Ángulo correspondiente al índice que ha recibido.	Convierte el índice correspondiente al vector frontal reducido ordenado, al ángulo, que equivale al índice, de la referencia de las medidas /laser_frontal
ordenar_vector_scan_fron tal	-----	laser_reducido_ordenado	Vector de rango reducido y ordenado perteneciente a las medidas frontales.
encuentra_max	search_array	index	Busca en un vector el índice del elemento medio dentro del rango de mayor longitud del rango de elementos máximos.
TF_robot_to_world	coordenadas de un punto (x,y) referidos al robot	coordenadas de un punto (x,y) referidos al mundo	Realiza la transformación de coordenadas
laser_scan_vector_to_wor ld_coordinates	-----	obstaculos_x [ ], obstaculos_y [ ]: vectores de las coordenadas de los obstáculos referidas al mundo	Transforma las coordenadas del obstáculo en los 180º frontales, referidas al robot, en coordenadas referidas al mundo.
laser_scan_point_to_wor ld_coordinates	(r, angulo)	(x,y)	Transforma una coordenada polar referida al robot a una coordenada cartesiana referida al mundo

PublishMarkers	<p>longitud: indica si es un punto o un vector.</p> <p>RGB: color del marker a visualizar.</p> <p>scale: tamaño del marker</p> <p>x: coordenada x (punto o vector de puntos)</p> <p>y: coordenada y (punto o vector de puntos)</p>	-----	Publica un punto o vector al tópico /visualization_marker_array para poder visualizarlo en Rviz. Estos se muestran y tienen vida ilimitada.
PublishMarkers_Obstaculo	<p>longitud: indica si es un punto o un vector.</p> <p>RGB: color del marker a visualizar.</p> <p>scale: tamaño del marker</p> <p>x: coordenada x (punto o vector de puntos)</p> <p>y: coordenada y (punto o vector de puntos)</p>	-----	Igual que la función PublishMarkers a diferencia que la visualización tienen un tiempo de vida finito. Nos interesa ver estas medidas en tiempo real, actualizándose.
visualizar_obstaculos	-----	-----	Llama a la función laser_scan_vector_to_world_coordinates() para obtener las coordenadas de los obstáculos y luego llamando a PublishMarkers_Obstaculo() los publica para visualizar.

#### 4.2.13.- Parámetros del planificador y del replanificador

Existen una serie de parámetros los cuales al modificar su valor cambia ligeramente el comportamiento de este subsistema.

- **modo\_pid:** Señal booleana si usar el controlador PID en lugar del pure pursuit.
- **interp\_dim:** Variable que indica si interpolar cada 2 puntos, o intentar interpolar con 3 puntos.
- **escala\_npoints:** Escalar el número de waypoints locales generados a partir de los waypoints globales. Cuanto mayor, más puntos habrá.
- **modo\_replanificar:** Señal booleana si usar el replanificador, es decir el bloque funcional de detección y evitación de obstáculos.
- **ang\_ini, ang\_fin:** Ángulos que fijan el rango de medidas frontales del láser scan, según la referencia descrito anteriormente.
- **tolerancia\_wp\_replan:** El margen de tolerancia que existe cuando se considera que un punto de la posición del robot es igual o coincide con un waypoint global.

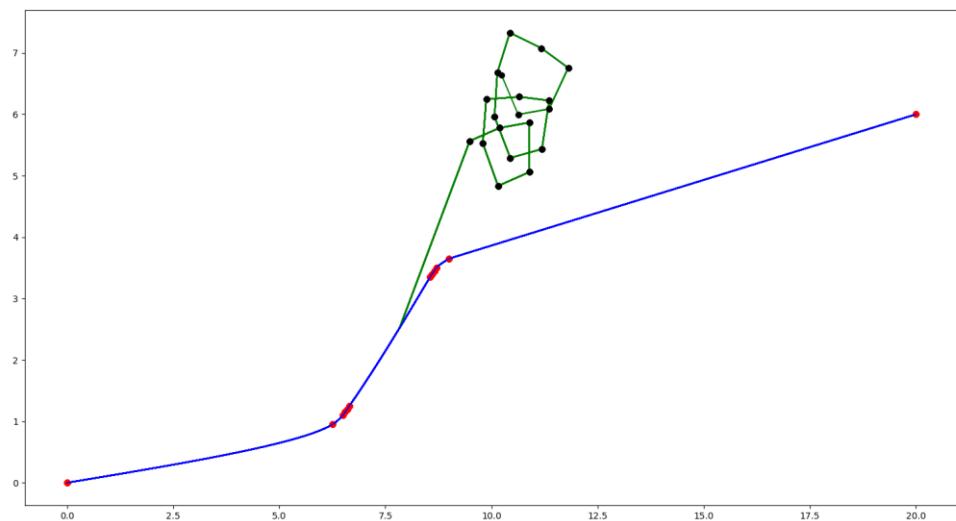
No siempre conviene que este sea el mínimo posible. En un caso que el robot circule con velocidades altas, conviene permitirle más tolerancia, ya que a grandes velocidades una pequeña desviación podría salirse de este margen. Es importante destacar que, la referencia de un waypoint a otro cambia una vez lo alcance (con tolerancia), por tanto si un waypoint se lo salta (y no lo alcanza), el robot dará la vuelta para alcanzar la referencia antes de seguir por los waypoints posteriores.

- **longitud\_tramos\_modo\_2:** parámetro que define la longitud de los tramos en el modo 2 del replanificador. Si está longitud es menor la trayectoria nueva será más curvilínea, pero en el extremo puede causar movimientos circulares; si es mayor, generará una trayectoria más recta, y en el extremo se encuentra el peligro de irse demasiado lejos del obstáculo o/y la trayectoria y por tanto desubicarse respecto de la trayectoria inicial.

Se muestra un vídeo ejemplificando este problema:

<https://drive.google.com/file/d/1OESXZixIBxW5QX78VcEMTVbKu6hUPgLB/view?usp=sharing>

Se adjunta también una gráfica de dicho experimento mostrando el problema mencionado que aparece:



**Figura 4.17:** Problema que aparece en cierto experimento si el parámetro “`longitud_tramos_modo_2`” no es el adecuado para la disposición de los obstáculos en el mapa

## 5.- Algoritmo de control “Pure Pursuit” para seguimiento de trayectorias

### 5.1.- Introducción

A la hora de controlar un robot móvil con ruedas, ya sea con configuración diferencial como este caso, o con otras características o configuraciones (como Ackermann, Omni Wheel, etc.) existen multitud de algoritmos disponibles y conocidos en el extenso campo de la robótica.

Aunque existen algoritmos complejos, como pueden ser el uso de MPC (Model Predictive Control), LQR (Linear Quadratic Regulator) u otros, que ofrecen muy buenos resultados, se propone para este proyecto el desarrollo del algoritmo **Pure Pursuit o Persecución Pura**, el cual es de los más simples existentes para el control de robots móviles, pero a su vez ofrece resultados bastante aceptables en cuanto al seguimiento de trayectorias y es considerado un controlador robusto.

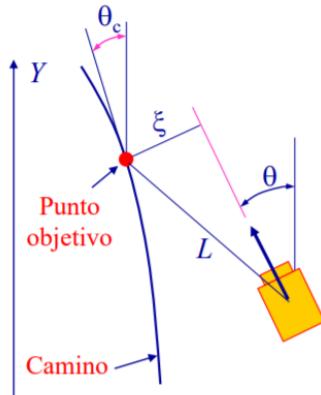
Así mismo, con el objetivo de realizar comparaciones de rendimiento y efectividad en el seguimiento de la trayectoria, se ha realizado en paralelo el desarrollo también de un algoritmo pseudo PID, basado en un paquete de ROS llamado “diff-drive”, cuyas comparaciones respecto del Pure Pursuit implementado se verán posteriormente.

### 5.2.- Desarrollo

Para implementar este algoritmo, conocido su funcionamiento de las referencias estudiadas acerca del mismo, así como de las clases teóricas, se ha decidido crear un nodo independiente que será el encargado de llevar a cabo el control de nuestro robot móvil diferencial a la hora de seguir con éxito la trayectoria usando Pure Pursuit.

Para ello, como ya se ha mencionado varias veces, partimos de una trayectoria que ha sido proporcionada por el conjunto de bloques funcionales Planificador e Interpolador, los cuales confeccionan la trayectoria en el formato deseado para este algoritmo de manera que como veremos, iremos recibiendo los waypoints mediante un “rastreo” con el conocido parámetro de LookAhead ( $L$ ), que es el único parámetro de este algoritmo de control simple y cuyo valor tiene efectos muy notables sobre el comportamiento del robot a la hora de seguir la trayectoria.

Dicho esto, se adjunta los siguientes esquemas ilustrativos del algoritmo para terminar de asentar las bases de su funcionamiento previo a la explicación del nodo desarrollado.



$$\theta_e = \theta - \theta_c$$

$$\gamma_e = \gamma - \gamma_c$$

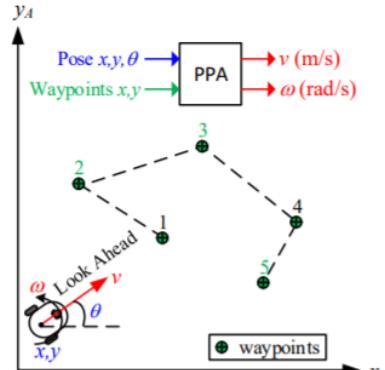


Fig.2. Inputs/outputs of PPA and other parameters on coordinate frame.

*Figura 5.1: Principio de funcionamiento del Pure Pursuit*

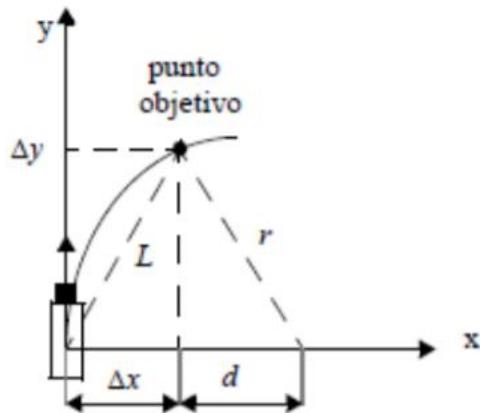
Como vemos en el esquema de la derecha, el algoritmo necesita como entradas la siguiente información:

- **“Pose” del robot:** Es una combinación de su posición en el espacio(x, y, z), así como de su orientación, dada por un *quaternion* con 4 componentes (x, y, z, w). Este tipo de información es muy común en el framework de ROS y por tanto es muy manejable para trabajar con ella, ya que existen mensajes con distintos formatos que permiten la transmisión de esta información, esto se verá más adelante.
- **Waypoints de la trayectoria:** Cómo es lógico también debe recibir los waypoints que describen la trayectoria. Sin embargo, aunque el esquema indica que sólo recibe las componentes (x, y) de la posición del robot, ha sido necesario entregarlos con ese formato tipo “Pose” comentado anteriormente para evitar problemas de compatibilidad. Como se ha visto en los apartados correspondientes y veremos al analizar el nodo, los waypoints de la trayectoria se calculan en su totalidad previamente a que el control comience a funcionar.

Visto esto y aunque se ha mencionado ligeramente, vemos que las salidas que proporciona el algoritmo son:

- **Velocidad lineal:** Esta componente de la velocidad del robot por diseño del algoritmo se deja con un valor constante, el cual puede ser variado en diferentes experimentos (o bien entre modos de funcionamiento (replan/normal)) para ver cómo responde el algoritmo de persecución de trayectoria con una velocidad lineal algo más agresiva frente a una velocidad lineal conservadora.

- **Velocidad angular:** Es en esta componente de la velocidad donde realmente actúa el control, de manera que, si observamos las siguientes ecuaciones:



$$r = \Delta x + d.$$

$$(r - \Delta x)^2 + (\Delta y)^2 = r^2$$

$$r = ((\Delta x)^2 + (\Delta y)^2) / (2 \Delta x)$$

$$r = L^2 / (2 \Delta x)$$

$$1/r = (2/L^2) \Delta x$$

*Figura 5.3: Desarrollo de las ecuaciones para obtener la ley de control del algoritmo Pure Pursuit*

Si nos fijamos en la expresión recuadrada en rojo, apreciamos que se trata de la curvatura de la propia curva que describiría el robot para ir desde su “Pose” actual hacia el **punto objetivo** tomado por el algoritmo en un instante de tiempo determinado. Dicho esto podemos reformular esta expresión para mostrarla como ley de control del algoritmo.

$$u = (-)2 \cdot \Delta x / L^2$$

Siendo el signo “-” necesario o no en función de las características del sistema de referencia local del robot y su trayectoria de referencia.

Será esta por tanto la ley de control que actuará sobre la velocidad angular del robot para que trate de realizar la curva explicada entre su Pose y el punto objetivo en cada momento, variando en cada instante al marcar nuevos puntos objetivos y descartar los anteriores.

Añadir además, cómo extracto del artículo [“*Implementation of the Pure Pursuit Path Tracking Algorithm*”] de 1992 por R. Craig Coulter, la idea de que este algoritmo de control Pure Pursuit se asemeja a un controlador tipo P, en el sentido de que como se cita: “Hay cierta similitud en forma respecto a un controlador proporcional donde la ganancia ( $K_p$ ) es  $2/L^2$ , aunque el error en este caso se corresponde con el offset en X (o en Y según el SR local del robot) respecto al punto objetivo del robot móvil en cada instante”.

### 5.3.- Algoritmo de seguimiento de waypoints

Aparte del algoritmo de control pure-pursuit, existe otro algoritmo de *suministro de waypoints* que se considera de igual importancia que el primero. El algoritmo de seguimiento de los waypoints está intrínsecamente relacionado con el parámetro Lookahead L. Se detallan los pasos del algoritmo a continuación:

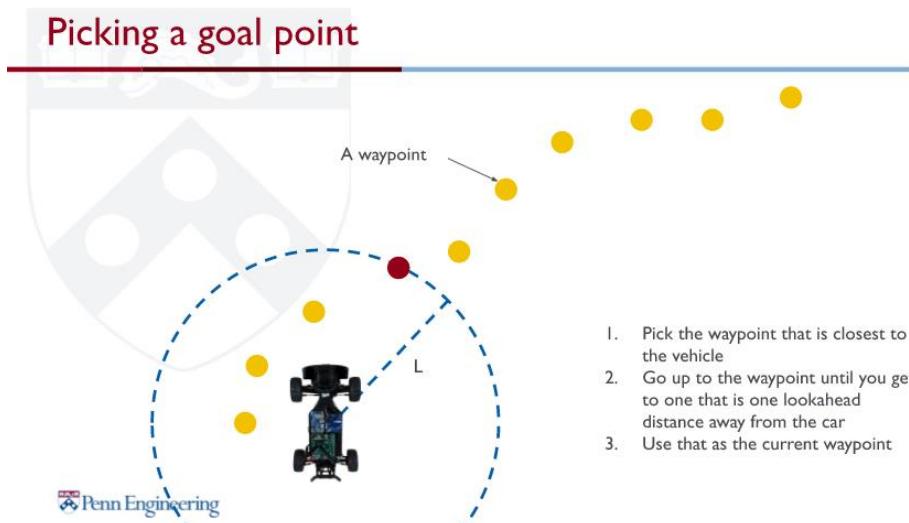
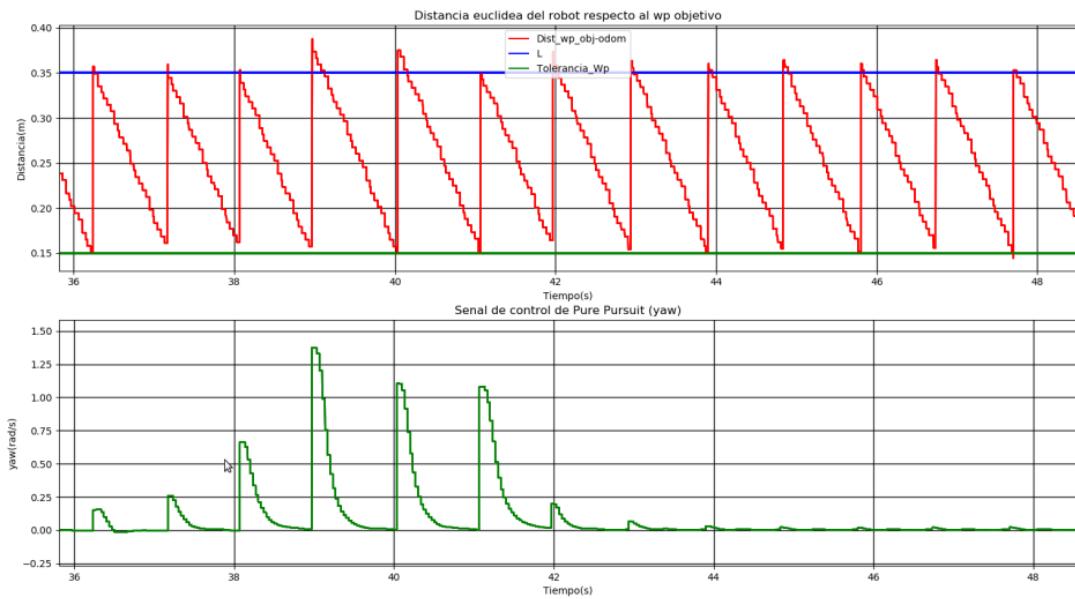


Figura 5.4: Imagen que ilustra los pasos a seguir en el algoritmo Pure Pursuit

1. Paso inicial: Elegir el waypoint más cercano al robot. En nuestro caso, el primer waypoint local dentro del vector de waypoints locales, coincidirá con la posición inicial del robot.
2. Paso iterativo: Avanzar hacia el waypoint que esté a una distancia L desde el robot. Dado el vector de waypoints locales, suponemos que la distancia entre waypoints es menor que la distancia L, algo que siempre ocurrirá ya que resulta en un mejor funcionamiento, existirá un conjunto de waypoints en la vecindad del robot cuya distancia será menor que L, y por tanto no se tomará como referencia para controlar. Se buscará un punto a distancia L, y se tomará como **punto objetivo** del algoritmo pure-pursuit, y se saltará las anteriores. Una vez alcanzado este punto, se iterará este proceso para buscar el siguiente.



**Figura 5.5:** Imagen que ilustra los pasos a seguir en el algoritmo Pure Pursuit

Se muestra en una gráfica ambos algoritmos en un experimento; el suministro de waypoints y el pure-pursuit. Se puede observar que en el suministro de waypoints, de un punto a otro, la distancia desde el robot hasta el waypoints en el momento de conmutación es siempre igual a la distancia Lookahead, y cuando esta distancia llega a ser cero, se pasa al siguiente punto.

Tras haber analizado los conceptos de este algoritmo y tener claro cómo funciona, se puede pasar a analizar el nodo desarrollado para implementarlo.

#### 5.4.- Nodo “Pure\_Pursuit.py”

Para no hacer excesivamente larga esta memoria, no se adjuntarán capturas de todo el código, sin embargo, se recomienda fervientemente acudir a los códigos para revisar la implementación que se va a explicar a continuación, siendo que además dicho código está comentado con detalle.

Se ha optado por desarrollar el nodo en lenguaje Python al considerarlo mejor opción en cuanto a lo intuitivo que es dicho lenguaje al sumergirse en un lenguaje de programación orientado a objetos (totalmente necesario para trabajar en el entorno de ROS), lo cual suponía una novedad para nosotros.

En este código realizaremos lo siguiente:

- Comenzamos importando todas las librerías de Python necesarias para realizar las operaciones y funciones a lo largo del código.
- Se ha definido `self.loop_rate = rospy.Rate(20)`, donde se consigue un tiempo de muestreo de 50ms, que frente al tiempo de control deseado, es una tasa de muestreo suficiente como para obtener un control adecuado.
- Tras ello declaramos una clase llamada **PurePursuit** que conlleva la mayor parte del peso de la implementación. Para hacerse una idea de la importancia de esta clase creada, se puede resumir como en el “main” del código lo único que se hace es iniciar el nodo ‘pure\_pursuit’, crear un objeto llamado “nodo” de la clase PurePursuit anterior y ejecutar la función “main” del nuevo objeto “nodo”, siendo que esta función está descrita dentro de la clase PurePursuit y es la que encierra el comportamiento cíclico del algoritmo comprendiendo un bucle típico de ROS en Python conformado por:

***While not rospy.is\_shutdown():***

...

*(algoritmo)*

...

***self.loop\_rate.sleep()***

- **Si entramos a analizar la clase “PurePursuit”:**

Tenemos en primer lugar el método “`__init__`” que es un método fundamental a la hora de inicializar los atributos o variables de la clase que se ha creado. En él, se inicializan una serie de variables, algunas inicializadas directamente en el propio código y otras mediante un archivo “.yaml”, para hacer más cómoda la configuración de dichos parámetros del código a través de dicho archivo. Por mencionar alguna de las variables más destacadas, tendremos por ejemplo “L”, correspondiente al parámetro de control del algoritmo, la distancia de Lookahead.

También tenemos dos parámetros más “velocidad\_replan” y “velocidad\_normal” que permiten fijar la velocidad lineal constante aplicada al robot para el caso de que se encuentre en modo replan, si ha detectado un obstáculo interfiriendo en la trayectoria original (se reducirá la velocidad) o bien se encuentre en modo normal, lo que implica

que seguirá la trayectoria original con una velocidad lineal mayor a la del replan al no haber detectado ningún obstáculo interfiriendo. Estos 3 parámetros son los únicos de este código configurables desde el archivo “.yaml”.

Dejando de lado el resto de variables útiles para el código, pasamos a la sección de este método `__init__` en la que se configuran los *Subscribers* de ROS que serán los que recojan la información necesaria de ciertos *Topics* para tener esa comunicación con el resto de subsistemas:

```
# Subscribers
rospy.Subscriber('/waypoints_local', PoseStamped, self.callback_wp_local)
rospy.Subscriber('/odom', Odometry, self.callback_odom)
rospy.Subscriber('/waypoint_return_index', Float64, self.callback_return_index)
rospy.Subscriber('/replan', Float64, self.callback_replan_signal)
rospy.Subscriber('/parada', Float64, self.callback_parada)
```

**Figura 5.6:** Subscribers de ROS implementados para el nodo controlador Pure Pursuit

Si vamos en orden:

- **/waypoints\_local:** De este *topic* se recogerán y almacenarán el conjunto de waypoints locales (ya interpolados respecto de los waypoints globales), de manera que definirán la trayectoria a seguir por el robot mediante el uso del algoritmo Pure Pursuit.
- **/odom:** De aquí recogeremos la información correspondiente a la odometría del robot, que originariamente se usaba para comparar ese offset en X o Y del robot respecto del punto objetivo fijado. Finalmente, no fue necesario ya que se consiguió referir los waypoints al sistema de referencia local del robot, teniendo así por tanto dichos offsets tanto en X como en Y de manera directa, sin necesidad de hacer operaciones con la odometría y las coordenadas de los waypoints referidas al sistema de referencia global.
- **/waypoint\_return\_index:** En este *topic* recibiremos por parte del replanificador una especie de “puntero” el cual apuntará al waypoint local, al que debe “acoplarse” el robot tras haber evitado el obstáculo, del vector almacenado al principio con el conjunto de waypoints de la trayectoria completa. Es decir, al detectar un obstáculo en la trayectoria, se replanifica una trayectoria alternativa por tramos, hasta que, al detectar que se ha rebasado el obstáculo en dicha trayectoria alternativa, tratará de buscar un corte o intersección del último waypoint de la trayectoria alternativa con la trayectoria original, la encontrará en alguno de esos waypoints de la trayectoria original y es ahí donde al encontrarla, el replanificador manda por este *topic* el índice respectivo del vector completo de waypoints que apunta a dicho waypoint de reinserción a la trayectoria. Así estaremos ignorando un cierto conjunto de waypoints que están obstruidos por el obstáculo para que el Pure Pursuit se acople de nuevo a la trayectoria original de manera adecuada.

- **/replan**: En este *topic* el replanificador mandará una señal booleana ('0' o '1') para indicar al controlador si se ha detectado un obstáculo en la trayectoria o no, ya que el Pure Pursuit modifica ligeramente su comportamiento según tenga que actuar en modo replan o en modo normal.
- **/parada**: Este topic también tiene como Publisher al Replanificador, de manera que de nuevo se tendrá una señal booleana ('0' o '1') para indicar al controlador que debe parar al robot. Esto será necesario en los dos siguientes casos:
  - Cuando el robot ha completado la trayectoria y, por tanto, ha alcanzado el último de los waypoints que definen dicha trayectoria.
  - En caso de que el obstáculo sea detectado a menos de 1.30 m, lo cual puede ocurrir si el obstáculo aparece repentinamente o se "cruza" en la trayectoria mientras el robot avanza, se ha considerado que por las características de funcionamiento del algoritmo replanificador, el obstáculo no se podría evitar y se decide parar el robot para evitar su colisión con dicho obstáculo.

Pasamos ahora a analizar el único Publisher que tendremos en este nodo:

```
#Publishers
self.vel_pub = rospy.Publisher('/cmd_vel', Twist, queue_size=1)
```

*Figura 5.7: Publishers de ROS implementados para el nodo controlador Pure Pursuit*

Como se mencionó en la explicación teórica del algoritmo, la única salida es la velocidad controlada del robot, con su componente lineal fijada a un valor constante (mayor si está en modo normal y algo más reducida si está en modo replan) y su componente angular siendo esta sobre la que actúa la ley de control del algoritmo. De esta manera, este nodo tras computar la velocidad angular necesaria a aplicar para que el robot "persiga" al punto objetivo determinado por el propio algoritmo en cada instante, publicará en el topic **/cmd\_vel** un mensaje con ambas componentes de velocidad computadas y en el formato necesario para aplicar al robot.

Por último, para finalizar el método "`__init__`", creamos un objeto "tf.TransformListener()":

```
# Listener
self.tf = tf.TransformListener()
```

*Figura 5.8: Creación de objeto listener para trabajar con las transformaciones entre "frames" o sistemas de referencia de ROS*

Este listener será usado para trabajar con las transformaciones entre sistemas de referencia en el código.

Tras terminar de explicar el método "`__init__`", se declaran una serie de funciones o métodos tipo "**callback**" los cuales tienen como objetivo recoger la información

procedente de los *subscribers* comentados en el apartado anterior. En cada “callback” la información se recogerá de cierta manera conveniente, ya sea haciendo “append” de elementos en un vector o list de Python o bien haciendo asignaciones simples a variables.

A continuación, tenemos el siguiente método **TF\_world\_to\_robot(self, x\_w, y\_w)**. Esta función tiene como objetivo transformar los puntos que recibe (**x\_w, y\_w**), que están referidos al sistema de referencia global (“world”/ “map” (son equivalentes, ambos son globales)) al sistema de referencia local del robot (“base\_link”). Para ello se comprueba si la transformación entre ambos “frames” está disponible, y si lo está, se realiza dicha transformación descrita de “map” a “base\_link”, con el objetivo básicamente de referir los waypoints (originalmente referidos al sistema de referencia global) al sistema de referencia local del robot. Tras realizar la transformación, devolverá los puntos entregados ya referidos al SR local del robot, (**x\_r, y\_r**).

El siguiente método corresponde a **publica\_velocidad(self, vx, vy, vz, wz)**. Esta función es bastante simple y su única utilidad es publicar las velocidades recibidas como parámetros (**vx, vy, vz, wz**) en el topic **/cmd\_vel**. Únicamente publicaremos las 3 componentes de la velocidad lineal y la componente Z de la velocidad angular, donde recibirá del algoritmo:

- $vx = velocidad\_replan/velocidad\_normal$ , en función del modo en el que estemos
- $vy = vz = 0$
- $wz = yaw = 2 \cdot dy / L^2$ , donde vemos la aplicación de la ley de control analizada en la parte teórica del algoritmo, con varias modificaciones como son el signo (-), que no es necesario, así como el offset, que es respecto del eje Y local del robot (dy), debido a que los ejes locales del robot (**x\_r, y\_r**) están invertidos respecto a los ejes habituales de los esquemas teóricos.

Este método también será usado para parar el robot en caso de que sea necesario publicando todas las velocidades a valor 0: `self.publica_velocidad(0.0, 0.0, 0.0, 0.0)`.

Continuamos con el método que implementa el algoritmo Pure Pursuit en sí mismo: **controladorPP(self, i)**.

Donde “i” es el índice que apuntará al waypoint dentro del vector de waypoints que definen la trayectoria, y que en cada instante irá recibiendo el índice correspondiente al waypoint que el algoritmo determine como punto objetivo.

En esta función, comenzamos haciendo uso de la función anteriormente comentada: `TF_world_to_robot()` para transformar el waypoint actual (punto objetivo) a coordenadas referidas respecto del sistema de referencia local (**xr, yr**) del robot (“base\_link”).

Como se explicó anteriormente, existe un **algoritmo de suministro de waypoints** relacionado con el parámetro L.

Todos los waypoints de la trayectoria inicial están almacenados en un vector ordenados, tal que el orden de los puntos coincide con el orden de seguimiento de la trayectoria. Se supone que la distancia entre puntos es menor a la distancia L, por tanto, en la vecindad del robot, habrá puntos que no se tomará como punto objetivo.

La búsqueda del punto objetivo desarrollado en el código, se trata de comparar la distancia euclídea del waypoint siguiente (x,y) con la distancia lookahead L, si esta distancia es menor que la L, se incrementa el índice hasta encontrar un waypoint (x,y) que tiene distancia euclídea igual o mayor que L. Una vez encontrado este punto, se le pasará el índice del vector de waypoints al controlador pure-pursuit que tomará como punto objetivo.

Permanecemos dentro de la función del controlador pure-pursuit aplicando las velocidades necesarias para reducir ese error u offset en la distancia al punto objetivo al valor impuesto.

Detallando un poco más lo que se realiza (en bucle) mientras este error no sea inferior a ese umbral, tenemos que se actualizan de nuevo las coordenadas (xr, yr), se configura como error para la señal de control el offset respecto del eje Y local del robot: **dy = yr**, debido esto a que, como se mencionaba, los ejes X, Y locales del robot están invertidos en Gazebo/ROS a cómo se colocan habitualmente en las formulaciones teóricas.

Posteriormente, se impone la ley de control ya conocida y alterada para el caso que nos ocupa:

```
# Orientacion que aplicamos al robot  
yaw = + 2*dy / (self.L ** 2)
```

*Figura 5.9: Ley de control aplicada a la componente Z de la velocidad angular*

Tras computar así la velocidad angular necesaria para controlar el seguimiento de la trayectoria, se imponen una serie de condiciones de manera que en orden de prioridad tendríamos:

- La señal “**parada**”, que impera sobre el control, y pararía el robot en caso de ser recibida.
- La señal “**replan**”, cuya contribución principal será variar la velocidad lineal del robot diferenciando si se encuentra en modo replan o en modo normal.

También se actualizan (xr, yr) de nuevo al final del bucle, al ser crítico sus valores a la hora de continuar entregando waypoints o no, es decir, tomar como punto objetivo el waypoint siguiente.

Finalmente, para concluir este nodo, llegamos al método **main(self)** de esta clase PurePursuit, en la cual se hace uso de varios de los métodos anteriores ya descritos.

Este método se repetirá en bucle y será el que en conjunto con todo lo descrito anteriormente, implemente en cada instante el algoritmo Pure Pursuit desarrollado.

Imponemos de nuevo en este método, la condición imperante de la señal “**parada**”, para parar el robot en el caso de que encuentre un obstáculo a una distancia inferior al mínimo establecido como distancia a la que el robot puede evitar dicho obstáculo. De esta manera, mientras esta señal esté inactiva, comenzamos asegurando que el robot está en reposo, introducimos una espera de 5 segundos, en los cuales los “**callback**” dispondrán del tiempo necesario para recibir y acomodar el vector completo del waypoints que definen la trayectoria para este nodo.

Almacenamos la longitud inicial, básicamente para conocer cuando, tras ir recorriendo los waypoints a lo largo del vector de waypoints, hemos llegado al último de ellos y, por tanto, completado la trayectoria.

Finalmente, permanecemos en un bucle hasta que se complete la trayectoria original, siendo que, a lo largo de la realización de la trayectoria, tenemos dos posibilidades:

- Mantenernos en la trayectoria original, en caso de que no se interponga ningún obstáculo en ella, donde iremos entregando waypoints al método `self.controladorPP(i)` cada vez que el error sea inferior al umbral ya mencionado.
- Se recibe la señal “**replan**” por parte del módulo replanificador, lo cual implica que se ha detectado un obstáculo que interfiere en la trayectoria original. Entonces mientras esta señal siga activa, el controlador PurePursuit estará recibiendo unos waypoints alternativos entregados por el replanificador, de manera que seguirá una trayectoria nueva, que permite evitar el obstáculo detectado. Una vez esta señal se desactiva, implica que el replanificador ya ha encontrado el waypoint de la trayectoria original en el que el robot debe reinserirse. Para ello, se actualiza el “puntero” al vector de waypoints (“*i*”), como ya vimos a través de la variable “`self.return_index`” que recibía dicho waypoint de reinserción en un *topic* donde publicaba el replanificador.

Tras esto, volvemos a la situación normal en la que estamos recorriendo la trayectoria original tras haber evitado exitosamente el obstáculo, permitiendo la opción de volver a evitar otros obstáculos en el resto de la trayectoria original, si los hubiese.

Como se comentaba, también se impone al final de este método “main” la parada del robot una vez se haya alcanzado el último waypoint, es decir, se haya completado con éxito la trayectoria.

## 5.5.- Análisis del parámetro “L”

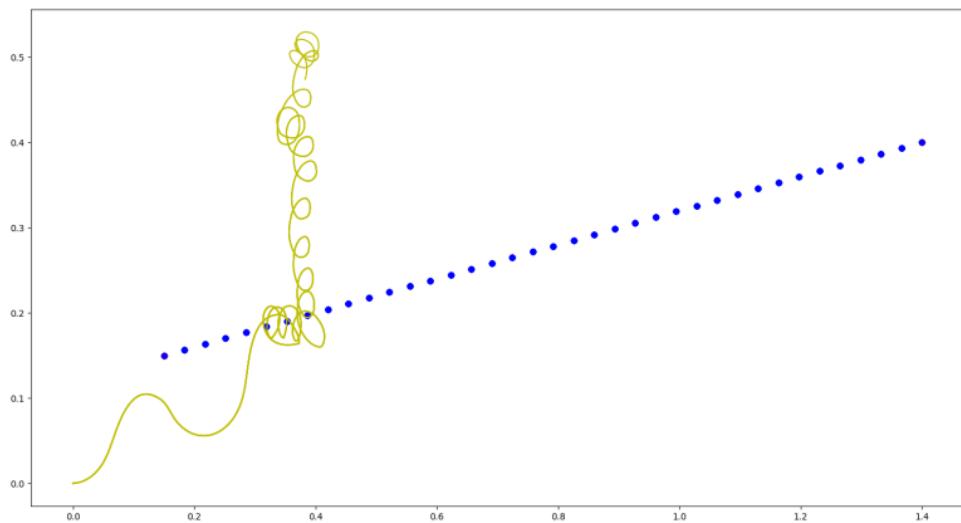
Experimentalmente se han probado diferentes valores de L, y para determinar cuál será su valor en el mejor de los casos observamos su comportamiento.

Aunque teóricamente es un único parámetro, en la práctica se ha observado que influyen dos variables más.

- **El margen de tolerancia que considera que el robot ha alcanzado un waypoint (wp\_tolerancia).** Esto es la diferencia entre la posición del robot actual (odometría) y la posición del waypoint a alcanzar. Se podría pensar que lo ideal sería que la tolerancia fuese cero, pero analizándolo vemos que no es así.

Se analiza qué comportamiento tiene un *control transitorio* en el contexto del pure-pursuit, es decir, donde no existe un régimen permanente porque la referencia cambia continuamente, y es en el instante en que se alcanza el punto objetivo cuando hay un cambio de referencia, algo muy común en la robótica. La pregunta es: ¿Con qué precisión se desea alcanzar esta referencia? La respuesta es, que depende de L.

Con un valor de L muy grande, el error cuadrático medio es mayor, ya que el robot tarda más en alcanzar el waypoint porque el accionamiento es menor. Pedirle al controlador una tolerancia cero con L grande puede provocar efectos indeseados. Por ejemplo, una situación en la que, aunque el robot se dirige aproximadamente en la misma dirección de la trayectoria, al carecer de un control más rápido, no alcanza la referencia (el punto objetivo) con la precisión que se le ha pedido (tolerancia cero) y se daría un caso en el que el robot daría la vuelta, dirigiéndose hacia la referencia no alcanzada, y entra en movimientos circulares\* alrededor del waypoint que no alcanzó hasta que lo alcance. El único mecanismo que tenemos para suministrar los waypoints de una forma más fluida, en sincronismo con el movimiento del robot, reside completamente en el valor de esta tolerancia. En este ejemplo, si se permitiera mayor tolerancia, el robot no daría la vuelta, y seguiría sobre la trayectoria como se deseaba, aunque no con una precisión exacta. Es importante remarcar que esta precisión inexacta no es un defecto, sino una característica del control, propio del parámetro L. A la hora de elegir la tolerancia, es necesario tener en cuenta la precisión que tendrá según la L para evitar pedirle al controlador algo de que no es capaz, limitado por la propia L.



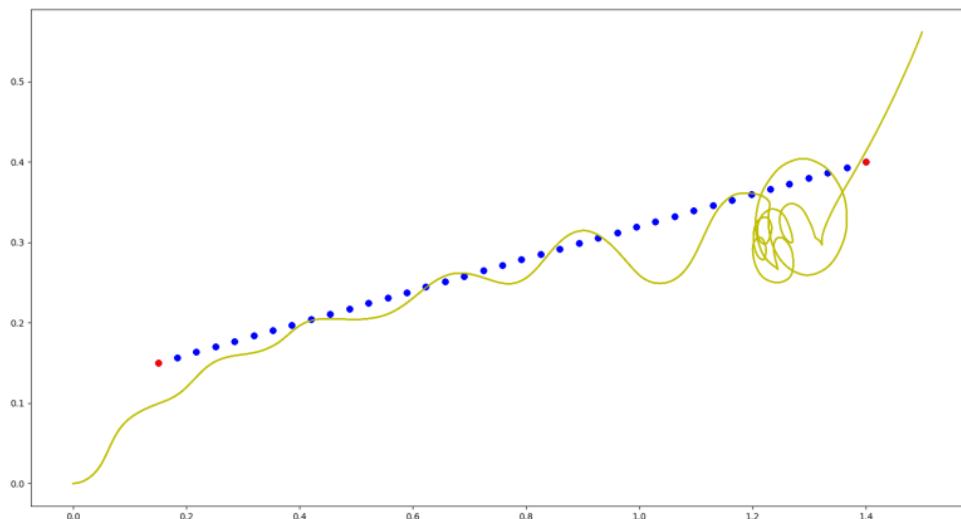
**Figura 5.10:** Efecto de elegir una tolerancia demasiado restrictiva respecto a considerar el punto objetivo como alcanzado

\* Como la señal de control es el giro, y por tanto si el error es distinto de cero, al cabo de un tiempo se producirían movimientos circulares.

Con un valor de  $L$  pequeño, habrá mayor accionamiento sobre el error, resultando en un control más rápido, y por tanto conviene una tolerancia mucho menor, ya que conseguirá un mejor acercamiento respecto del punto objetivo y obtenemos un seguimiento más preciso de la trayectoria. El peligro de una  $L$  demasiada pequeña reside en las posibles oscilaciones provocadas por un control demasiado agresivo.

- **El número de puntos interpolados** (escala\_npoints). Lo importante de este parámetro no es tanto el número de puntos sino *la distancia entre puntos*. En el programa esta escala se define como **número de puntos por metro**. Según el valor de  $L$ , la distancia entre puntos es fundamental para determinar si la trayectoria será controlable o no. A controlable nos referimos a un comportamiento predecible según las condiciones de trabajo.

Decimos que no es controlable si el waypoint está demasiado lejos ( $L$  pequeña), resultando en efectos indeseados del controlador. Si está demasiado lejos, aplicará un giro durante un tiempo que podría resultar en oscilaciones hasta empezar a hacer movimientos circulares, podría incluso alcanzar la trayectoria correctamente después de dicho movimiento, pero es indeseado, y más importante, es un comportamiento no predecible, y por tanto, no controlable (aunque alcance la referencia al final).



**Figura 5.11:** Ejemplo de movimiento oscilante al final de la trayectoria propuesta tras el cual, consigue retomar la referencia y sobre pasa el punto objetivo por la inercia del movimiento

También existe el caso que no es controlable si el waypoint está demasiado cerca ( $L$  grande), provocando que todos los puntos en la vecindad del robot, a distancias menores que  $L$ , ni se tomen como referencia y, por tanto, resultando en un estancamiento del robot y un avance nulo.

### ¿Cómo determinar el valor de los tres parámetros?

Las tres variables están muy relacionadas, y la decisión sobre una repercute en las demás.

Cuando se expresa a continuación como “ $L$  grande” o “ $L$  pequeño” es una medida completamente relativa a la forma y dimensión de la trayectoria, e incluso de las dimensiones físicas del robot.

1. El principal determinante viene siendo la longitud de trayectoria inicial, ya que evidentemente, es la variable a controlar (el error es una distancia). Según la longitud de la trayectoria, y de especial interés las curvas, se determina un valor de  $L$  adecuado. Por ejemplo, segmentando un tramo curvilíneo en tramos rectos, se puede estimar la  $L$  máxima que se permitiría recorrer dicha trayectoria. Esta restricción de  $L$  según la curvatura de la trayectoria inspira la idea de un control adaptativo del parámetro  $L$ , ya que una trayectoria es un conjunto de tramos con distintas características geométricas. Se menciona que se ha implementado una pequeña aplicación de una  $L$  adaptativo en el replanificador que se explicará más adelante.

2. Una vez se tiene el valor de  $L$ , se deberían generar puntos sobre la trayectoria que tengan una distancia entre sí menor que la distancia  $L$ . Hemos verificado que más de diez puntos por  $L$ , es poco notable en el seguimiento de la trayectoria, ya que el algoritmo de suministro de waypoints se los saltará y no tienen ningún efecto al no ser elegidos como puntos objetivo, y si lo tienen, es despreciable.
3. Una vez fijado el valor de  $L$  y se asegura que existe un valor adecuado de puntos, hay que preguntarse ¿con qué precisión se llegará al waypoint? Ésta depende principalmente de  $L$ , y como se ha explicado antes, como regla general, si la  $L$  es grande conviene permitir una tolerancia mayor y si es pequeña, una tolerancia menor.

### **5.6.- Influencia de la velocidad del robot en el control**

Una variable muy importante que no se ha mencionado es la velocidad del robot. Se ha considerado la velocidad constante en todo el análisis, pero si esta velocidad lineal fuese más rápida, sería prudente aumentar la tolerancia para alcanzar el waypoint.

Respecto al propio control pure-pursuit, existe una restricción cinemática impuesta por los motores del robot. Si existe mucho error, el controlador calculará un giro (accionamiento) excesivo, que el robot puede que no sea capaz de aplicar. En el caso del turtlebot3, el giro máximo es 2.84 rad/s. Si el controlador calcula giros mayores a este valor, habrá una saturación de la señal de control, sacándolo del rango de funcionamiento, y en una zona no válida de control, basándonos en la teoría de control de sistemas lineales.

Teniendo en cuenta esta saturación de señal de control, encontramos una limitación para  $L$  pequeñas.

### 5.7.- Control adaptativo de “L” en el replanificador

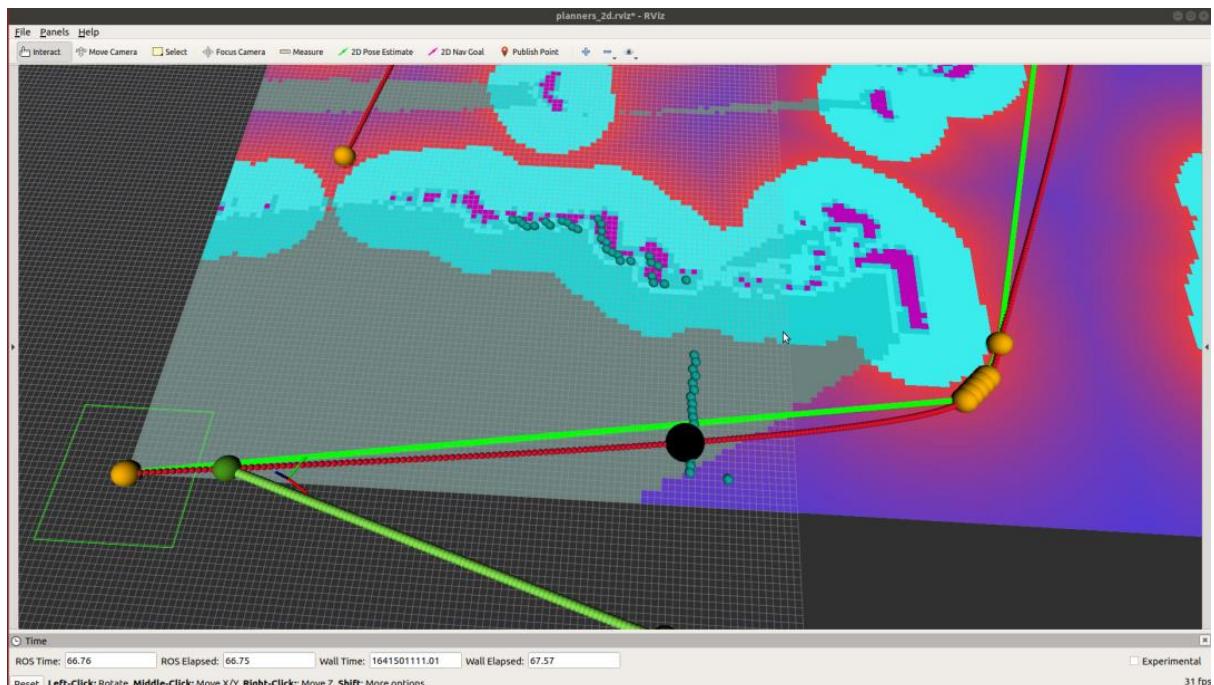
En el replanificador se tiene en cuenta el parámetro L fundamental en cuanto al comportamiento del robot.

Cuando se detecta un obstáculo, para evitarlo, el algoritmo general del replanificador que se ha desarrollado, se basa en buscar una apertura y rodearlo hasta poder volver a la trayectoria inicial.

En el momento de detectar el obstáculo, se produce un cambio brusco de referencia observable en el suministro de waypoints al generar una trayectoria alternativa a la original, por tanto, para obtener un comportamiento más robusto, a precio de permitir más error, conviene aumentar la L, de esta manera, se gira, divergiendo de la trayectoria inicial, sin pedirle una precisión muy alta en el modo 1.

Una L pequeña provocaría volver atrás hasta el primer waypoint replanificado, provocando posibles oscilaciones, incluso movimientos circulares.

Una L grande permite ignorar los puntos que están por detrás, lo cuál nos interesa, es decir nos interesa que el seguimiento no sea exacta al tramo nuevo, queremos que “busque” la nueva trayectoria, a diferencia de recorrerla punto por punto.



**Figura 5.12:** Desviación inicial al detectar un obstáculo con una L grande en el modo 1

Después de la desviación, en el modo 2, se puede considerar que el robot se ha “estabilizado” más sobre esta nueva trayectoria, y por tanto conviene disminuir la L, pidiendo más precisión sobre los nuevos tramos de la trayectoria. Esta nueva trayectoria se ha construido donde no hay obstáculos, y en su vecindad podría existir obstáculos, por tanto, conviene seguir este tramo lo más fielmente posible.

Cuando encuentra un punto de retorno de la trayectoria inicial, similar al modo 1, en el modo 3, existirá otro cambio notable en el seguimiento de waypoints, una discontinuidad en la pendiente debido a la unión de dos trayectorias planificadas independientemente. Por tanto, conviene un control más robusto, permitiendo un margen de error, un error transitorio que luego se corregirá, en este momento crítico de retomar la trayectoria.

Cabe mencionar que retomar la trayectoria fue uno de los problemas más complicados del proyecto, no solo el problema de gestión de waypoints, sino el control robusto adaptando la L que se acaba de explicar.

### **5.8.- Diseño erróneo del Pure Pursuit. ¿Una variación mejor?**

Debido a la importancia que le dimos al algoritmo de control en sí frente al *algoritmo de suministro de waypoints* para un controlador pure pursuit, ya que la mayoría de la información disponible hace hincapié en el funcionamiento del control; el parámetro L, el error que se controla y la señal de control que se aplica. Tomamos una decisión basándonos en nuestra intuición para suministrar los waypoints que vemos interesante comentar. Como el control de bajo nivel es pure-pursuit, no nos percatamos de la influencia y la importancia de la decisión de cómo y cuándo se deben suministrar los puntos objetivos, es decir la condición de conmutación de waypoints.

Basándonos en la suposición que, dada una trayectoria, que es un conjunto de puntos discretos, el controlador debe alcanzar o al menos de forma aproximada, *todos los puntos* que componen la trayectoria. Llegamos a la conclusión, erróneamente, de que cómo dice el nombre, el robot “busca” el punto *sin alcanzarlo*.

Configuramos el algoritmo de suministro de waypoints para recorrer los puntos del vector en orden y secuencialmente, controlando el punto objetivo hasta que estuviera a una distancia L de ella, y entonces enviar el siguiente. De esta manera, recorre todos los puntos de la trayectoria o al menos lo “busca”, manteniendo la resolución de la trayectoria y aprovechando el rango máximo.

Nos percatamos de que el algoritmo pure pursuit no estaba realmente implementado a la hora de tocar el parámetro L. Al variarlo, y considerablemente, no notamos sus efectos en el seguimiento de la trayectoria. Primero concluimos que la razón era que funcionaba demasiado bien porque teníamos demasiados waypoints (interpolando muchos puntos). Y es justamente esta razón por la que funcionaba tan bien y por qué

no era el pure pursuit. Estábamos “forzando” una precisión que a  $L$  grandes no debería tener. Solo pudimos observar el efecto de una  $L$  grande en las curvas, pero este efecto sucede en el pure pursuit real, por tanto, tampoco sonó ninguna alarma. A mayores  $L$ , se pierde resolución de la trayectoria, o se busca antes de alcanzar, provocando un recorrido distinto a la trayectoria. Además, ambos tienen la característica de no poder rodear curvas con  $L$  mucho mayores que la propia curva.

Reflexionamos sobre este diseño, podemos argumentar una ventaja respecto al pure pursuit real.

El pure pursuit real busca un punto a una distancia  $L$ , y después intenta alcanzar este punto, es decir, que la posición del robot sea igual a la del punto. Una vez montado sobre la trayectoria, lo sigue con gran fidelidad, pero consideramos que en nuestra variación **el seguimiento de la trayectoria es más fluida, y aprovecha el rango máximo de waypoints generados.**

Teníamos una implementación que suministraba todos los waypoints de forma secuencial y ordenada, siguiendo una referencia *total* de la trayectoria, el pure pursuit verdadero tiene un seguimiento más discreto o una referencia *submuestreada*, causando mayor error cuadrático medio respecto al seguimiento de la trayectoria completa. En nuestro caso de  $L$  grandes, no ocurría los saltos grandes

Por aclarar esto último, si ponemos un ejemplo en el que tenemos la trayectoria tras interpolar *muestrada* en 3000 waypoints, **la versión errónea tomaba como referencia todos ellos secuencialmente**, realizando un control más exhaustivo respecto al seguimiento a la trayectoria definida por ese conjunto *total* de waypoints.

Por su parte, el algoritmo **Pure Pursuit en su versión “pura”, iría omitiendo cierto conjunto de waypoints a la hora de ir adoptando nuevos puntos objetivo a través de la distancia de lookahead**, de ahí esa referencia *submuestreada* mencionada anteriormente, donde, por ejemplo, según el valor de “ $L$ ”, ocurriría que, al adoptar el siguiente punto objetivo, ignore, por ejemplo, los 10 waypoints intermedios entre el punto objetivo anterior y el próximo punto objetivo que adoptará.

La reducción de referencias respecto a la trayectoria inicial, que aumenta mientras aumente la  $L$ , es similar a un control parcial de la totalidad de la trayectoria, la  $L$  restringe cuánto se permite controlar respecto del conjunto total de puntos, cuantos menos puntos se controlen, menos parecido será a la trayectoria real, esto se repercute en el **error cuadrático medio** que aumenta mientras aumente la  $L$ .

En nuestro caso (la versión errónea), a  $L$  mayores, no perdíamos esta resolución de puntos, y controlamos el total de puntos.

Nuestro diseño tenía un defecto que no era observable. Al pasar al siguiente punto, si está a una distancia  $L$ , no influía en alcanzar el punto final, ya que, aunque no

controlara ninguno de los últimos puntos, llevaba la velocidad lineal constante que se ponía a cero en el momento que la odometría es aproximadamente igual al punto final con una tolerancia.

Aunque nuestro diseño funcionaba bien, proporcionaba un control robusto y un seguimiento de trayectorias fiel, decidimos corregirlo e implementar el pure-pursuit de verdad, tal cual se describe teóricamente, ya que es el objetivo del proyecto.

### **5.9.- Limitación del Pure Pursuit en el final de la trayectoria según “L”**

Una vez implementado el pure pursuit verdadero, observamos un efecto que antes no ocurría, con L grandes no alcanza el punto final. Primero lo consideramos un fallo, un defecto, pero luego concluimos que es característica del propio seguimiento de waypoints del pure pursuit.

Una opción fue dejar este efecto, pero para obtener un control que alcance la referencia, siendo la última referencia el punto final, implementamos un L adaptativo al final de la trayectoria. Si el robot se queda a varios puntos del punto final porque no encuentra ningún punto a una distancia L o mayor, se queda atascado, sin referencia y sin finalizar la trayectoria completa. En este caso, implementamos que la L se cambie a una L pequeña, para así poder finalizar la trayectoria.

Aunque cabe mencionar que al llegar al final de la trayectoria y parar el robot, por la inercia del movimiento, el robot necesita una distancia extra sobre el punto final para detenerse por completo.

### **5.10.- Experimentos con diferentes L y otros parámetros**

- Se adjuntan una serie de enlaces a vídeos de experimentos con valores que permiten un comportamiento adecuado del algoritmo Pure Pursuit (también se ha usado estos valores para los videos que muestran experimentos en otros mapas):

- Primeros tests sobre el algoritmo Pure Pursuit:

<https://drive.google.com/file/d/1Jr1wQuobQGOtHiTRhqkRBvLJxsFtUxF-/view?usp=sharing>

- Test de trayectoria completa junto a evitación de obstáculos:

<https://drive.google.com/file/d/1ub1ssFTyw7eKZ8E4MWrc0cv2vjI80S85/view?usp=sharing>

- Experimento algo más exigente comprobando la funcionalidad del replanificador a la hora de evitar obstáculos:

<https://drive.google.com/file/d/155e1lf4u8hpr1C9ITMYaJXrjwVLTYUpP/view?usp=sharing>

- Experimento con cierta trayectoria diferente y conjunto de obstáculos:

<https://drive.google.com/file/d/1kZDDmNJZOB2c9TaMS9Q1aEnjGxCQJGCW/view?usp=sharing>

- Experimento con trayectoria alternativa y varios obstáculos:

[https://drive.google.com/file/d/1ZT-qtf3LH5JYSANm6xh7\\_mHYBpOyLfp8/view?usp=sharing](https://drive.google.com/file/d/1ZT-qtf3LH5JYSANm6xh7_mHYBpOyLfp8/view?usp=sharing)

- Análisis del comportamiento del algoritmo al variar ciertos parámetros fundamentales:

Para esta sección no adjuntan video ya que ya hay un número elevado de ellos a lo largo de la memoria, simplemente mostraremos gráficamente cómo el robot realiza cierta trayectoria mediante variaciones tanto del parámetro “L”, distancia de Lookahead y “escala\_npoints”, que es un parámetro del interpolador, mediante el cual interpolamos un mayor o menor número de waypoints en un cierto tramo.

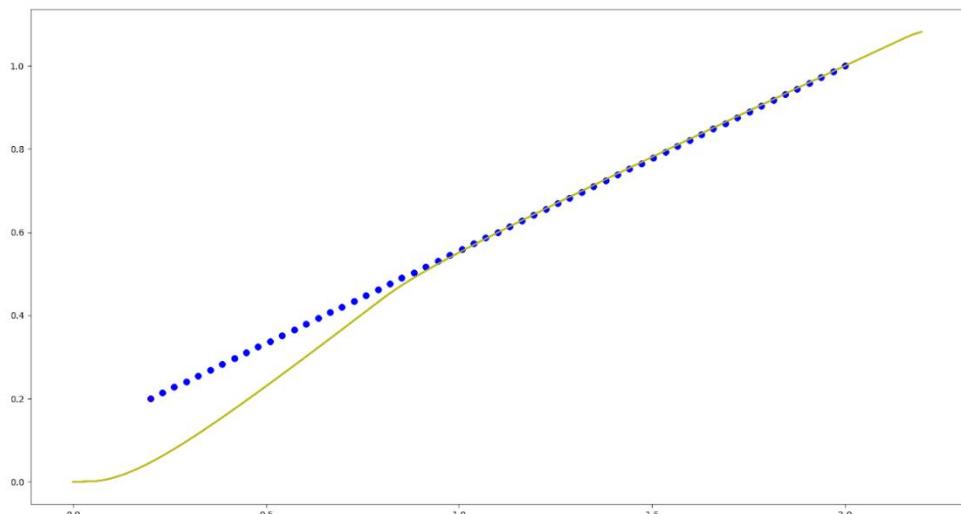
En las gráficas siguientes se representan:

- Puntos azules: waypoints locales interpolados a partir de las globales
- Puntos rojos: waypoints globales (algunos aparecen superpuestas con las locales)
- Línea amarilla: odometría del robot

El eje de abscisas es el eje x, en metros; y el eje de ordenadas es el eje y, en metros.

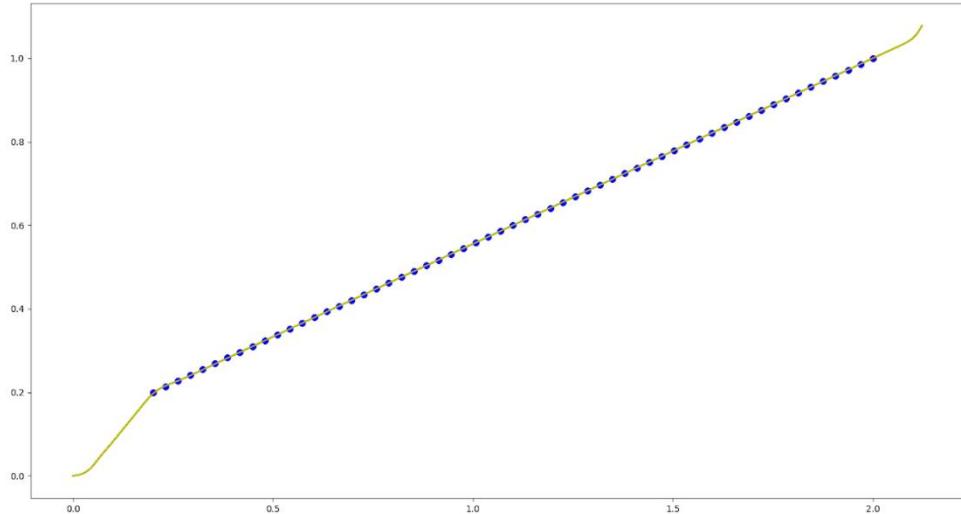
Analizamos el contraste de la referencia de la trayectoria y lo que recorre el robot en la realidad.

- **$L = 1.0 \text{ [m]}$  ;  $\text{Tolerancia\_Wp\_replan} = 0.15 \text{ [m]}$**



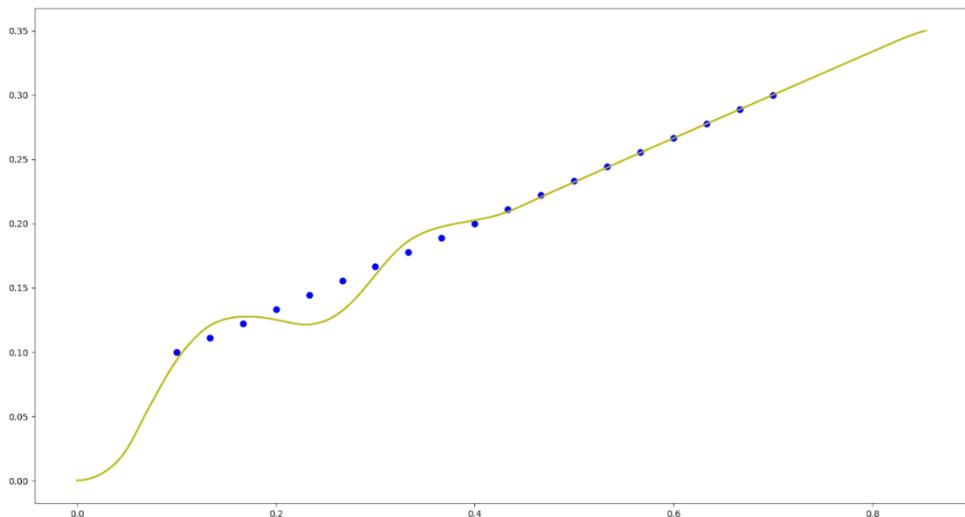
**Figura 5.12:** Comportamiento del algoritmo Pure Pursuit con:  $L=1.0$ ;  $\text{Tolerancia\_Wp\_replan}=0.15$

- **$L = 0.2 \text{ [m]}$  ;  $\text{Tolerancia\_Wp\_replan} = 0.05 \text{ [m]}$**  (Mejores valores adoptados para el seguimiento de trayectorias generalmente, se han usado en la mayoría de vídeos demostrativos, o bien bastante similares).



*Figura 5.13: Comportamiento del algoritmo Pure Pursuit con:  $L=0.2$ ;  $\text{Tolerancia\_Wp\_replan}=0.05$*

- **$L = 0.14 \text{ [m]}$  ;  $\text{Tolerancia\_Wp\_replan} = 0.055 \text{ [m]}$**

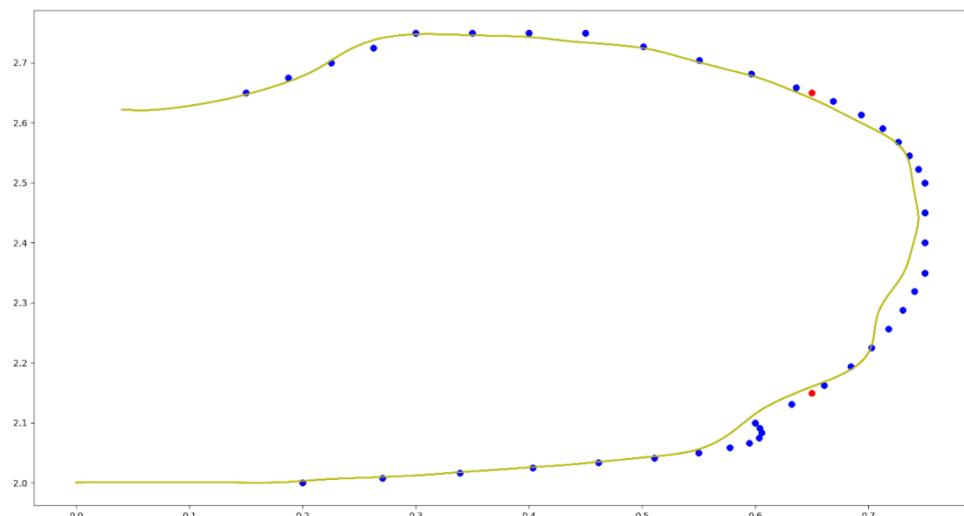


*Figura 5.14: Comportamiento del algoritmo Pure Pursuit con:  $L=0.14$ ;  $\text{Tolerancia\_Wp\_replan}=0.055$*

### Comparación de las figuras 5.12, 5.13, 5.14:

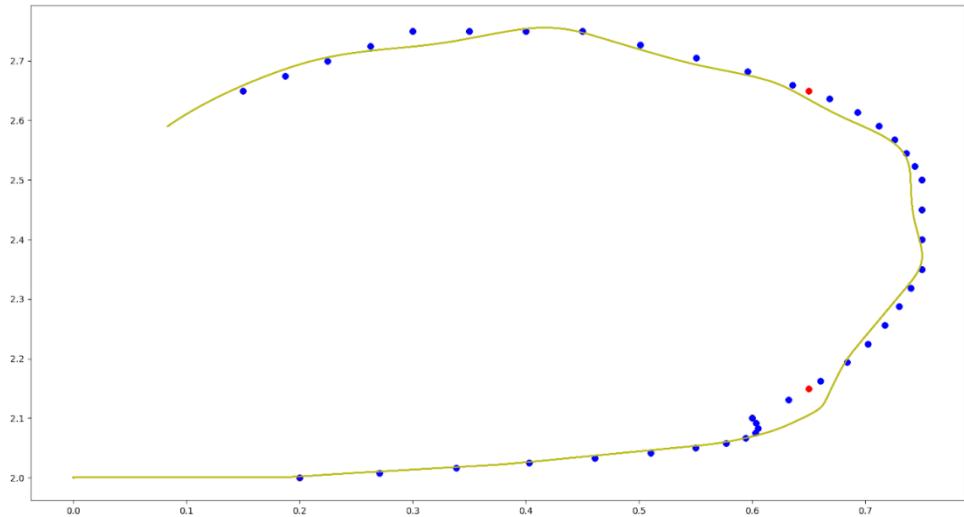
Para un mismo tramo se analiza los distintos comportamientos al variar la L. Observamos que, para una L grande, tiene mayor error cuadrático medio, donde se le aplica un giro conservador, llegando más tarde a la referencia transitoria. Para una L pequeña, aunque tiene menos error cuadrático medio que la L grande, se producen oscilaciones sobre la trayectoria, lo cual puede ser un efecto indeseado. Se busca un término medio entre un control demasiado rápido y exigente, y uno lento y conservador, y obtenemos un control que sigue la trayectoria lo más fielmente posible.

- **L = 0.14 [m] ; Tolerancia\_Wp\_replan = 0.055 [m], para trayectoria curvilínea**



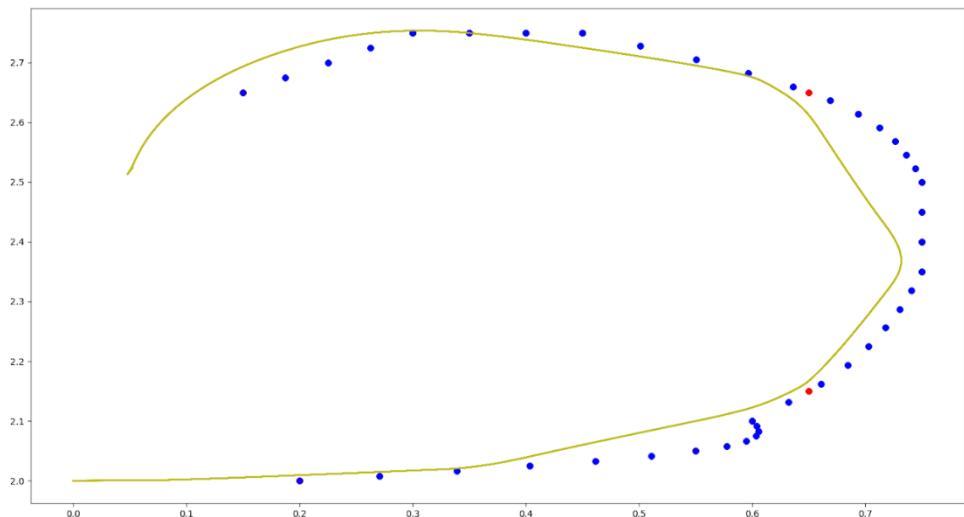
**Figura 5.15:** Comportamiento del algoritmo Pure Pursuit con:  $L=0.14$ ;  $Tolerancia\_Wp\_replan=0.055$ . Caso en que tenemos trayectoria curvilínea

- **$L = 0.2 \text{ [m]}$  ;  $\text{Tolerancia\_Wp\_replan} = 0.05 \text{ [m]}$ , para trayectoria curvilínea**



**Figura 5.16:** Comportamiento del algoritmo Pure Pursuit con:  $L=0.2$ ;  $\text{Tolerancia\_Wp\_replan}=0.05$ . Caso en que tenemos trayectoria curvilínea

- **$L = 0.35 \text{ [m]}$  ;  $\text{Tolerancia\_Wp\_replan} = 0.10 \text{ [m]}$ , para trayectoria curvilínea**



**Figura 5.17:** Comportamiento del algoritmo Pure Pursuit con:  $L=0.35$ ;  $\text{Tolerancia\_Wp\_replan}=0.10$ . Caso en que tenemos trayectoria curvilínea

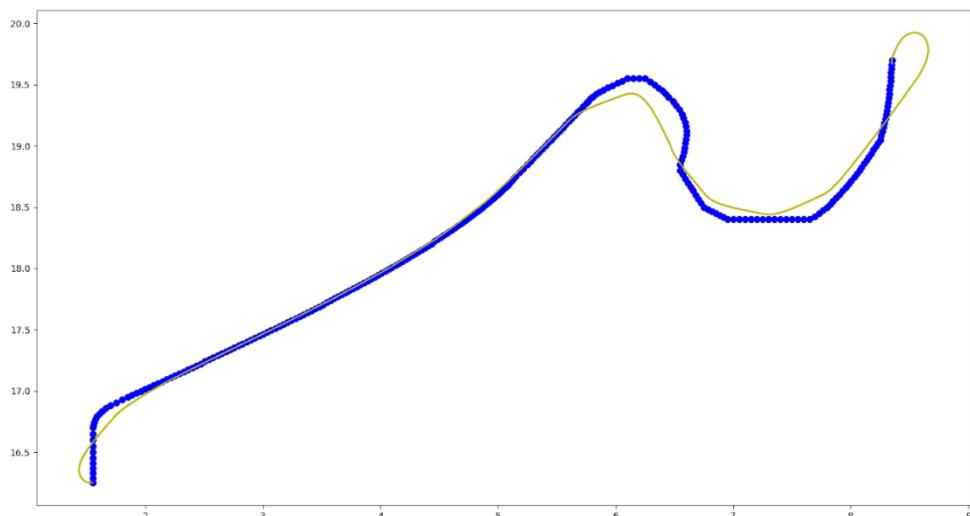
### Comparación de las figuras 5.15, 5.16, 5.17:

Se analiza ahora la variación en el comportamiento del algoritmo al seguir la trayectoria, al ser esta una trayectoria curvilínea, que pone a prueba las “debilidades” de este algoritmo.

Vemos cómo para esta trayectoria concreta, obtenemos un menor error cuadrático medio con valores *pequeños* de L (figuras 5.15 y 5.16), sin embargo, por las características de la trayectoria, los accionamientos agresivos que aporta el tener L pequeña ahora sí son necesarios e incluso convenientes. Se aprecia esto también al tener un cierto mayor valor de L (figura 5.17) como aumenta dicho error cuadrático medio y empeora el seguimiento de la trayectoria curvilínea.

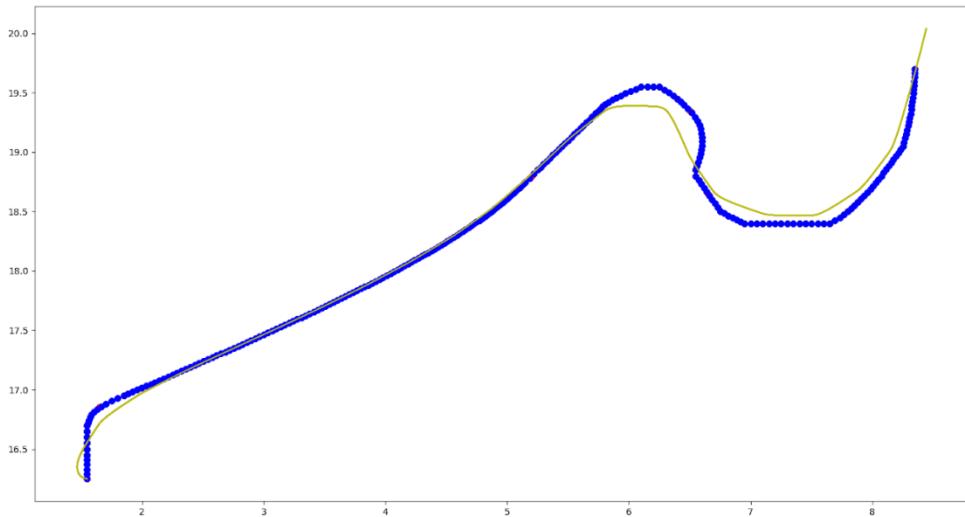
Además de esta serie de gráficas que ilustran la variación en el comportamiento del algoritmo en una trayectoria propuesta simple, se adjuntan también ciertas pruebas sobre una trayectoria más compleja, similar a lo que podría solicitarse en una situación real:

- **L = 0.8 [m] ; Tolerancia\_Wp\_replan = 0.30 [m], para trayectoria “exigente”**



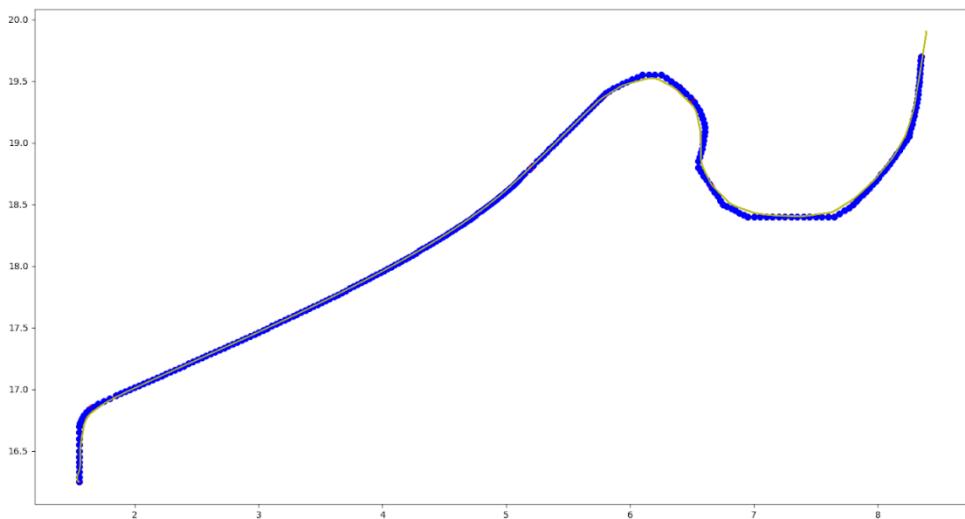
**Figura 5.18:** Comportamiento del algoritmo Pure Pursuit con:  $L=0.8$ ;  $\text{Tolerancia\_Wp\_replan}=0.30$ . Caso en que tenemos trayectoria relativamente “exigente”

- **$L = 0.7 \text{ [m]}$  ;  $\text{Tolerancia\_Wp\_replan} = 0.30 \text{ [m]}$ , para trayectoria “exigente”**



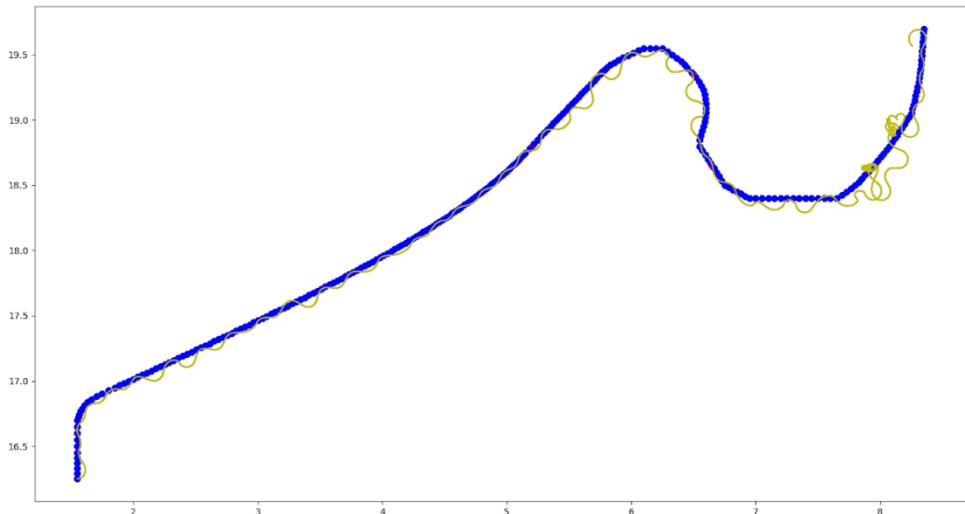
**Figura 5.19:** Comportamiento del algoritmo Pure Pursuit con:  $L=0.7$ ;  $\text{Tolerancia\_Wp\_replan}=0.30$ . Caso en que tenemos trayectoria relativamente “exigente”

- **$L = 0.20 \text{ [m]}$  ;  $\text{Tolerancia\_Wp\_replan} = 0.10 \text{ [m]}$ , para trayectoria “exigente” (parámetros analizados anteriormente como candidatos para obtener los mejores resultados respecto del seguimiento)**



**Figura 5.20:** Comportamiento del algoritmo Pure Pursuit con:  $L=0.20$ ;  $\text{Tolerancia\_Wp\_replan}=0.10$ . Caso en que tenemos trayectoria relativamente “exigente”

- $L = 0.16 \text{ [m]}$  ;  $\text{Tolerancia\_Wp\_replan} = 0.15 \text{ [m]}$ , para trayectoria “exigente”



**Figura 5.21:** Comportamiento del algoritmo Pure Pursuit con:  $L=0.16$ ;  $\text{Tolerancia\_Wp\_replan}=0.15$ . Caso en que tenemos trayectoria relativamente “exigente”

Visto esto, se puede concluir que, para realizar trayectorias complejas y extensas en el espacio, es conveniente encontrar un valor de  $L$  adecuado, así como aprovechar la posibilidad del interpolador de generar gran cantidad de waypoints locales, para obtener así una gran resolución en la trayectoria definida por dichos waypoints que son dados por parte del sistema conjunto planificador + interpolador.

Si expandimos las posibilidades de desarrollo, podríamos pensar en cómo estos resultados abrirían la puerta a la posibilidad de usar  $L$  adaptativo junto a *Machine Learning*, para, por ejemplo, dada una trayectoria fija deseada (un circuito de carreras, por ejemplo) hacer que el robot recorra dicha trayectoria un cierto número de veces para que “optimice” la forma en que la recorre hasta lograr ajustar los valores que lo permiten de manera autónoma.

Esto es sólo un ejemplo y se podrían plantear muchas mejoras que ampliarían la utilidad, robustez y potencia de este algoritmo de control en muchas aplicaciones de robótica.

### 5.11.- Comparación Pure Pursuit – Pseudo PID

Una vez realizados experimentos tanto con el controlador PID del paquete “diff\_drive” y con nuestro controlador Pure Pursuit se llegan a las siguientes conclusiones:

- El hecho de que el control Pure Pursuit suponga velocidad lineal constante, hace que en comparación con el controlador PID, el tiempo que necesita el robot para alcanzar el punto objetivo sea mucho menor. Se adjunta ejemplo de realización de la trayectoria (siendo incluso bastante más corta) por parte del control PID, donde vemos el elevado tiempo que tarda en completarla:  
[https://drive.google.com/file/d/1RNNgiL8YqxlL1s-s\\_ZGxOQNem1ssral3/view?usp=sharing](https://drive.google.com/file/d/1RNNgiL8YqxlL1s-s_ZGxOQNem1ssral3/view?usp=sharing)
- En cuanto al número de parámetros del control, el Pure Pursuit tan solo tiene el parámetro “look-ahead” (L), mientras que el controlador PID tiene tres (K<sub>p</sub>, K<sub>a</sub>, K<sub>b</sub>).
- Debido a que el Pure Pursuit está desarrollado por nosotros, podemos en cualquier momento, modificar las tolerancias que se permiten respecto al error para realizar diferentes pruebas y ver cómo varía el comportamiento del robot. Sin embargo, el “diff\_drive” también ofrece esta posibilidad.
- A diferencia del PID, Pure Pursuit es un método de control que no busca error 0 con respecto al waypoint objetivo, este es un motivo fundamental que hace que el PID tarde más en guiar el robot hasta el destino.

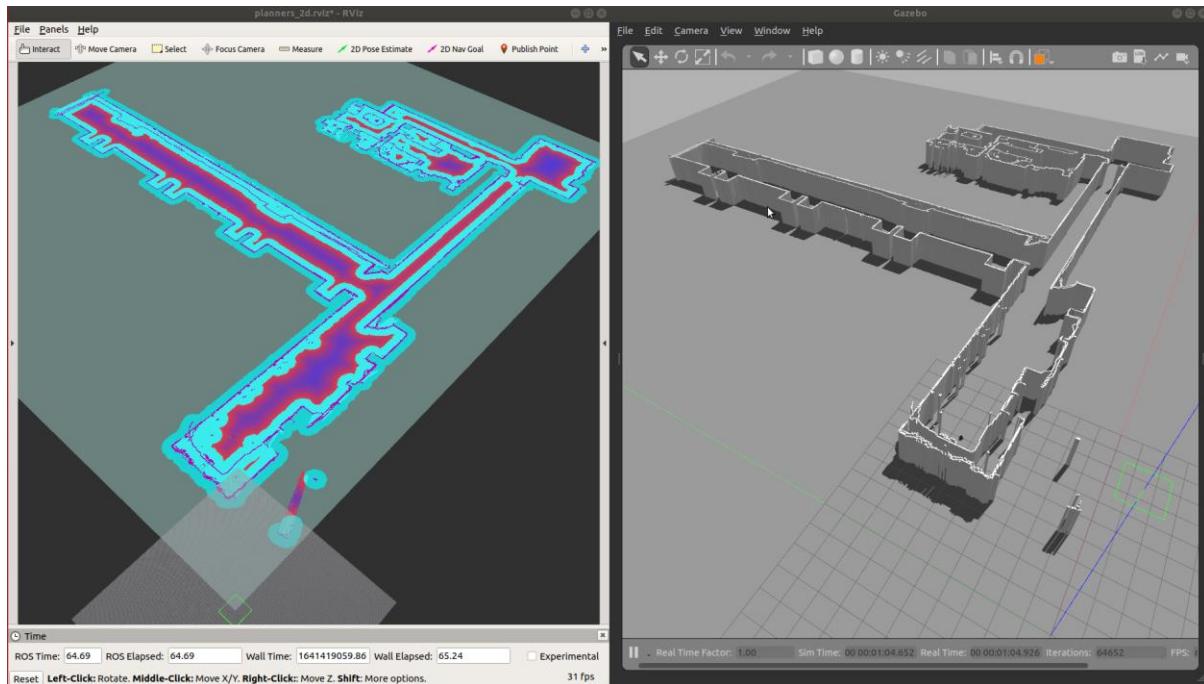
## 6.- Experimentos en varios mapas

### 6.1.- Mapa Alternativo 1

#### 6.1.1.- Test 1

Se ha tomado un mapa alternativo al del resto de ejemplos para hacer pruebas con Pure Pursuit, así como también la parte de evitación de obstáculos.

Para ello el mapa elegido ha sido el siguiente, con su correspondiente mapa de costes:

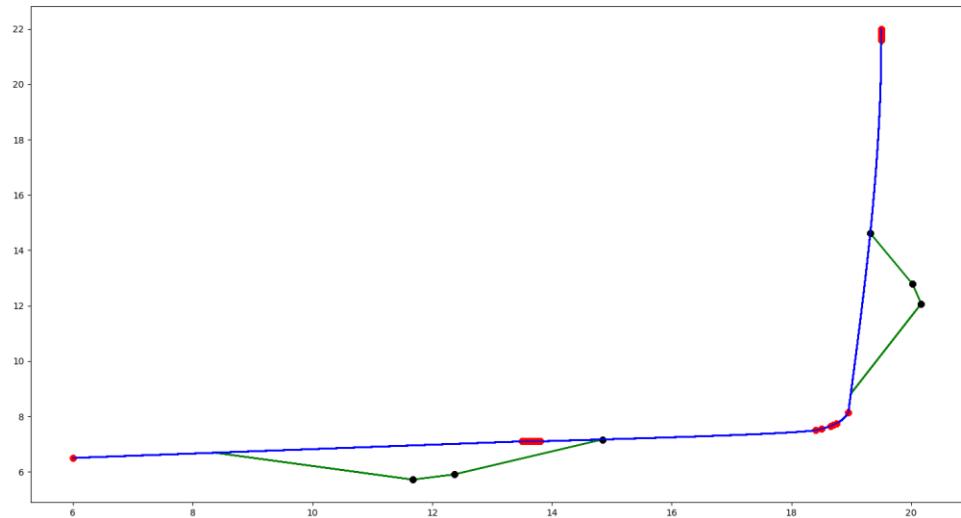


*Figura 6.1: Mapa alternativo para las siguientes pruebas*

Como simplemente se trata de realizar y comprobar los algoritmos en un mapa diferente se adjunta enlace a un video realizando una trayectoria en la que se colocan dos obstáculos que interferirán:

<https://drive.google.com/file/d/1qS-METbHKRBMZsk6rGnEHj2hiX-InQVQ/view?usp=sharing>

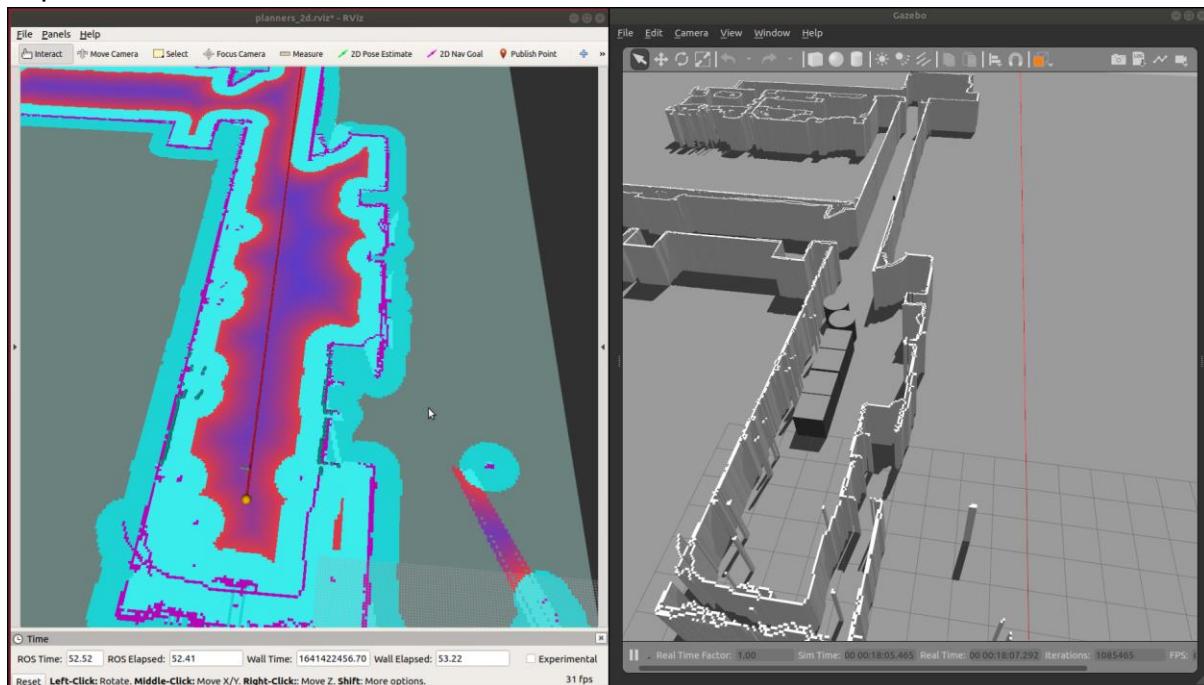
Se adjunta además la gráfica generada de dicho experimento:



**Figura 6.2:** Gráfica correspondiente al experimento anteriormente realizado en el video del enlace anterior. (Eje X: Distancia en m / Eje Y: Distancia en m)

### 6.1.2.- Test 2

En este nuevo experimento, colocamos una serie de obstáculos uno detrás de otro obstruyendo un tramo más extenso de la trayectoria, para comprobar si sorteará todos con éxito y posteriormente se reinsertará a la trayectoria original como se espera.

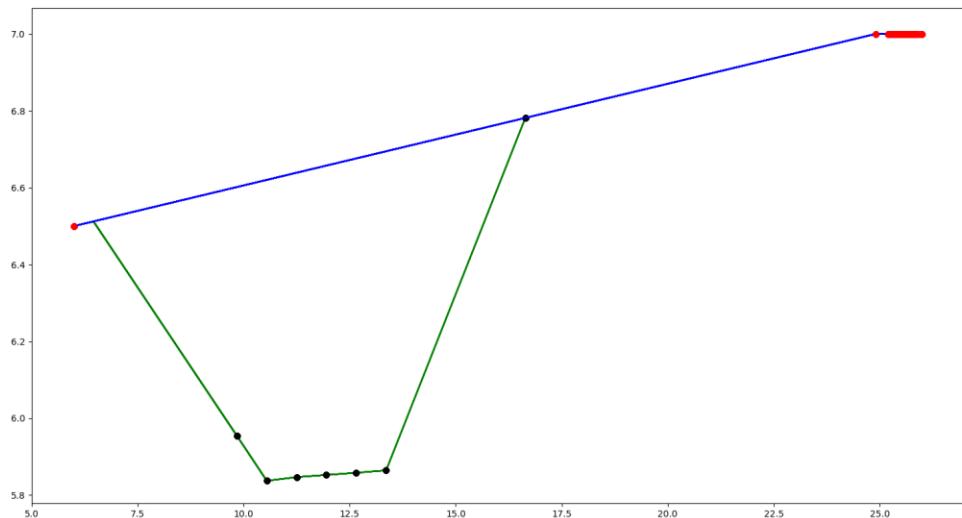


**Figura 6.3:** Experimento con serie de obstáculos planteado para el Test 2

De nuevo, se adjunta un video demostrativo del test para mejor comprensión del experimento:

<https://drive.google.com/file/d/16s1F3CRRoO7jMvvIZfvvqZAxgz26wAN0/view?usp=sharing>

Se adjunta además la gráfica generada de dicho experimento:



**Figura 6.4:** Gráfica correspondiente al experimento anteriormente realizado en el video del enlace anterior. (Eje X: Distancia en m / Eje Y: Distancia en m)

## 6.2.- Mapa Alternativo 2

### 6.2.1.- Test 1

Se ha tomado otro mapa alternativo, que podría equivaler a la estructura de una especie de oficina sin el mobiliario. Se puede ver este mapa en Gazebo, así como su mapa de costes 2D en la siguiente figura:

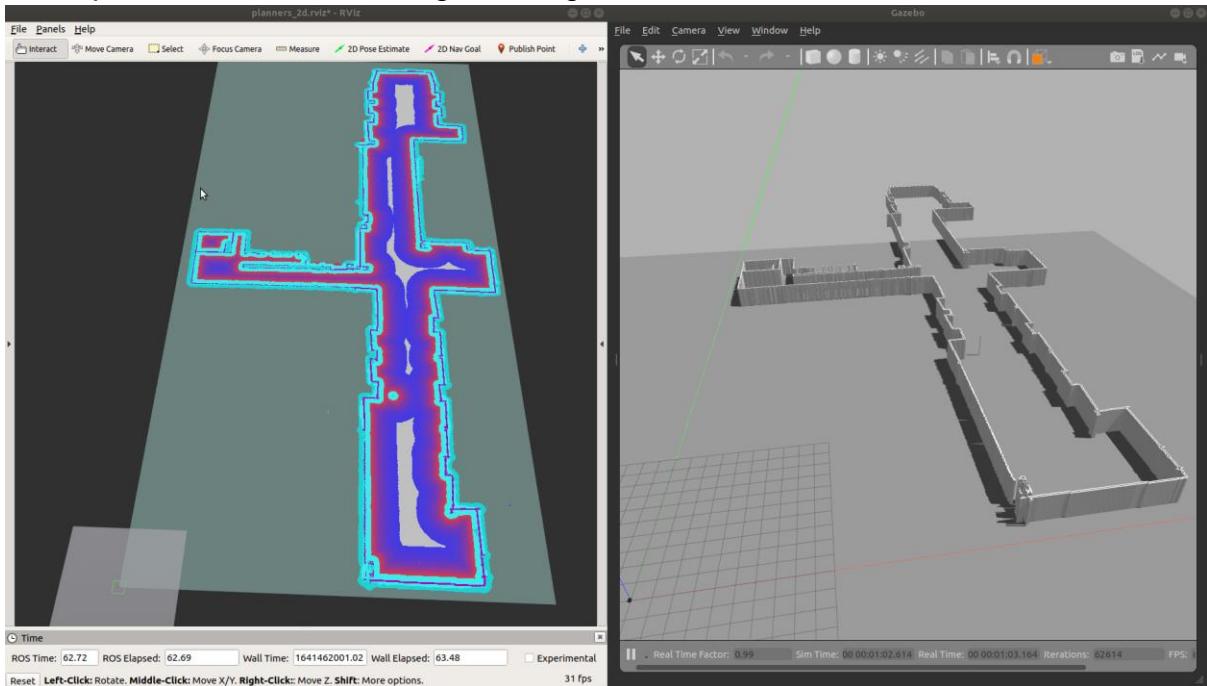
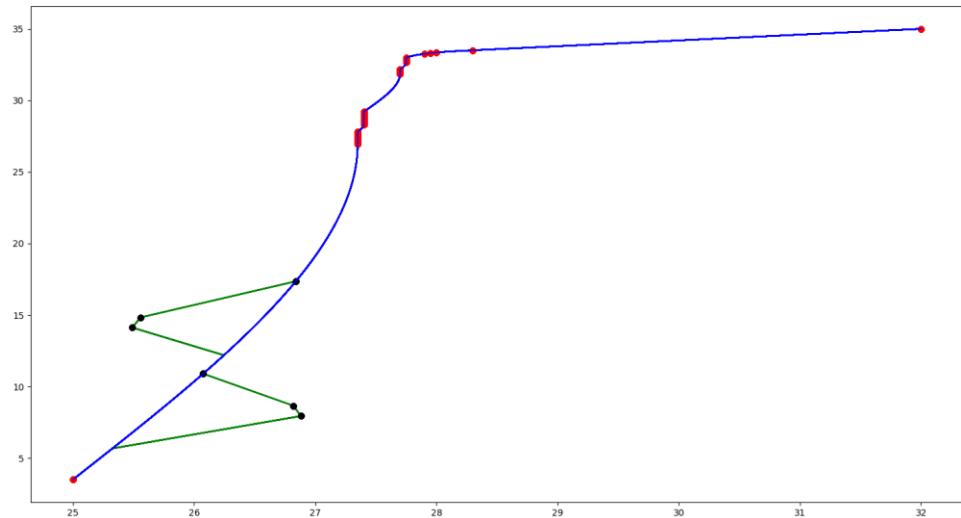


Figura 6.5: Mapa alternativo para las siguientes pruebas

Se adjunta video del primer experimento en este nuevo mapa alternativo, para comprobar la funcionalidad de los algoritmos desarrollados:

[https://drive.google.com/file/d/1Hx2h5akOmpbwDBTVk9TjAc26szZX\\_Im3/view?usp=sharing](https://drive.google.com/file/d/1Hx2h5akOmpbwDBTVk9TjAc26szZX_Im3/view?usp=sharing)

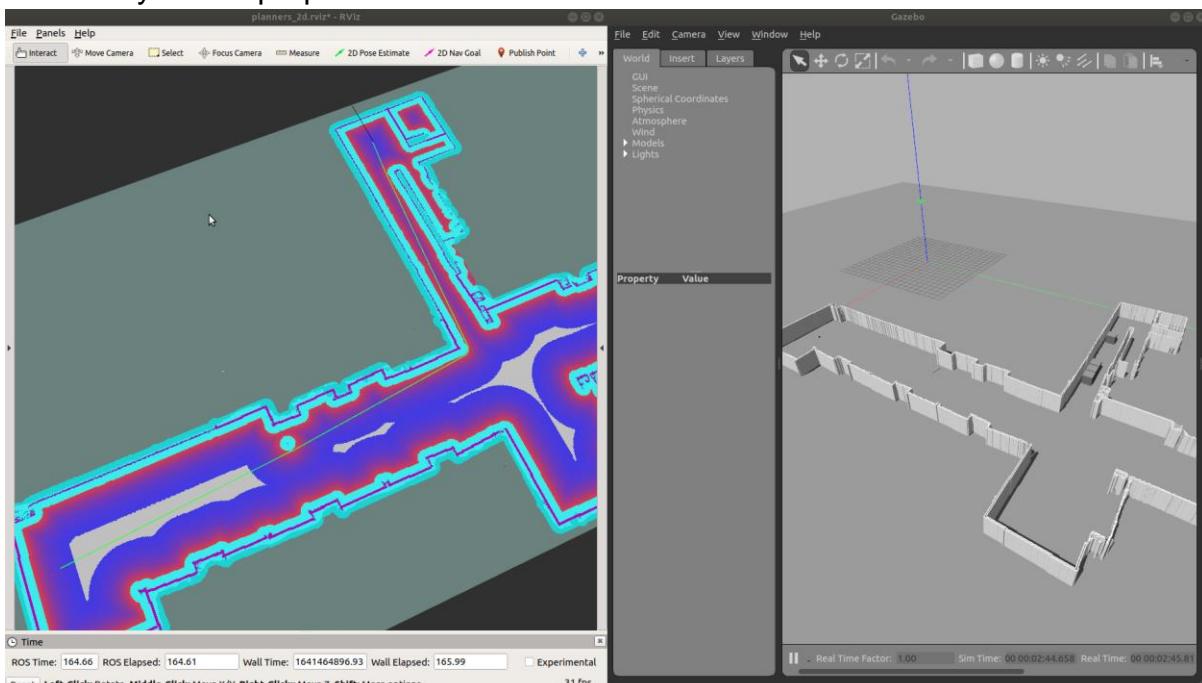
Así mismo se adjunta también la gráfica correspondiente al experimento anterior:



**Figura 6.6:** Gráfica correspondiente al experimento anteriormente realizado en el video del enlace anterior. (Eje X: Distancia en m / Eje Y: Distancia en m)

### 6.2.2.- Test 2

Se realiza ahora un test con una serie de obstáculos consecutivos que interferirán con la trayectoria propuesta:

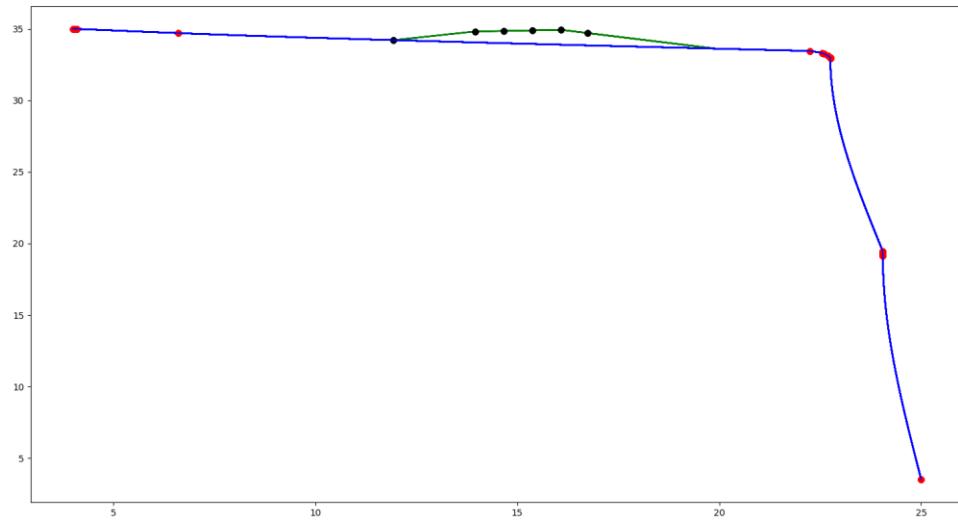


**Figura 6.7:** Segundo experimento propuesto para este mapa alternativo

Se adjunta ahora un enlace al video comprobatorio de dicho experimento:

<https://drive.google.com/file/d/1FMRF-9WEn8xAx7k6KJCz85wSWM26yRR/view?usp=sharing>

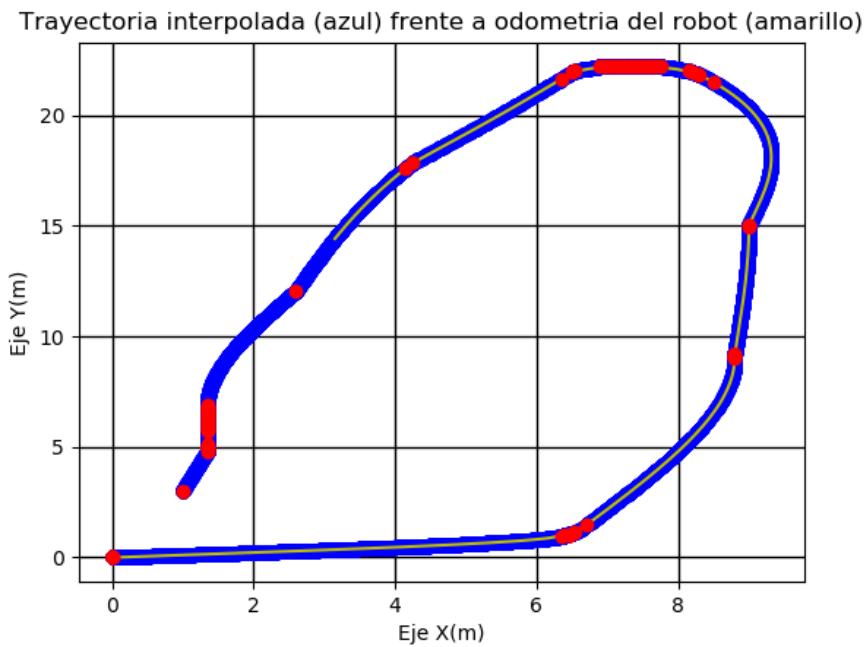
Así mismo se adjunta también la gráfica correspondiente al experimento anterior:



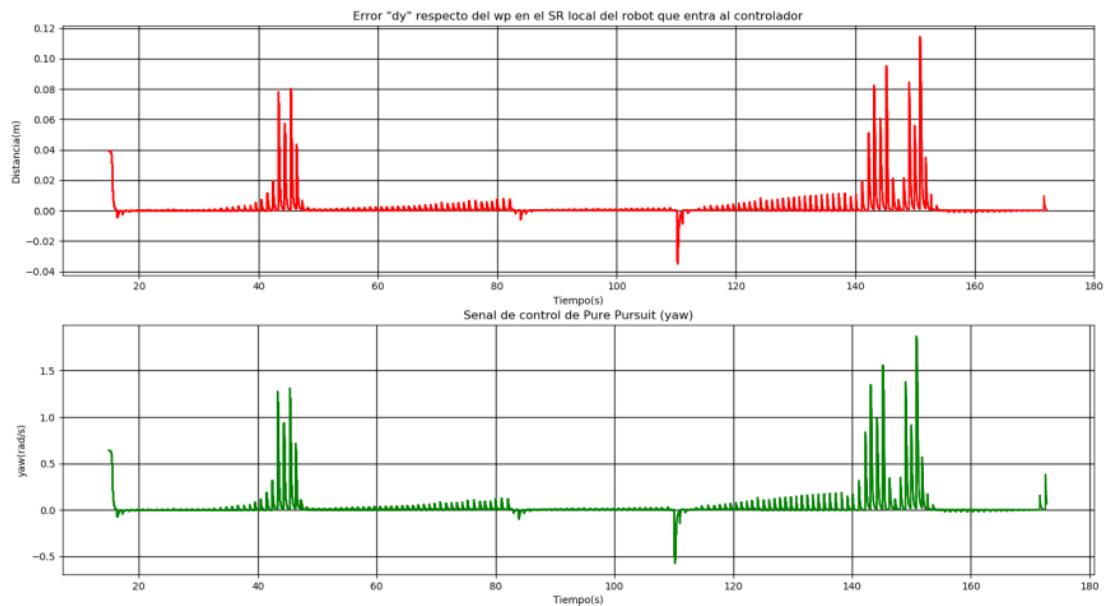
**Figura 6.8:** Gráfica correspondiente al experimento anteriormente realizado en el video del enlace anterior. (Eje X: Distancia en m / Eje Y: Distancia en m)

## 7.- RESULTADOS GRÁFICOS DEL CONTROLADOR Y PLANIFICADOR

### 7.1.- EXPERIMENTO SIN OBSTÁCULO



*Figura 7.1.1: Gráfica de la trayectoria del experimento sin obstáculo*



*Figura 7.1.2: Respuesta del controlador en el seguimiento de la trayectoria*

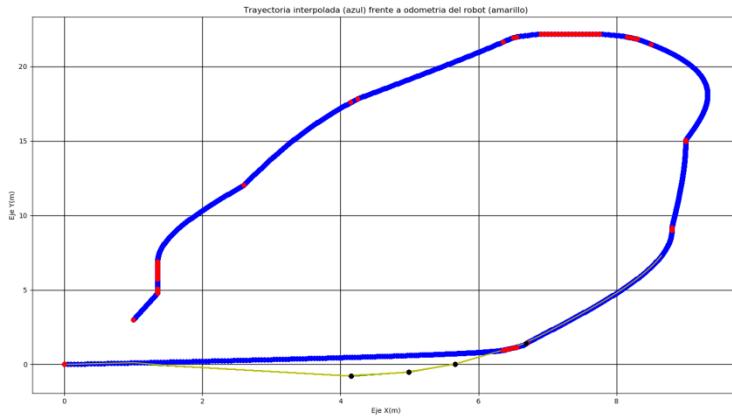
Nos encontramos ante la gráfica de respuesta del controlador, donde arriba se muestra el error, que es la distancia en y desde el punto objetivo hasta la posición del robot, y abajo se muestra la señal de control, que se aplica según el error.

Recordando la expresión que relaciona la señal de control y el error (según nuestro sistema de referencia del robot)

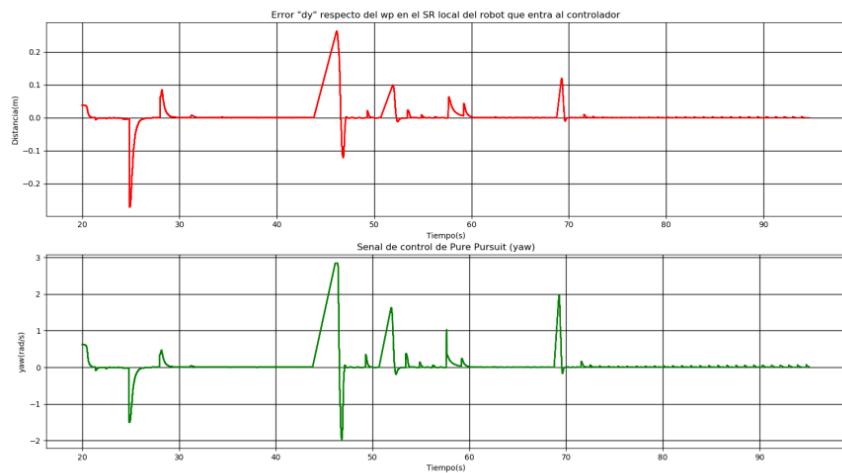
$$yaw = + \frac{\Delta y}{L^2}$$

En la gráfica observamos como yaw y  $\Delta y$  son directamente proporcionales, con signo igual. Se observa que  $\Delta y$  es igual al yaw escalado a una  $K_p = 1/L^2$ , por eso, las gráficas tienen la misma forma.

## 7.2.- EXPERIMENTO CON OBSTÁCULO



*Figura 7.2.1: Detalle de la gráfica de la trayectoria*



*Figura 7.2.2: Detalle de la gráfica del controlador en modo replanificador*

Los picos en la señal de control representan giros notables que aplica el controlador. Respecto a los modos del planificador, se observa que los picos se corresponden con la transición de un modo a otro. A los 25 segundos detecta el obstáculo y entra en modo evitación, y se desvía de la trayectoria (modo 1), resultando en un giro hacia la derecha. A los 45 segundos, empieza a rodear el obstáculo (modo 2), aplicando un giro hacia la izquierda. Finalmente, a los 70 segundos retoma la trayectoria, y el giro se debe a la discontinuidad en el cambio de pendiente de la unión de trayectorias (replanificada y planificada). Las líneas horizontales se deben al recorrido sobre una línea recta, donde no es necesario aplicar un giro, ya que hay un seguimiento fiel de la trayectoria.

## 8.- Dificultades presentadas a lo largo del proyecto

Se comentarán brevemente los problemas destacados durante el desarrollo del proyecto.

### **8.1.- ROS**

El primer contacto para montar el proyecto se podría considerar abrumador, estuvimos probando diferentes paquetes, tanto de ROS como de github, y se tuvo que desmenuzar el proyecto. Merece mención el hecho de que aunque ROS disponga de muchos paquetes y su comunidad sea muy activa en cuanto a desarrollo, como es lógico, es difícil que satisfaga todas las necesidades de un proyecto concreto. Por tanto, lo más común es que un proyecto de este tipo termine consistiendo en un conjunto de paquetes del que se extraen ciertas funcionalidades necesarias, que se combinan con nodos desarrollados íntegramente para cumplir otras funcionalidades respecto del proyecto global.

Aun así, de nuevo, ningún paquete va a venir preparado al 100% para su uso concreto en cierta aplicación. Sino que basándose en lo que proporciona, suele ser necesario tratar su información, reorganizarla para las necesidades del proyecto y, lo más importante, ser capaz de combinarla con los nodos propios que realmente implementan los algoritmos de la aplicación u otros elementos del proyecto.

Es aquí donde se vuelve a comprobar que la robótica es un campo en el cuál las aplicaciones desarrolladas necesariamente se basan en la integración o funcionamiento de múltiples subsistemas que se comunican entre sí.

### **8.2.- Gestionar tipos de mensajes**

Un problema general respecto a la comunicación entre los distintos nodos, fue el formato o el tipo de mensaje que se publica y recibe.

Un ejemplo clásico que se ha presentado en todos los nodos ha sido el mensaje PoseStamped, donde las coordenadas (x,y) solo es una parte del mensaje. Fue necesario realizar cálculos para los cuaternions, incluir el frame\_id, que indica a qué sistema de referencia está referido la coordenada, etc.

Otro problema recurrente y se resolvió después de varias iteraciones fue la gestión de los arrays. Existe la posibilidad de enviar arrays directamente por los tópicos, pero vimos más factible enviar mensajes de un elemento, y en el nodo que suscribe al tópico, dentro de las **funciones callback**, almacena en un vector elemento por elemento.

### 8.3.- Transformaciones

Este problema lo consideramos como el cuello de botella de nuestro proyecto. Hubo un momento en el que teníamos el código del pure pursuit y del replanificador desarrollado, pero ninguno funcionaba ya que ambos dependían de este problema clave. Entendimos la importancia de las transformaciones de un sistema de coordenadas a otros en la robótica, y pudimos observar su ubicuidad en el control de un robot.

Lo primero que hicimos fue crear un tf listener, explicadas en los tutoriales de ROS, pero no funcionaba y encontramos todo tipo de problemas. Pudimos comprobar las transformaciones con el comando echo /tf y sin esta información habría sido imposible solventar el problema.

Uno de ellos fue que tuvimos que darnos cuenta que las transformaciones de ROS tienen un cierto retraso y esto se debe a que /tf necesita recibir varias transformaciones de varios emisores, información que tarda un tiempo en enviarse. Esto se traduce en el código en usar la función dado por ROS que devuelve true, cuando es posible realizar la transformación.

El siguiente, y posiblemente el más crucial fue enlazar el árbol de las transformaciones. El árbol de transformaciones mantiene las relaciones entre los múltiples sistemas de referencias en un árbol.

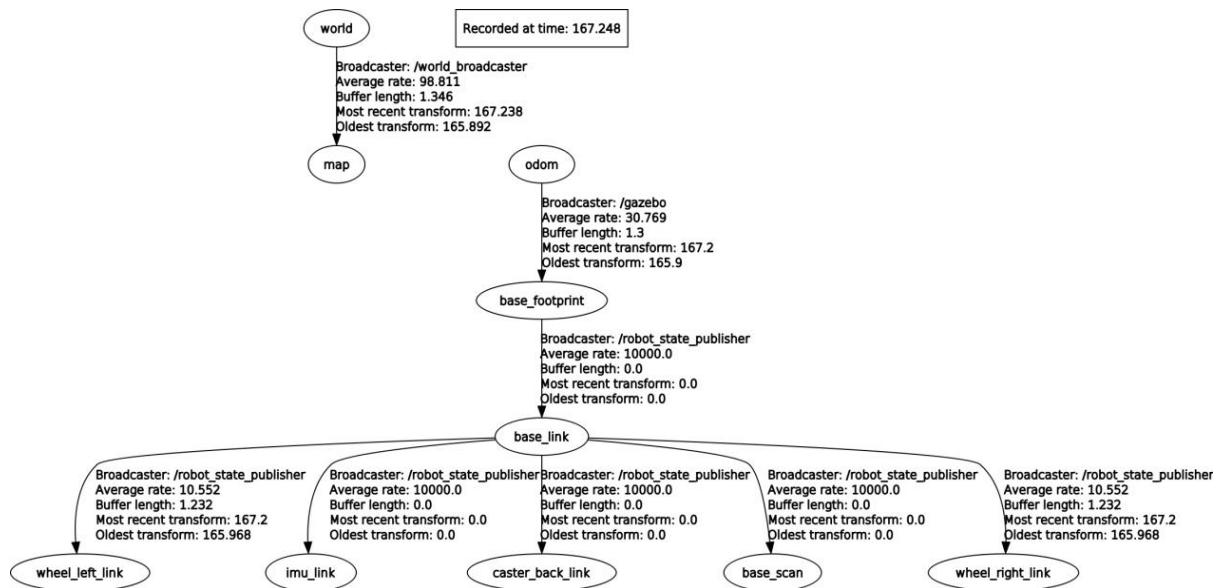
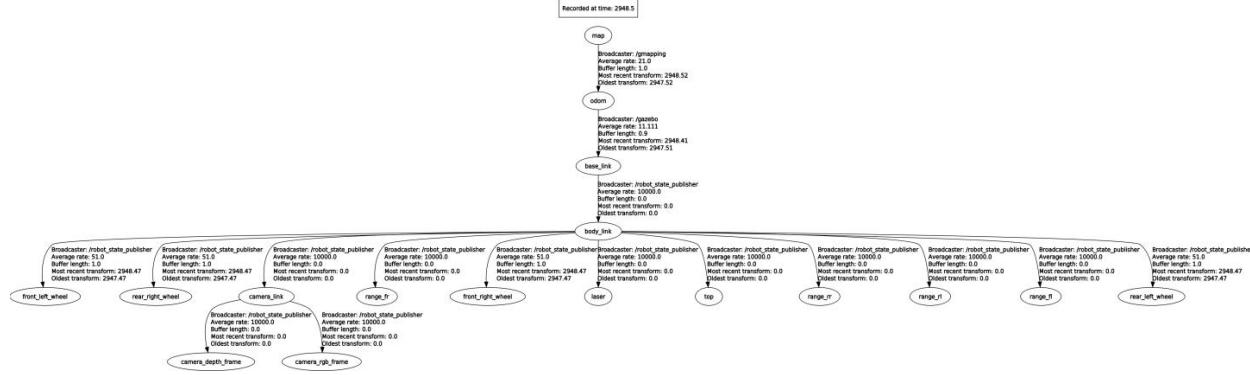


Figura 8.1: Árbol inicial

Inicialmente, como se puede observar, el sistema de referencia del mundo y los pertenecientes al robot estaban desacoplados. Descubrimos que esta fue la razón por la que no funcionaban las transformaciones, y también nos extrañaba que

estuvieran desconectados en el principio. Representamos el árbol de transformación de los programas realizados en las prácticas y comprobamos que tenía el conexiónado que buscábamos, pero desafortunadamente, concluimos que el uso del paquete tf venía usado en el paquete move\_base.

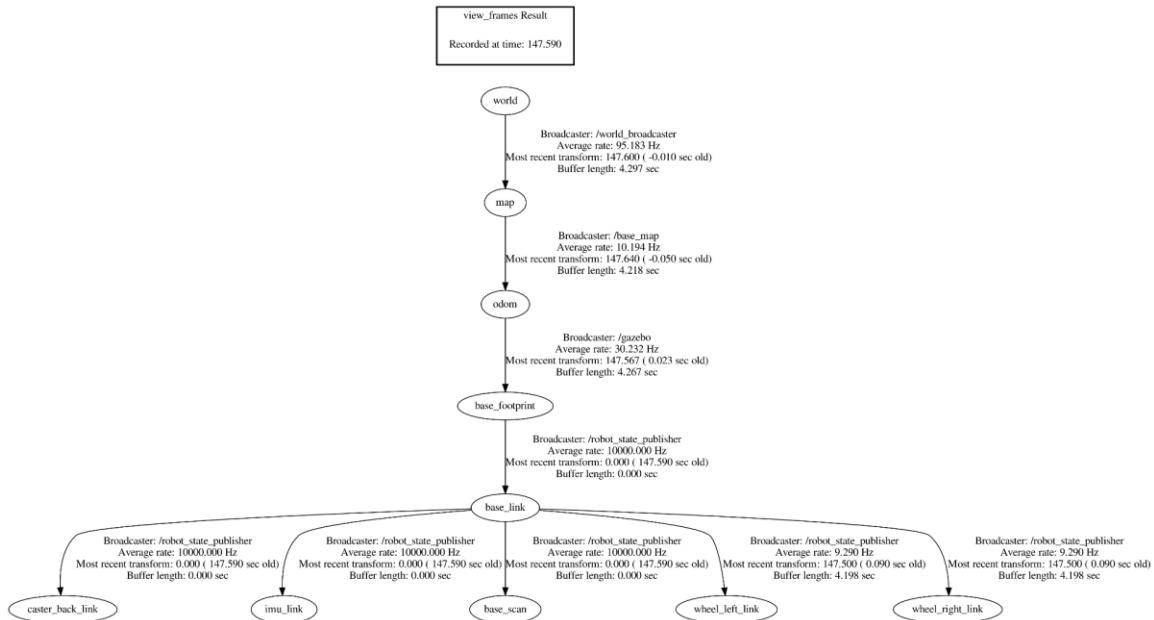


**Figura 8.2: Árbol de las prácticas**

Encontramos la solución y fue muy simple. Había que incluir el paquete tf en el launch, indicando los dos sistemas de referencia a unir, y así modificando el árbol.

```
<node pkg="tf" type="static_transform_publisher" name="base_map" args="0 0 0 0 0 0 1 /map /odom 100" />
```

El árbol resultante es:



**Figura 8.3: Árbol resultante tras usar tf**

Se puede observar que ahora /map y /odom están conectados donde antes no lo estaban. Finalmente, con la función desarrollada anteriormente e incluyendo el paquete en el launch, obtuvimos las transformaciones entre los sistemas de referencia con éxito. Fueron de interés las transformaciones del mundo al robot en el pure pursuit, para transformar las coordenadas de los waypoints referidos al mundo a coordenadas referidas al robot; y del robot al mundo en el replanificador, para transformar las medidas referidas al robot a coordenadas referidas al mundo, para así compararlos con obstáculos, como se ha explicado anteriormente.

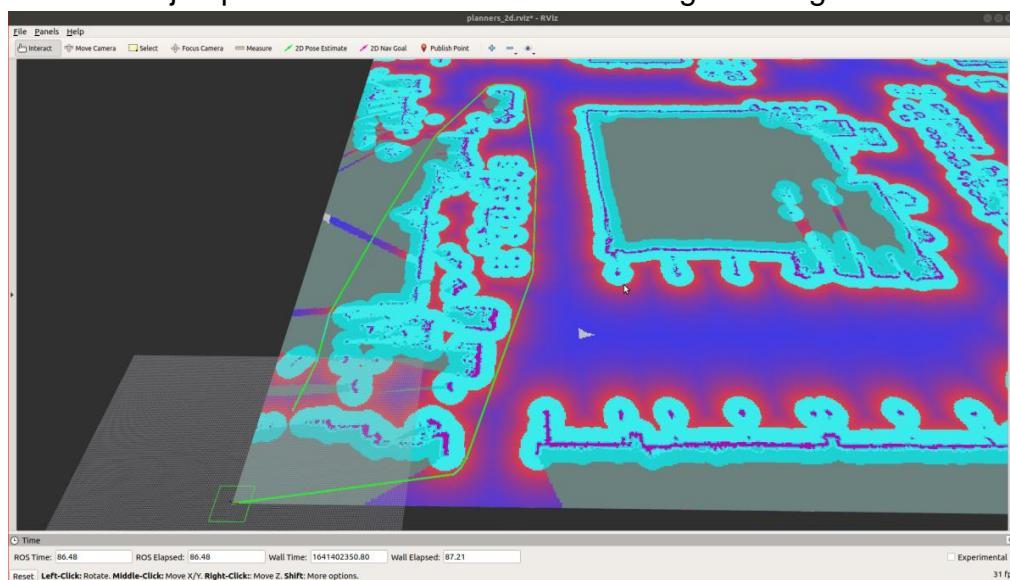
#### 8.4.- Visualizaciones

Como se puede intuir, es imprescindible tener una visualización adecuada de lo que está ocurriendo en la simulación, para ello la herramienta más comúnmente usada en Gazebo/ROS es RViz, un potente simulador gráfico capaz de mostrar diferente información en tiempo real de lo que ocurre en la simulación.

Tenemos dos casos fundamentales donde la visualización obtiene un gran peso a la hora de permitir un desarrollo adecuado y corrección de errores:

##### 8.4.1.- Primeras fases de implementación de planificador e interpolador:

Es totalmente necesario tener una visualización clara de la trayectoria generada, o en su defecto, los waypoints globales que proporcionaba el planificador basado en Lazy Theta \*. Una vez conseguida la visualización de dicho waypoints globales y tras desarrollar el interpolador, se consideró también necesaria obtener la visualización del conjunto total de waypoints locales interpolados, para comprobar que definían correctamente la trayectoria y serían válidos para ser entregados al controlador. Podemos ver un ejemplo de esta visualización en la siguiente figura:

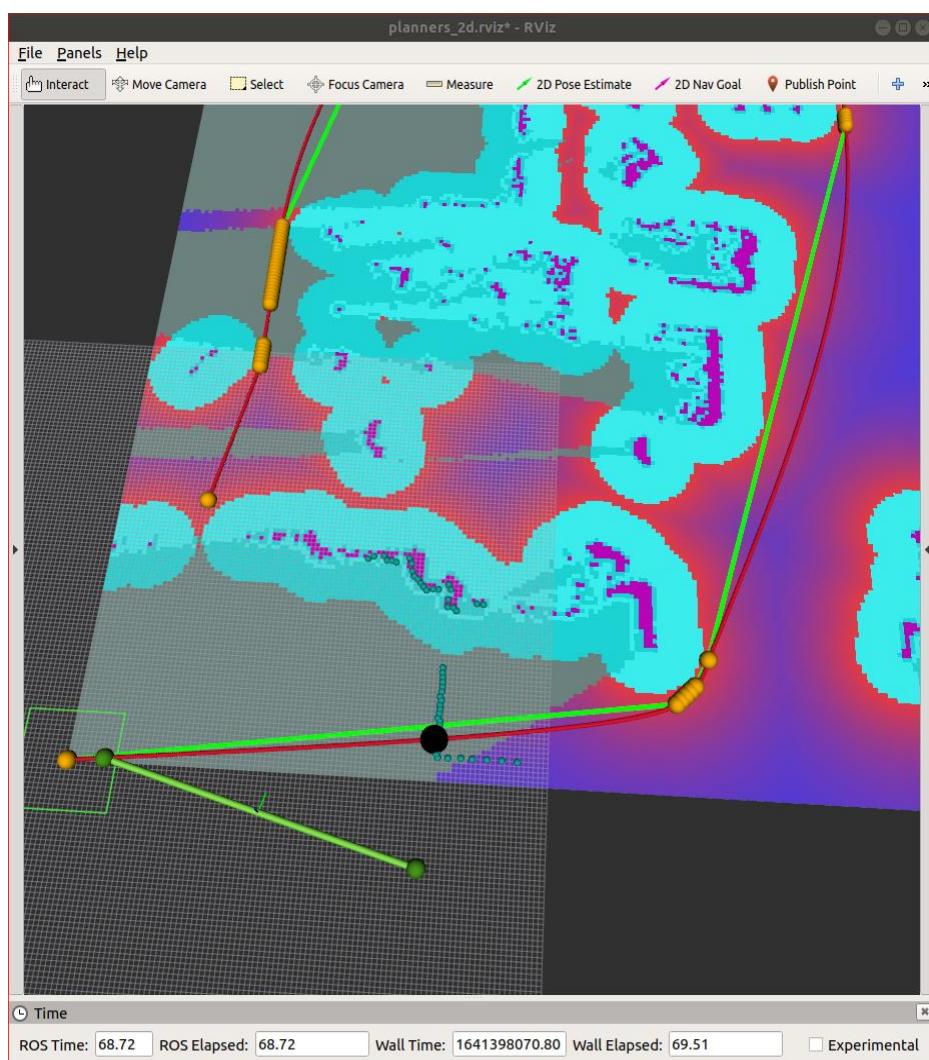


*Figura 8.4: Ejemplo de visualización de waypoints/trayectoria*

#### 8.4.2.- Fases de desarrollo del replanificador:

Por como está desarrollado el planificador, basado en el escaneo del entorno mediante el Lidar, como ya se ha comentado, no bastaba con realizar la visualización del topic `/scan`, sino que era necesario visualizar los puntos que muestran las coordenadas de los obstáculos desde “la vista del robot”, es decir, tras realizar las transformaciones entre sistemas de referencia. Con esto se obtenía una representación más realista de lo que de verdad el robot detecta cómo obstáculo en cada instante.

Recalcar que conseguir visualizar estos elementos representativos de los obstáculos fue clave a la hora de conseguir desarrollar el replanificador, aunque también presentaba ciertos problemas a la hora de que generaba puntos que crecían infinitamente en RViz y finalmente se cerraba el programa. Para esto era necesario resetear el array de “markers” que representaba los puntos obstáculo en cada instante. Podemos ver un ejemplo de esta visualización en la siguiente figura:



**Figura 8.5:** Ejemplo de visualización de los puntos obstáculos que detecta el robot a través del LIDAR, durante el modo replan

## 9.- Conclusión

Durante el desarrollo del trabajo se ha hecho evidente la razón por utilizar ROS. ROS es un marco distribuido de procesos (nodos) que permite que los ejecutables se diseñen individualmente y se acoplen libremente en tiempo de ejecución. Esta posibilidad de poder trabajar en paralelo y desacoplar los elementos para poder trabajar individualmente con ellos, es una situación ideal en la robótica ya que la robótica es el integrador de tecnologías.

Sin embargo, un robot se trata de un sistema complejo compuesto por muchos elementos individuales, y poder trabajar con un nodo por separado como un bloque funcional que recibe entradas y salidas, facilita mucho el trabajo a la hora de diseñar el nodo, y además encontrar los fallos que puedan producirse en el caso de que el sistema global no funcione. Es fácil desacoplar los bloques funcionales y poder trabajar individualmente con ellos y dividir el sistema total en partes.

Respecto a una perspectiva más personal, supone una motivación sumergirse en un entorno de desarrollo tan orientado a la robótica y que como se ha dicho, ofrece tantas posibilidades. Además, no nos arrepentimos, de hecho, todo lo contrario, de haber elegido un robot móvil sobre el que desarrollar un algoritmo como tema del proyecto. ¿Por qué?, realmente no hubiera sido tan gratificante trabajar, por ejemplo, con un brazo robótico, que era una de las opciones que teníamos presente a la hora de elegir, ya que lo habíamos tratado en cursos anteriores y además, termina siendo un sistema muy estático en la industria, estando su desarrollo bastante “estancado”. Sin embargo, trabajar con un robot móvil ha ofrecido la oportunidad de tratar muchos aspectos de robótica desconocidos hasta ahora para nosotros, como son los planificadores de trayectorias (con sus diferentes estrategias de planificación), la detección de obstáculos y estrategias de evitación, así como el propio control de seguimiento de trayectorias, entre otros.

Por tanto, consideramos que un proyecto como este abre mucho más la visión general (muy necesaria para afrontar problemas complejos de resolver) para la resolución de problemas en robótica y la gestión de algoritmos en un campo mucho más “vivo” en la actualidad y, por tanto, con mucha más demanda de investigación y desarrollo.

## 10.- Referencias

- Manual de ROBOTIS para trabajar con el robot *TurtleBot3*:  
<https://emanual.robotis.com/docs/en/platform/turtlebot3/overview/>
- [ ] Información sobre el sensor LIDAR:  
<https://es.wikipedia.org/wiki/LIDAR>
- Descripción conceptual y matemática del algoritmo Pure Pursuit:  
<https://dergipark.org.tr/en/download/article-file/1927194>
- [ ] Pure Pursuit picking a goal point:  
<https://f1tenths-coursekit.readthedocs.io/en/latest/lectures/ModuleD/lecture13.html>
- Paquete “diff\_drive” original sobre el que nos basamos en el nodo “diff\_drive\_go\_to\_goal” que implementa el controlador PID (dados: Pose: (x,y,yaw) y Waypoints: (x,y,yaw), proporciona: velocidades angular y lineal(v,w)):  
[https://github.com/merose/diff\\_drive](https://github.com/merose/diff_drive)
- Información sobre el algoritmo “Lazy Theta\*”:  
[https://www.researchgate.net/publication/220743619\\_Lazy\\_Theta\\_Any-Angle\\_Path\\_Planning\\_and\\_Path\\_Length\\_Analysis\\_in\\_3D](https://www.researchgate.net/publication/220743619_Lazy_Theta_Any-Angle_Path_Planning_and_Path_Length_Analysis_in_3D)
- Vídeo demostrativo del algoritmo “Lazy Theta\*”:  
[https://www.youtube.com/watch?v=mAaYVTedqPQ&ab\\_channel=FahriAliRahman](https://www.youtube.com/watch?v=mAaYVTedqPQ&ab_channel=FahriAliRahman)
- Paquete sin modificar del planificador:  
[https://github.com/robotics-upo/Heuristic\\_path\\_planners](https://github.com/robotics-upo/Heuristic_path_planners)
- Paquete de los mapas usados:  
<https://github.com/shilohc/map2gazebo>
- Información sobre el paquete tf de la wiki de ROS:  
<http://wiki.ros.org/tf>
- Tutoriales de Husarion:  
<https://husarion.com/tutorials/>
- Blog explicativo sobre ROS:  
<https://rostutorial.com/>

- Documentación de ROS explicativa del formato de los mensajes del paquete *geometry\_msgs* (la gran mayoría de los mensajes utilizados para comunicar subsistemas están definidos aquí):

[http://wiki.ros.org/geometry\\_msgs](http://wiki.ros.org/geometry_msgs)

- Ejemplo de búsqueda en una página con ciertos códigos relacionados con ROS en Python, crucial para comprender cómo se trabajaba con ciertos métodos muy concretos de ROS en lenguaje Python (los métodos relacionados frecuentemente eran útiles también):

<https://python.hotexamples.com/es/examples/tf/TransformListener/canTransform/python-transformlistener-cantransform-method-examples.html>

- Plataforma con varios cursos gratuitos básicos sobre ROS, así como Python y C++ aplicado a ROS:

<https://www.theconstructsim.com/>

- Canal de Youtube público y gratuito de la plataforma anterior:

<https://www.youtube.com/c/TheConstruct>

- Blog con descripción bastante detallada del algoritmo Pure Pursuit:

<https://vinesmsuic.github.io/2020/09/29/robotics-purepersuit/#what-is-pure-pursuit>

- Vídeo de una “lectura” de Penn Engineering (University of Pennsylvania) sobre el algoritmo Pure Pursuit:

[https://www.youtube.com/watch?v=r\\_FEKkeN\\_fg&ab\\_channel=Real-TimemLABUPenn](https://www.youtube.com/watch?v=r_FEKkeN_fg&ab_channel=Real-TimemLABUPenn)

- Página web colaborativa de resolución de dudas de ROS:

<https://answers.ros.org/questions/>