

PROYECTO N°16: MODELADO, SIMULACION Y CONTROL DE UN ROBOT MANIPULADOR CON PLATAFORMA MOVIL

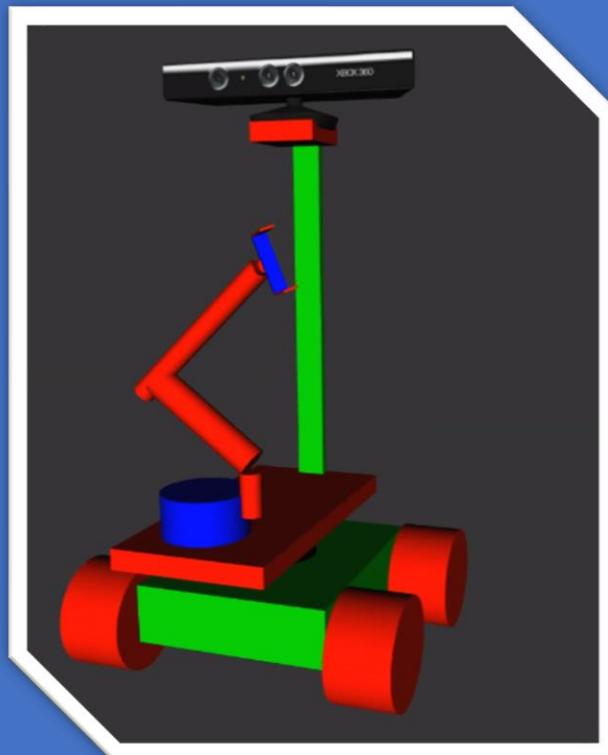


Tabla de contenidos

1. OBJETIVOS DEL PROYECTO.....	2
2. PLATAFORMA MÓVIL	3
2.1. PURE PURSUIT	3
2.2. RQT_GRAPH	4
2.3. RESULTADOS.....	5
2.3.1. MODIFICACIÓN DE LAS VELOCIDADES Y LOOKAHEAD DISTANCE	5
2.3.2. PRUEBA 1: OSCILACIONES	5
2.3.3. PRUEBA 2: OSCILACIONES A FRECUENCIA BAJA	5
2.3.4. PRUEBA 3: LOOKAHEAD DISTANCE DEMASIADO GRANDE	6
2.3.5. PRUEBA 4: LOOKAHEAD DISTANCE Y VELOCIDADES GRANDES	6
2.3.6. RUIDO GAUSSIANO EN LA POSICION EN EL PURE PURSUIT	7
3. BRAZO MANIPULADOR	8
3.1. CALCULO DEL MODELO DINAMICO POR PAR COMPUTADO	8
3.2. DESARROLLO DEL CONTROLADOR POR PAR COMPUTADO EN ROS.....	8
3.3. RQT_GRAPH	10
3.4. RESULTADOS.....	11
3.4.1. Comparación distintos parámetros de Par computado	11
3.4.2. Comparación controladores PI frente a Par computado	15
4. PROBLEMAS ENCONTRADOS	17
4.1. MAPA GIRADO.....	17
4.2. PROBLEMA 2.....	17
4.3. PROBLEMA PURE PURSUIT	17
4.4. PROBLEMA AL RECIBIR LOS WAYPOINTS DEL PAR COMPUTADO	17
4.5. INTENTO DE PLANIFICADOR DE TRAYECTORIAS.....	18
5. CONCLUSION.....	18
6. REFERENCIAS	19

1. OBJETIVOS DEL PROYECTO

El objetivo del proyecto es investigar y desarrollar los controladores específicos de un robot móvil con un brazo manipulador. Para ello, se han valorado diferentes métodos de control, llegando a la conclusión de implementar un Pure Pursuit para la plataforma móvil y un Par Computado para el brazo manipulador.

En un principio el equipo, para conseguir todos los objetivos que abarca el proyecto, se dividió en dos para avanzar de forma paralela, puesto que se ha tenido que estudiar como programar en C++, así como el hecho de realizar los cálculos de ambos controles.

En primera instancia, el equipo de Lisa y Guido empezó buscando un modelo completo para el robot, pero que al mismo tiempo estuviese a la altura de nuestros conocimientos de ROS.

El estudio profundo del funcionamiento de ROS realizado por dicho primer equipo, dedicándose al entendimiento y control de dos complejos y completos modelos en paralelo, ha permitido seleccionar el mejor de ellos por contener el modelo real tanto de una plataforma móvil, como de un brazo manipulador (el cual consta de 6 GDL).

Seguidamente, dado que faltaba una simulación visual funcional del movimiento del brazo manipulador, se ha averiguado una manera de obtenerla y se ha encontrado la solución en el software *MoveIt!*.

Una vez con el modelo corregido, se ha pasado al segundo equipo toda la información necesaria (masas, inercias, factores de reducción, longitudes, entre otras) para que fuera posible modelar las ecuaciones de ambos controles.

En primer lugar, el equipo de Claudia y Carlos se ha centrado en el cálculo del Par Computado para lo cual se han seguido los siguientes pasos:

- Primero, se han creado varios scripts de Matlab con los que calcular las ecuaciones que modelan la dinámica del brazo de 6 GDL.
- En segundo lugar, se ha creado una simulación de Simulink, con la que comprobar el correcto funcionamiento del Par Computado antes de ser introducido en el modelo de ROS.

Posteriormente, se han obtenido las ecuaciones del Pure Pursuit, cuya dificultad en cálculo es menor que las del Par Computado.

A pesar de la división inicial, la evolución del proyecto ha dado lugar a una reorganización de las tareas. En este punto, al disponer de las ecuaciones de ambos métodos de control y tener un cierto manejo del modelo escogido en ROS, el equipo completo se dispuso a implementar dichos controles dentro del mismo.

Con el objetivo de sustituir el controlador de la plataforma móvil, Guido ha desarrollado un publisher/subscriber con el cual se pueda controlar la velocidad de las ruedas de la plataforma móvil skid-steering, siendo Claudia y Carlos los autores de incorporar a dicho .cpp las ecuaciones propias del Pure Pursuit.

Del mismo modo, Lisa ha programado y configurado un action server capaz de recibir la trayectoria procedente de un client proporcionado por *MoveIt!*, así como creado un plugin capaz de comunicar con Gazebo, permitiendo a Claudia y Carlos incluir, del mismo modo que se hizo con el anterior control, las ecuaciones del Par Computado en dicho .cpp.

Finalmente, se ha conseguido unir ambos métodos de control en una simulación conjunta, aunque desacoplada, del robot.

2. PLATAFORMA MÓVIL

2.1. PURE PURSUIT

Para controlar la plataforma móvil se ha hecho un Pure Pursuit. El modelo ya tenía un controlador de alto nivel, un *ros_controller* capaz de leer los parámetros insertados en un archivo *.yaml*.

Por ello, se ha desconectado el *ros_controller* y se ha hecho un nuevo paquete por el Pure Pursuit.

Este nuevo controlador implementado calcula la velocidad que el robot necesita para cumplir la trayectoria, y la publica en el topic de la velocidad. Para calcular la velocidad, el Pure Pursuit se suscribe a la odometría del robot y al camino publicado por el planificador de movimiento Move Base que utiliza un algoritmo A*.

Suscribe: */odometry/filtered*

Suscribe: */move_base/TrajectoryPlannerROS/global_plan*

Publica: */ommp_velocity_controller/cmd_vel*

Se tiene un vector de waypoints, el cual se haya publicado en el topic */move_base/TrajectoryPlannerROS/global_plan*. Para calcular el punto de intersección, se toma el waypoint que tiene una distancia mayor que la mirada hacia adelante desde el centro del robot (Lookahead), y se dibuja una línea recta que pasa por este waypoint y el anterior.

Posteriormente, se calcula la intersección entre la línea recta obtenida y la circunferencia de radio igual a la distancia Lookahead centrada en el robot.

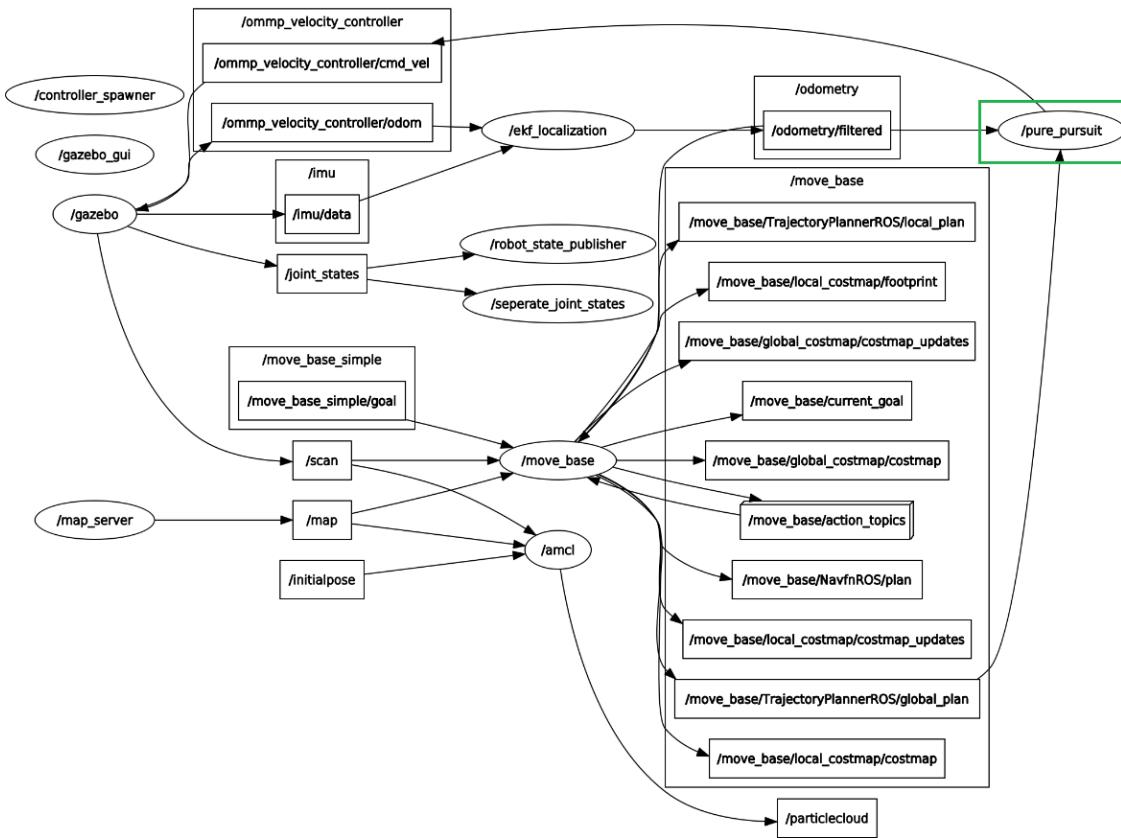
El resultado de esta intersección se usa para calcular la velocidad angular que permite al robot seguir la trayectoria deseada.

El Pure Pursuit consta de varias funciones:

- `void computeVelocity(nav_msgs::Odometry odom)`: recibe en la entrada un mensaje de tipo *nav_msgs/Odometry*, y después, la función calcula una nueva velocidad cada vez que se da una nueva odometría en la entrada. A continuación, se toma la posición actual del robot y se encuentran dos casos:
 - Por un lado, si la distancia entre el robot es menor que la tolerancia, significa que el robot ha alcanzado la meta
 - Por otro, si la distancia es mayor, seguimos calculando la velocidad.
- `void receivePath(nav_msgs::Path path)`: recibe como entrada una ruta de tipo *nav_msgs/Path* llamada “new_path”. Sin embargo, esto lo gestiona el planificador de movimiento, que en nuestro caso es Move Base.
- `KDL::Frame transformPoint(const geometry_msgs::Pose& pose, const geometry_msgs::Transform& tf)`: es una función que genera una posición relativa al sistema de referencia del robot en lugar del sistema de referencia del mapa. Recibe como entrada un mensaje de tipo *geomtry_msgs/Pose* y uno de tipo *geomtry_msgs/Transform*. Esta función se utiliza para saber si el robot ha alcanzado la meta. Toma el Goal, y calcula la distancia entre este y el robot, puesto que la posición del Goal es dada con respecto al mapa. Esta distancia se utiliza para decidir cuándo debe pararse el robot.

2.2. RQT_GRAPH

- El nodo *move_base* recibe la posición deseada desde *2DNavGoal* de *Rviz*, la posición actual del robot móvil desde el topic */odometry/filtered*, y calcula la trayectoria.
- La trayectoria deseada se transmite al controlador */pure_pursuit* mediante el topic */move_base/TrajectoryPlannerROS/global_plan*.
- El controlador publica las velocidades necesarias para cumplir la ruta en el topic */ommp_velocity_controller/cmd_vel*, el cual las envía a Gazebo.
- */ekf_localization* recibe la odometría de Gazebo a través de */ommp_velocity_controller/odom* y, fusionándola con las informaciones obtenidas de */imu/data*, publica la odometría filtrada */odometry/filtered*



2.3. RESULTADOS

2.3.1. MODIFICACIÓN DE LAS VELOCIDADES Y LOOKAHEAD DISTANCE

En este apartado, se va a ver cómo cambia la trayectoria en función de los parámetros principales del Pure Pursuit. Se van a modificar la distancia de Lookahead (el parámetro que decide la lejanía de la intersección con la trayectoria), la velocidad lineal (constante en el Pure Pursuit) y la máxima velocidad angular (límite de la velocidad angular calculada por el Pure Pursuit).

Hay dos objetivos principales:

- Recuperar la ruta
- Mantener la ruta

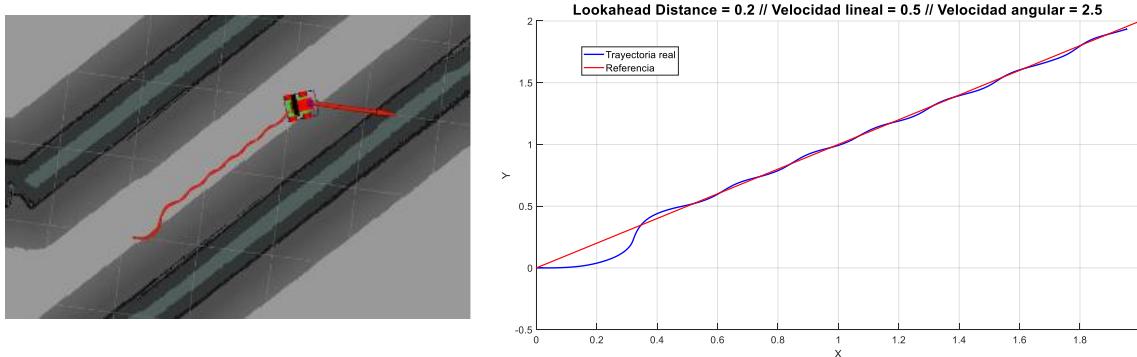
No obstante, el controlador no puede seguir exactamente caminos directos entre waypoints y los parámetros se deben ajustar para optimizar el rendimiento y converger en la ruta a lo largo del tiempo. Otro límite que debe ser ajustado se debe a que este algoritmo no estabiliza al robot en un punto. Para solventar este problema, se debe aplicar un umbral de distancia para la ubicación del robot, de modo que este se detenga cerca del objetivo deseado.

A continuación, se van a hacer una serie de test del controlador en la misma trayectoria, de manera que se pueda visualizar claramente cómo cambia la trayectoria realizada por el robot en función de los parámetros.

2.3.2. PRUEBA 1: OSCILACIONES

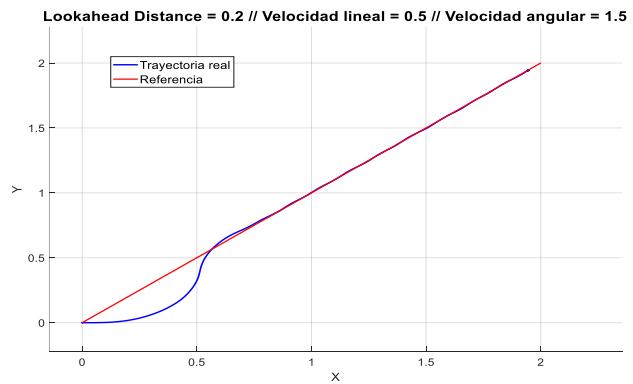
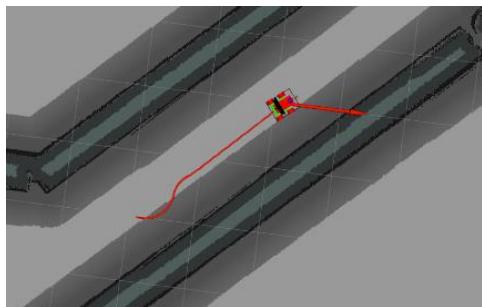
En esta primera prueba, se ha elegido una distancia de Lookahead pequeña, igual a 0.2, una velocidad lineal de 0.5, que es un valor estándar, y una velocidad angular máxima de 2.5, lo cual se trata de un valor alto.

Con una distancia de Lookahead pequeña el robot se mueve rápidamente hacia la ruta. Sin embargo, como se puede ver en la siguiente figura, el robot sobrepasa la trayectoria y oscila a lo largo de la trayectoria deseada.



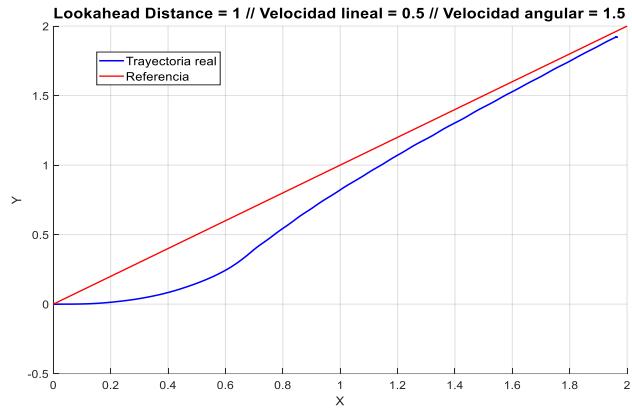
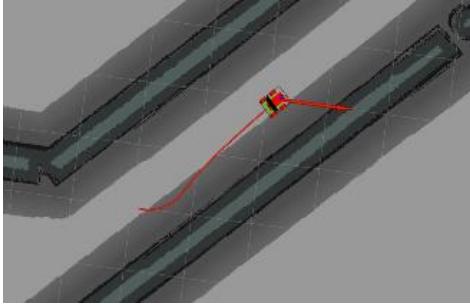
2.3.3. PRUEBA 2: OSCILACIONES A FRECUENCIA BAJA

Con la misma Lookahead Distance (de 0.2) y la misma velocidad lineal (de 0.5), pero bajando la velocidad angular máxima a 1.5, el robot sigue mejor la trayectoria. No obstante existen aún demasiadas oscilaciones, aunque de una frecuencia menor. Esto se debe a que, con una velocidad de rotación máxima más baja, el robot gira más lentamente y sigue la trayectoria con mayor fidelidad.



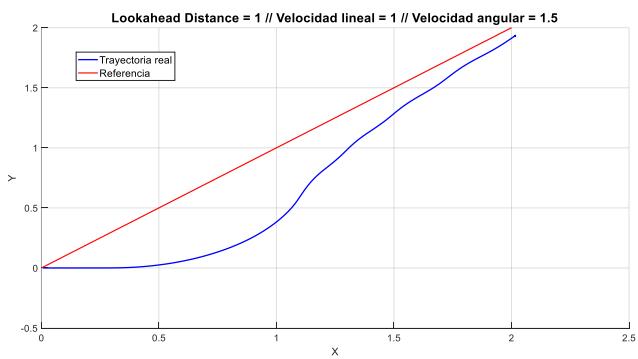
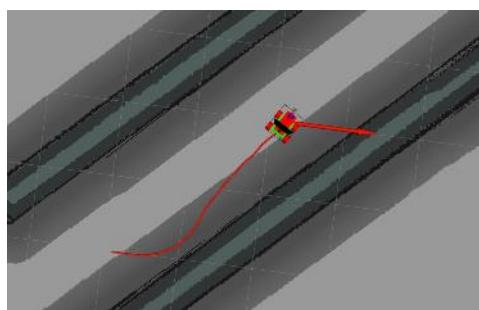
2.3.4. PRUEBA 3: LOOKAHEAD DISTANCE DEMASIADO GRANDE

En la siguiente prueba, se ha tomado una distancia de Lookahead de 1, más grande que la anterior, una velocidad lineal de 0.5 y una velocidad angular máxima de 1.5, siendo valores estándar. En este caso el robot no oscila, pero tarda en acercarse a la trayectoria de referencia, lo cual es lógico dado que la Lookahead es bastante grande.



2.3.5. PRUEBA 4: LOOKAHEAD DISTANCE Y VELOCIDADES GRANDES

Seguidamente, se ha elegido una distancia de Lookahead de 1 y una velocidad lineal de 1, ambas muy grandes, mientras que la velocidad angular máxima permanece a 1.5. En este caso, se puede apreciar como el robot no llega a la referencia en una ruta pequeña.

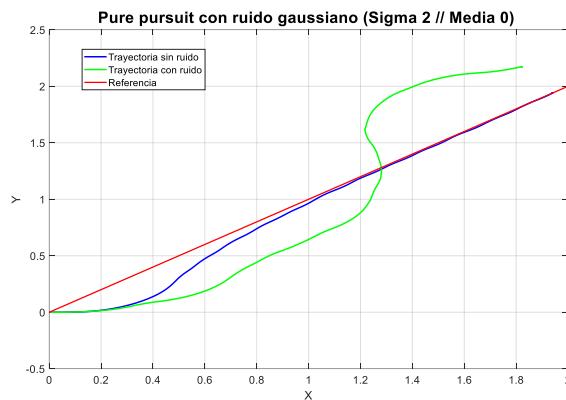
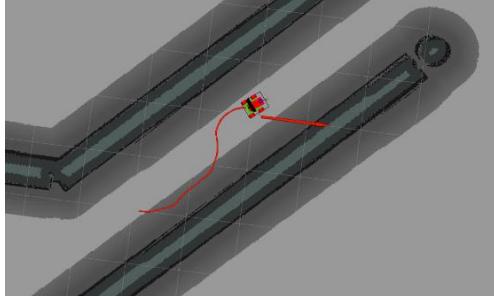


2.3.6. RUIDO GAUSSIANO EN LA POSICION EN EL PURE PURSUIT

En este último apartado, se van a mantener constante los parámetros del Pure Pursuit y, además, se ha añadido un ruido con distribución gaussiana. En todos los test de esta parte se ha utilizado una distancia de Lookahead de 0.5, una velocidad lineal de 0.5, y una velocidad angular máxima de 1.5, siendo estos los parámetros que proporcionan el mejor control que se ha encontrado, lo cual permite al robot seguir la trayectoria bien hasta el Goal.

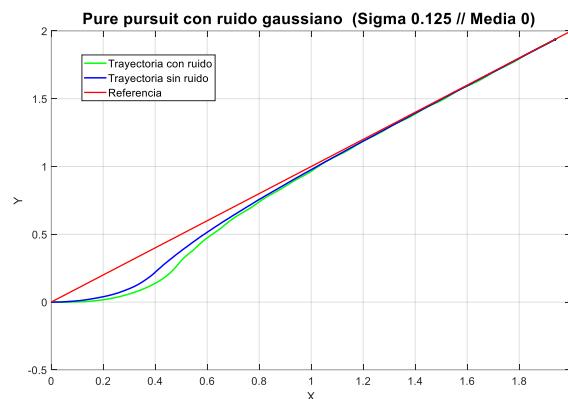
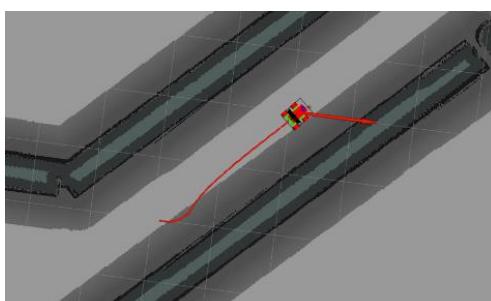
En ambas gráficas, la línea verde representa la ruta en condiciones ideales, mientras que la ruta con ruido está en rojo. Además, tener en cuenta que se ha aplicado un ruido gaussiano distinto en cada eje, tanto en x como en y.

En las siguientes figuras, se puede observar la ruta realizada por la plataforma móvil en caso de añadir un ruido con distribución normal con media en 0 y desviación estándar de 2. Evidentemente, se trata de una exageración y el ruido es demasiado grande, pero de esta forma se puede apreciar con mayor detalle el error que sale. Asimismo, de estas gráficas se deduce que el Pure Pursuit necesita de una posición precisa del robot para funcionar bien.



Por último, se asumen un valor de media nulo y una desviación estándar de 0.125, lo cual son unos errores razonables para la localización. De esta forma, se va a realizar una simulación con un ruido parecido a lo que puede salir en una aplicación real.

Se puede observar como en esta ocasión, con un ruido pequeño, pero razonable, el Pure Pursuit se comporta bien y el error con la trayectoria de referencia acaba siendo prácticamente nulo.



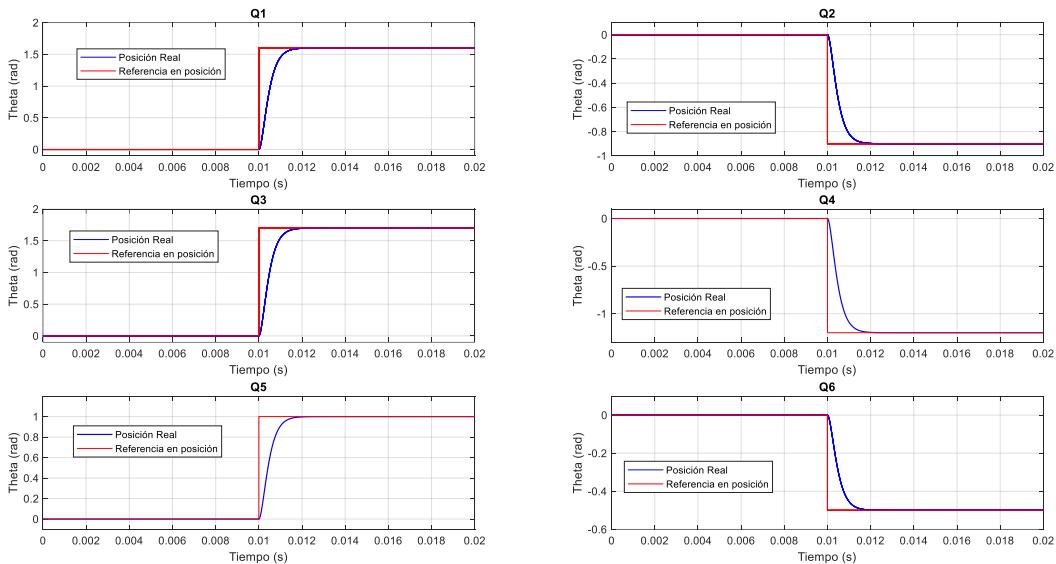
3. BRAZO MANIPULADOR

3.1. CALCULO DEL MODELO DINAMICO POR PAR COMPUTADO

En esta parte, se explica el trabajo realizado sobre el brazo manipulador. Para el diseño del controlador Par Computado se necesita el modelo dinámico del robot. Primero se han obtenido los parámetros de Denavit Hartenberg y, a continuación, se han introducido en un script de Matlab diseñado mediante el método Newton-Euler. De esta forma, ha resultado sencillo calcular la matriz de masas (M), el vector de fuerzas centrífugas y de Colioris (V), y el vector de términos de gravedad. Se ha considerado razonable despreciar el vector de componentes de fricción.

Para la obtención final de los pares que salen de aplicar este control, tan solo queda el diseño de las matrices diagonales K_p y K_v (parámetros de control). Esto se realizará más adelante con el modelo funcionando.

Antes de introducir las ecuaciones en ROS, se ha realizado una comprobación en Simulink, poniendo los valores de las diagonales de las matrices mencionadas a 1. En esta comprobación teórica se visualiza si el control sigue la respuesta ante escalón (para ello se ha utilizado la configuración del brazo denominada “Lisa”). En la siguiente gráfica, se puede demostrar el seguimiento de la trayectoria de todas las articulaciones. Tener en cuenta, que el tiempo de subida resultante que se ha considerado en dicha simulación ha sido de 0.01s (utilizado para calcular K_p y K_v).



Puesto que se ha demostrado el correcto funcionamiento, se ha supuesto que el controlador estaba listo para ser introducido en el modelo de ROS.

3.2. DESARROLLO DEL CONTROLADOR POR PAR COMPUTADO EN ROS

El primer paso fue la configuración de *MoveIt!*, un software de código abierto para ROS, lo cual proporciona a los desarrolladores la posibilidad de interaccionar con la interfaz gráfica *Rviz* y, mediante el nodo *move_group*, permite llevar a cabo tareas de planificación de trayectorias, control, etc... Mediante el *MotionPlanning* de *Rviz*, se pueden pasar puntos a los que se quiere mover el manipulador, tras lo cual *MoveIt!* calcula la trayectoria y la pasa al controlador.

Por lo tanto, se ha convertido el file .xacro, donde se encuentra el modelo del robot, en un file .urdf. Asimismo, se ha configurado *MoveIt!* mediante *moveit_setup_assistant*, de manera que se ha

obtenido el paquete (*ommp_lisa_moveit_config*) que contiene los archivos .launch y .yaml requeridos por *Rviz*.

```
> rosrun moveit_setup_assistant setup_assistant.launch
```

De esta forma, se ha conseguido introducir un planificador RRT (*Rapidly Exploring Random Trees*) para el brazo y, con el objetivo de comprobar el funcionamiento de la simulación del brazo, se ha utilizado un controlador propio de ROS.

```
> type : effort_controllers/JointTrajectoryController
```

Es decir, un PID cuyos parámetros proporcional, integral y derivativo se pueden definir en el *ros_controllers.yaml*. Esto ha sido útil para realizar un *tuning* de dichos parámetros de forma sencilla. Tras recibir la trayectoria calculada por *MoveIt!* bajo la forma de *waypoints* y la posición actual de cada articulación del brazo (publicada en tiempo real mediante un mensaje */joint_states* por parte del *joint_state_controller*), el controlador manda el par deseado al *hardware interface*.

Sin embargo, ha sido posible introducir un controlador de tipo *effort_controllers* solo tras cambiar el tipo del actuador de cada articulación de *PositionJointInterface* a *EffortJointInterface*, de modo que pueda recibir un comando de tipo par.

```
<hardwareInterface> hardware_interface/EffortJointInterface </hardwareInterface>
```

Puesto que se tenían los parámetros del PID propio de ROS tuneados anteriormente, se ha comenzado diseñando un archivo .cpp que contiene un controlador PI. Tras comprobar el correcto funcionamiento del nodo del controlador, se han incorporado las ecuaciones del par computado. Por lo tanto, se ha exportado la clase bajo la forma de Plugin al ROS Package System, para que el *controller_manager* pueda cargar el controlador.

```
PLUGINLIB_EXPORT_CLASS(par_computado_ns::ParComputado,  
                        controller_interface::ControllerBase)
```

Dicha clase hereda funciones de la siguiente clase:

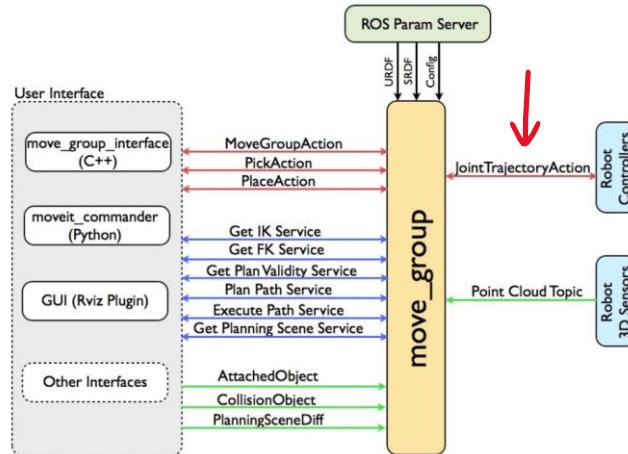
```
controller_interface::Controller <hardware_interface::EffortJointInterface>
```

Más concretamente, se ha hecho uso de las siguientes funciones:

- `bool init(hardware_interface::EffortJointInterface *robot, ros::NodeHandle &n):` función llamada para arrancar el controlador desde un *non-realtime thread* con un puntero hacia el *hardware interface*. Aquí se reciben los nombres de las articulaciones que se quieren controlar (`elbow_joint`, `shoulder_lift_joint`, `shoulder_pan_joint`, `wrist_1_joint`, `wrist_2_joint`, `wrist_3_joint`): esta información está incluida en el *ros_controllers.yaml*. Además, se arranca el servidor de acciones, cuyo propósito se explicará más adelante.
- `void starting(const ros::Time &time):` función llamada justo antes de la primera llamada al `update`. Aquí se inicializa el controlador.
- `void update(const ros::Time &time, const ros::Duration &period):` función llamada a intervalos regulares. Aquí se encuentra el código del controlador: tras recibir la referencia y la posición actual, a través de las ecuaciones del par computado, se calculan y se emiten los pares necesarios para cumplir la trayectoria deseada.
- `double getPosition():` función que proporciona la posición actual de cada articulación.
- `double getVelocity():` función que proporciona la velocidad actual de cada articulación.
- `void setCommand(double pos_target):` función que envía los comandos hacia los *joints*.

Dado que, los resultados de la planificación de *MoveIt!* se publican en forma de acción, nuestro controlador *par_computado* debe proporcionar un servidor de acciones de tipo

`control_msgs/FollowJointTrajectory`. Esta función sirve para recibir los resultados de la planificación de trayectoria del nodo `move_group` y pasarlo como referencia al controlador.

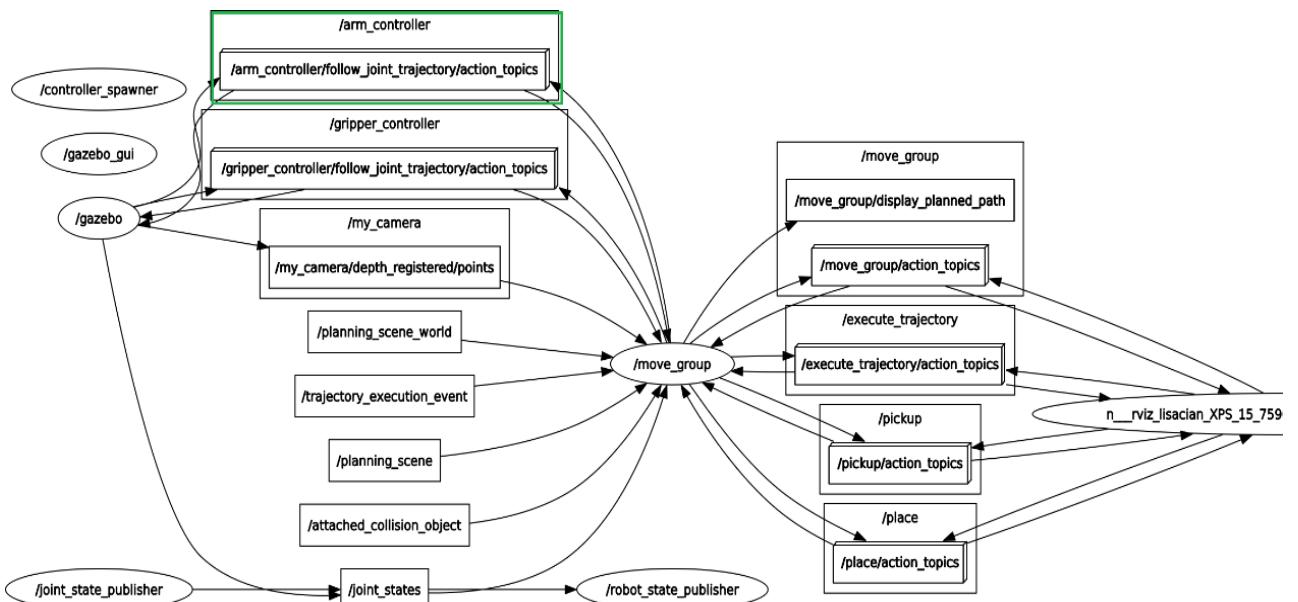


Por lo tanto, se ha logrado programar las siguientes partes de la acción:

- `arm_controller/follow_joint_trajectory/goal`: comando que permite recibir la ruta planificada por `Moveit!` en forma de `trajectory_msgs::JointTrajectory`.
- `arm_controller/follow_joint_trajectory/cancel`: comando que puede interrumpir la acción que se está ejecutando en cualquier momento.

3.3. RQT GRAPH

- El nodo `move_group` recibe la posición deseada desde `MotionPlanning` de `Rviz`, la posición actual de las articulaciones desde el topic `/joint_states` y calcula la trayectoria.
- La trayectoria deseada se transmite al controlador `/arm_controller` de tipo `par_computado` mediante el servidor de acciones.
- El controlador manda a `gazebo` el par necesario para cumplir la ruta
- `Gazebo` envía el feedback del controlador al nodo `move_group` a través de `/joint_states`.



3.4. RESULTADOS

Antes de mostrar los diferentes resultados obtenidos, se ha de explicar la forma de obtener los diferentes errores. Se ha optado por la representación en escala logarítmica puesto que el rango de los distintos valores queda reducido a uno de tamaño más manejable.

Además, cabe mencionar que, para la obtención de los datos, se han creado en ROS unos tópicos que publican las informaciones en archivos .csv.

```
> rostopic echo /topic -p > datos.csv
```

Los datos de dichos archivos se han graficado en Matlab.

3.4.1. Comparación distintos parámetros de Par computado

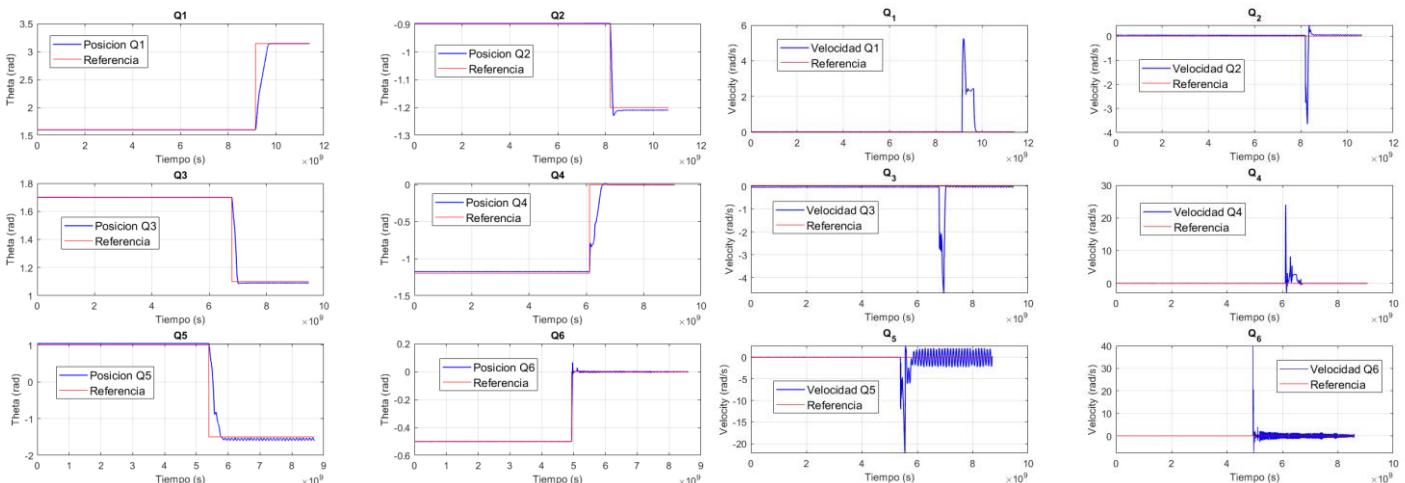
Para encontrar unos buenos parámetros de control, se han tenido que realizar diversas pruebas variando estos en función de los resultados que se obtenían. Los resultados más representativos han sido los obtenidos de las configuraciones expuestas a continuación.

3.4.1.1. Configuración 1

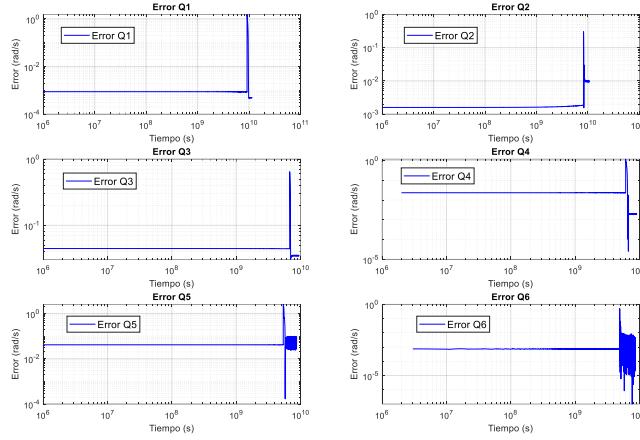
En esta primera configuración, se han introducido los siguientes parámetros para kp y kv:

- $kp_1 = 1500, kp_2 = 1500, kp_3 = 1500, kp_4 = 250, kp_5 = 250, kp_6 = 1;$
- $kv_1 = 50, kv_2 = 50, kv_3 = 50, kv_4 = 10, kv_5 = 10, kv_6 = 1;$

Con estos parámetros se han obtenido resultados razonables, pero no demasiado buenos, puesto que, como se puede observar en las gráficas tanto de posición como de velocidad de las distintas articulaciones, sobre todo en la articulación n°5, se alcanza la referencia, pero sobre oscila demasiado.



Además, como se puede apreciar en la gráfica de la página siguiente, el error no es grande, aunque en el régimen permanente de la articulación 5 oscila bastante. Esto se ha mejorado con la siguiente configuración en la cual se ha optado por aumentar tanto las kp como las kv.

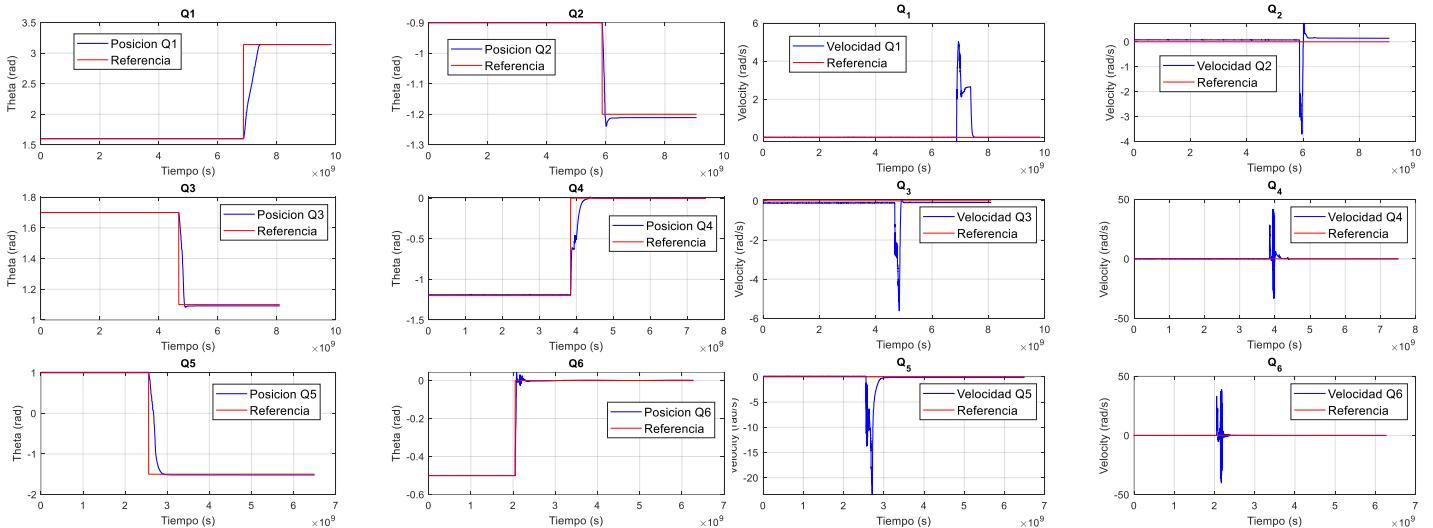


3.4.1.2. Configuración 2

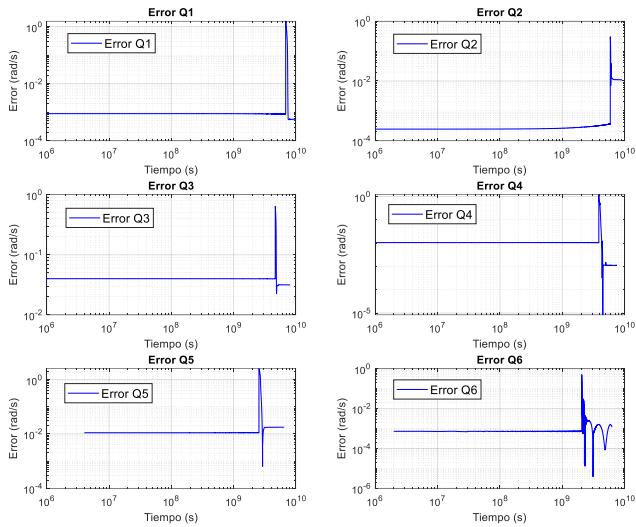
Con esta segunda configuración, se han introducido los siguientes parámetros para kp y kv:

- $kp_1 = 1750, kp_2 = 1750, kp_3 = 1750, kp_4 = 500, kp_5 = 750, kp_6 = 1;$
- $kv_1 = 50, kv_2 = 50, kv_3 = 50, kv_4 = 15, kv_5 = 25, kv_6 = 1;$

Se puede comprobar que, aumentando los valores de los distintos parámetros el control mejora considerablemente, eliminando la sobre oscilación no deseada, tanto en la articulación nº 5 como en la nº 6. Además, se puede observar que la articulación nº 4 asciende de forma más suave.



Con respecto a los errores, se comprueba que el razonamiento anterior es correcto, pues se esperaba una reducción del ruido en el régimen permanente. Además, en las distintas articulaciones también se ha conseguido reducir la magnitud del error.



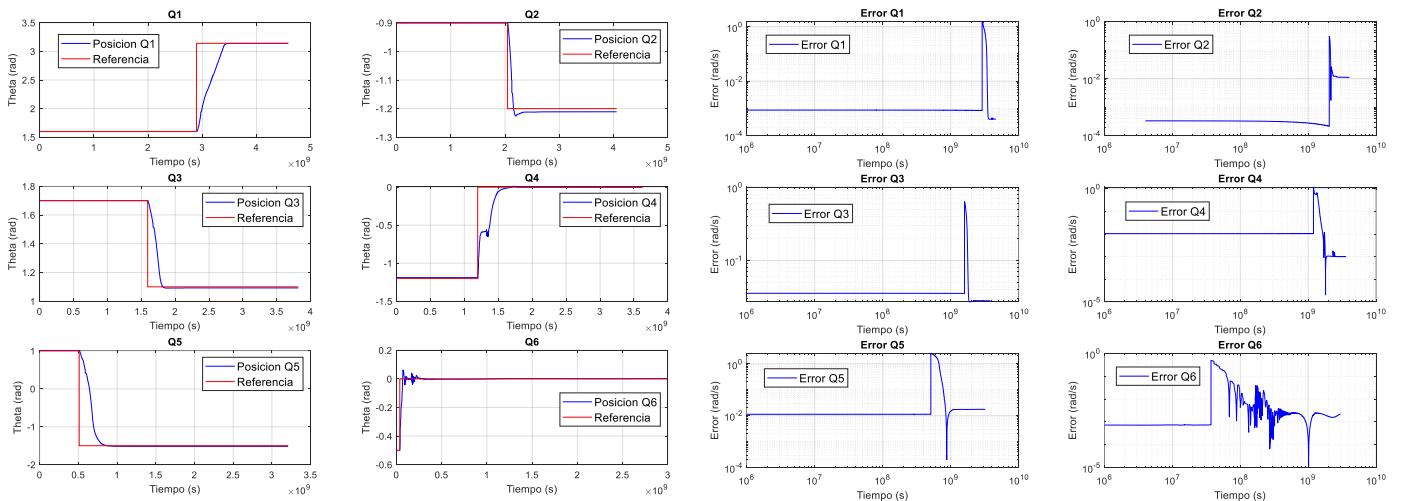
Para la siguiente configuración, se ha tratado de corregir el pico producido en la gráfica de posición de la articulación nº 3 para que llegue a la referencia de una forma suave. Para ello, se han subido los parámetros de dicha articulación.

3.4.1.3. Configuración 3

Para esta última configuración, se han introducido los siguientes parámetros para k_p y k_v :

- $k_p1 = 1750, k_p2 = 2000, k_p3 = 2000, k_p4 = 500, k_p5 = 750, k_p6 = 1;$
- $k_v1 = 50, k_v2 = 70, k_v3 = 70, k_v4 = 15, k_v5 = 25, k_v6 = 1;$

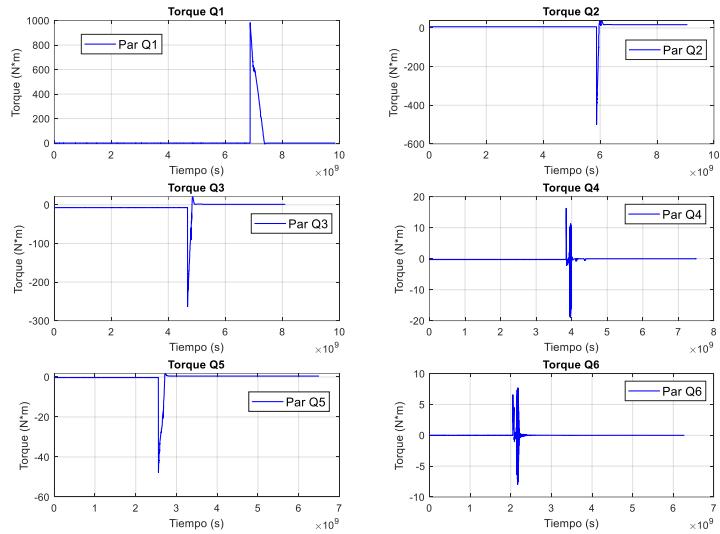
En la posición se observa que el pico de la articulación nº3 se corrige, pero eso implica que el control sobre la articulación siguiente, es decir, la nº 4, empeore. Además, los errores también son mayores, sobre todo se observa en la articulación nº 6.



3.4.1.4. CONCLUSIÓN DE LAS PRUEBAS

Par Tras múltiples pruebas, se ha llegado a la conclusión de que los mejores resultados se han obtenido con los parámetros de la configuración 2, puesto que, como en el caso anterior, se mejoraba una articulación y empeoraba el resto.

El par calculado para cada instante de tiempo referente al control de la configuración nº 2, se puede observar en la gráfica de abajo.



3.4.2. Comparación controladores PI frente a Par computado

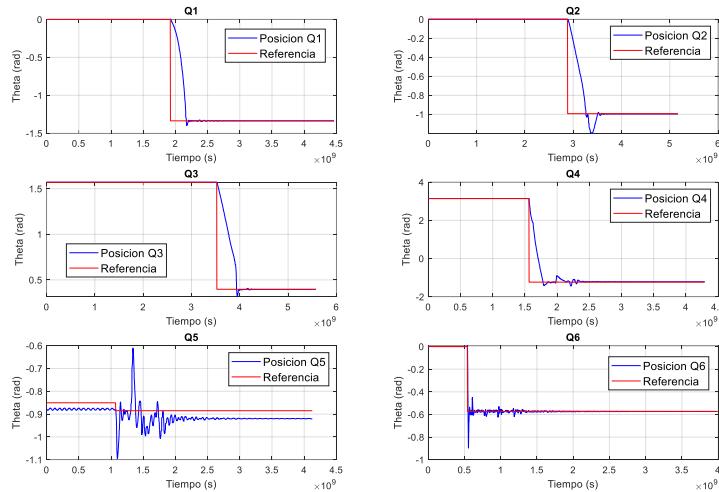
Una vez lograda una configuración de los parámetros k_p y k_v que permiten implementar un control de par computado suave y estable, se va a comparar en posición con el PI, previamente calculado del mismo modo, para poder analizar los resultados de ambos controladores al mismo tiempo y, así, contrastar los resultados con la teoría impartida en la asignatura.

En primera instancia, se espera que el Par Computado ejerza un control más limpio y con menos ruido, ya que, dada la tendencia lineal del PI, se realizan aproximaciones que deberían empeorar el control, si se compara con el antes mencionado Par Computado.

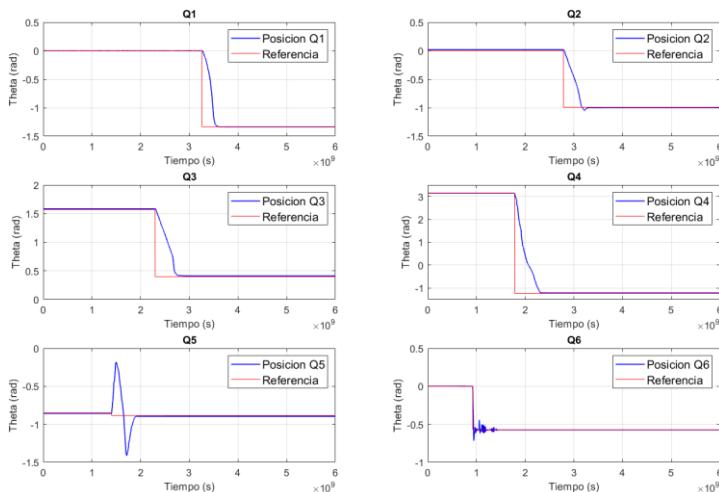
3.4.2.1. Posición

En posición, los resultados obtenidos para ambos controles han sido los siguientes:

- Resultados en posición referidos al control PI:



- Resultados en posición referidos al control Par Computado:



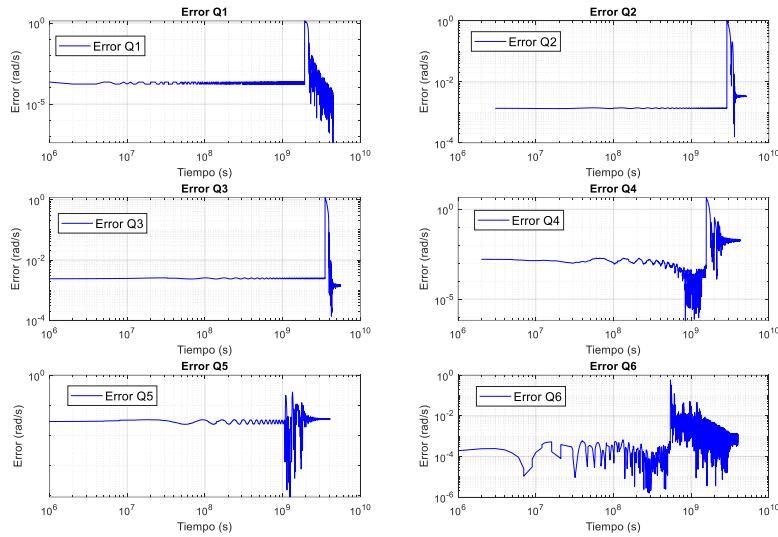
Si se comparan ambas simulaciones, se puede apreciar, ya con dichas gráficas, como el Par Computado proporciona un control mucho más suave que el PI, arreglando los problemas de ruido y sobre oscilaciones. No obstante, para una mayor precisión a la hora de visualizar los errores

cometidos en uno y otro caso, se muestra el error cometido en posición por cada controlador en escala logarítmica.

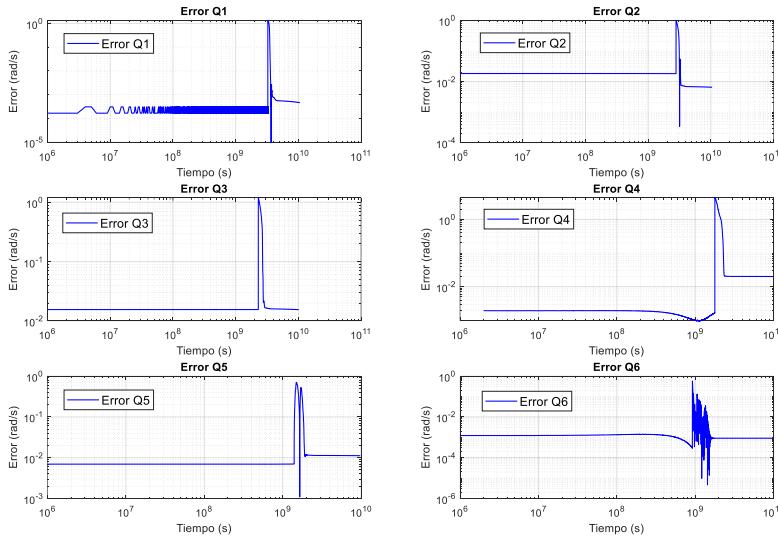
3.4.2.2. Error

Los resultados visualizados en forma de error en posición son los siguientes:

- Errores en posición referidos al control PI:



- Errores en posición referidos al control Par Computado:



Finalmente, aquí si se puede visualizar de mejor manera la mejoría que introduce el control por Par Computado. Se observa como en la primera articulación (shoulder_lift_joint) el Par Computado introduce un poco más de ruido al comienzo, pero alcanza de forma mucho más estable que el PI el waypoint dado. Con el resto de articulaciones, la mejoría es aún más notable, pues podemos ver como el ruido desaparece en gran medida en su mayoría y decrementa la amplitud de las oscilaciones.

4. PROBLEMAS ENCONTRADOS

No todos los problemas encontrados durante el desarrollo del proyecto serán expuestos en la memoria, debido a la gran cantidad de los mismos. Sin embargo, se hará una breve mención de aquellos que han supuesto una mayor dificultad a la hora de resolverse.

4.1. MAPA GIRADO

En el proyecto inicial la transformada del mapa giraba un ángulo indefinido, por lo que el robot no podía moverse dentro del mapa correctamente.

Finalmente, se observó que el error se encontraba en el mapa y no en el control por Pure Pursuit. Por este motivo, se tuvo que cambiar el mapa y cargar un mapa con una transformada sin rotación.

4.2. PROBLEMA 2

El hecho de poner en funcionamiento el action server ocasionó un gran problema, puesto que el client de *MoveIt!* no veía al controlador. Aunque el controller_manager cargaba correctamente el arm_controller (type: par_computado/ParComputado), al lanzar moveit aparecía este error:

```
[ WARN]: Waiting for arm_controller/follow_joint_trajectory to come up  
[ERROR]: Action client not connected: arm_controller/follow_joint_trajectory
```

Al final, se descubrió que dicho error aparecía porque se necesitaba una JointTrajectoryAction para conectar *MoveIt!* (move_group) con el controlador, y se ha conseguido arreglar dicho problema programando un action server dentro del plugin del controlador.

4.3. PROBLEMA PURE PURSUIT

Otra de las grandes dificultades que nos encontramos resultó al tratar de sustituir el controlador de la plataforma ya existente, y conectar el Pure Pursuit que se ha diseñado.

El modelo inicial ya disponía de un controlador de alto nivel, un ros_controller que lee parámetros de un archivo .yaml. Por lo tanto, ha sido posible cambiar el topic donde el controlador del modelo publica a un topic que no es leído por ningún otro nodo. De esta forma, el nodo Pure Pursuit puede publicar sobre el topic /ommp_velocity_controller/cmd_vel. De esta forma se ha solucionado el problema, consiguiendo, así, desconectar el controlador original.

4.4. PROBLEMA AL RECIBIR LOS WAYPOINTS DEL PAR COMPUTADO

Tras comprobar el correcto funcionamiento del Par Computado en Simulink y tener en correcto funcionamiento el action server, era de esperar que el control debía funcionar en ROS. En contraposición, el brazo no seguía la referencia como es debido.

Finalmente, se detectó el error que se estaba cometiendo en el orden en el que se cogían las referencias en posición y velocidad de las articulaciones del brazo. El shoulder_pan_joint, el cual es la articulación nº1 para nosotros, se encontraba sustituida por el elbow_joint, nuestra 3^a articulación. Al conocer este cambio en el orden de recepción de las referencias, se crearon variables auxiliares que permitieron solventar el problema, reordenando las articulaciones, y permitiendo al control funcionar correctamente.

4.5. INTENTO DE PLANIFICADOR DE TRAYECTORIAS

También se ha intentado hacer un plugin para Movebase, para calcular la trayectoria mediante de un algoritmo RA (Relaxed A*). Se ha realizado el plugin en c++ conectado a Movebase y la librería del paquete funciona. Se cree que el problema se haya en el código que implementa el Relaxed A*, pues lo más probable es que exista un overflow

No obstante, debido a la implementación de los objetivos principales y a la falta de tiempo, no ha sido posible encontrar la solución, lo cual ha impedido la posibilidad de conectar al proyecto porque salía un error muy raro.

5. CONCLUSION

A pesar de que se intentó realizar un control conjunto de la plataforma móvil y del brazo manipulador y no se ha conseguido, se ha logrado realizar una única simulación en la que ambos controles desacoplados funcionan correctamente.

Se puede considerar, que el objetivo del proyecto se ha cumplido, puesto que dado un punto la plataforma móvil consigue llegar de una forma relativamente buena y, además, el brazo manipulador de 6 GDL es capaz de llegar a una posición de referencia cualquiera, dentro de sus posibilidades.

Todo nuestro proyecto se puede encontrar en Github:
https://github.com/guidosassaroli/mobile_manipulator.git

6. REFERENCIAS

- https://github.com/panagelak/Open_Mobile_Manipulator
- <https://link.springer.com/content/pdf/10.1007%2F978-3-319-26054-9.pdf>
- http://wiki.ros.org/joint_trajectory_action
- <http://wiki.ros.org/actionlib>
- https://sir.upc.edu/projects/rostutorials/7-actions_tutorial/index.html#the-ros-action-server
- <https://moveit.ros.org/documentation/concepts/>
- http://docs.ros.org/en/api/control_msgs/html/action/FollowJointTrajectory.html
- https://github.com/ros-controls/ros_controllers/blob/melodic-devel/joint_trajectory_controller/include/joint_trajectory_controller/joint_trajectory_controller_impl.h
- https://docs.ros.org/en/api/actionlib/html/classactionlib_1_1SimpleActionServer.html
- http://wiki.ros.org/ros_control/Tutorials/Writing%20a%20new%20controller
- [http://library.isr.ist.utl.pt/docs/roswiki/actionlib\(2f\)DetailedDescription.html](http://library.isr.ist.utl.pt/docs/roswiki/actionlib(2f)DetailedDescription.html)
- <https://raw.githubusercontent.com/StevenShiChina/books/master/Mastering%20ROS%20for%20Robotics%20Programming.pdf>
- <http://wiki.ros.org/tf>
- https://www.youtube.com/watch?v=QyvHhY4Y_Y8
- Control de robot móviles con ruedas. Transparencias prof. Ollero
- <http://wiki.ros.org/en>
- <http://wiki.ros.org/navigation/Tutorials/Writing%20A%20Global%20Path%20Planner%20As%20Plugin%20in%20ROS>
- http://wiki.ros.org/global_planner
- https://github.com/larics/pure_pursuit
- http://wiki.ros.org/purepursuit_planner
- https://www.ri.cmu.edu/pub_files/pub3/coulter_r_craig_1992_1/coulter_r_craig_1992_1.pdf