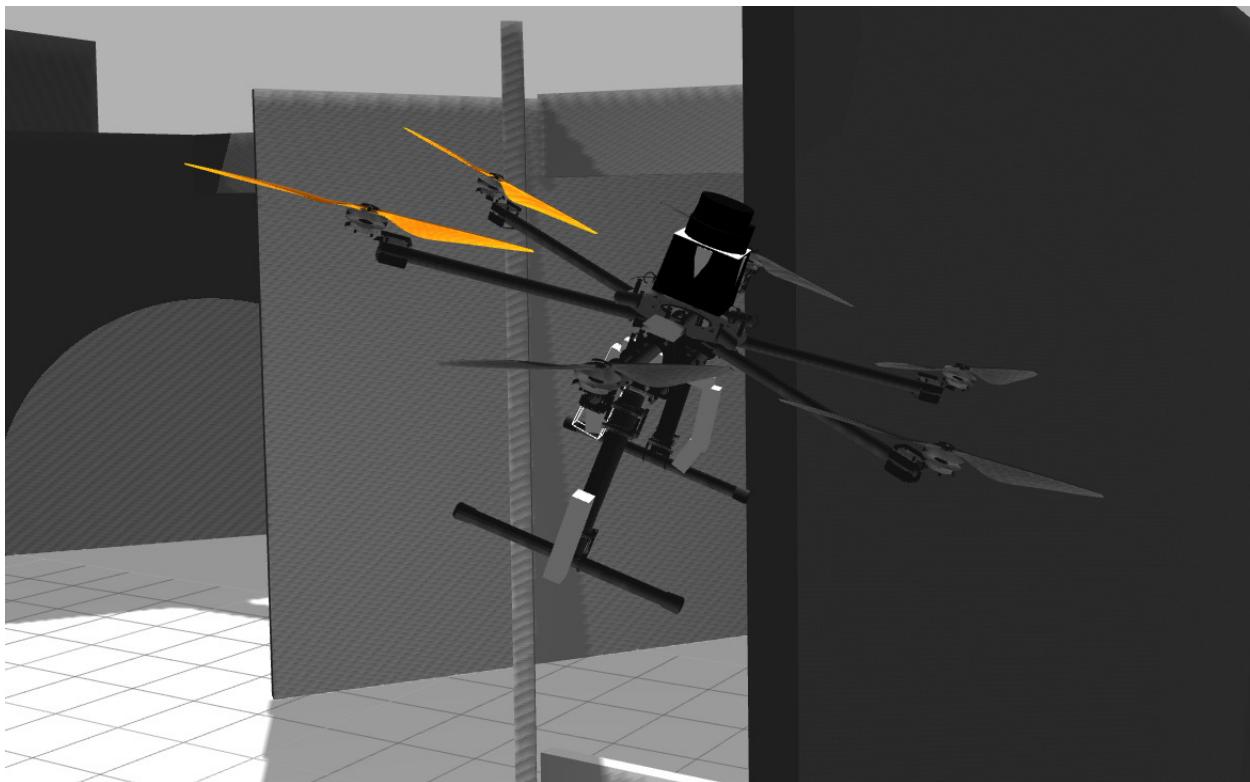




Robótica Avanzada

Memoria proyecto: Multirotor Repartidor

12 de junio del 2021



Autores:

Eduardo Sotelo Castillo
Jorge Rodríguez Rubio
Alejandro Rodríguez Armesto

Índice

Introducción	4
Objetivos	4
Tareas realizadas	6
Preparación de la simulación:	7
Mapeo del entorno	8
Actuadores	8
Sensores	9
Localización	9
Planificación	11
Control	13
Nodo central	13
Resultados de los experimentos	15
Funcionamiento de la garra	15
Funcionamiento del control	16
Funcionamiento de la localización	17
Funcionamiento de la planificación	21
Funcionamiento del sistema completo	22
Distribución de trabajos	23
Conclusiones	23
ANEXO 1: GRVC UAL	25
Herramientas de UAL	25
ANEXO 3: Algoritmo ICP y odometría	27
Algoritmo ICP	27

Introducción

En este proyecto se pretenden abordar todos los conceptos aprendidos durante el transcurso de la asignatura, intentando a la vez implementar un sistema de interés para el futuro cercano. Con esto en mente, hemos decidido crear un sistema que controle un UAV multirotor para que transporte paquetes a puntos definidos de un mapa tridimensional previamente mapeado. Todo ello, integrado en un sistema de ROS y utilizando técnicas de localización, planificación y control dadas en la asignatura.

Objetivos

El objetivo de este proyecto es el de diseñar un sistema capaz de hacer que un UAV multirotor reparta paquetes en zonas mapeadas con anterioridad, siendo capaz de localizarse en el mapa, moverse a cualquier posición de este para repartir las cargas siguiendo caminos bien planeados y volviendo al punto de origen para recoger nuevas cargas.

Idealmente, el UAV ha de partir de un punto inicial predefinido y conocido hacia un punto dado por el usuario. Una vez llegue a ese punto, debe soltar la carga y volver al punto inicial a volver a cargarse.

Podemos dividir los objetivos en distintas tareas concretas y relativamente independientes:

- Actuadores: El UAV deberá llevar integrado algún tipo de gripper que le permita sujetar cargas y soltarlas de manera segura. Este gripper debe poder ser controlado de forma sencilla y abrirse y cerrarse según se le ordene.
- Sensores: Además de GPS y IMU, el UAV necesitará de un LIDAR para localizarse. Se tratará de un LIDAR 3D montado en la parte superior del UAV que permita tomar medidas de todo el entorno.
- Mapa: Para realizar las simulaciones es necesario no solo crear un entorno en Gazebo sino además mapearlo para disponer de un fichero con todos los datos de dicho entorno.



- Localización: El UAV debe ser capaz de localizarse en todo momento en el mapa, utilizando los datos del LIDAR y del GPS e integrándolos mediante un filtro de Kalman. Idealmente, obtendremos esta localización con una frecuencia de 30 Hz
- Control: El UAV se controlará por medio del paquete UAL (*paquete grvc-ual, más información en el anexo 1*), que nos permite elegir la velocidad que queremos que tenga el UAV. De ese modo, y con un controlador PID, haremos de ser capaces de que el UAV alcance la posición del mapa que le ordenemos.
- Planificación: Utilizando un mapa ya cargado en memoria, el UAV debe ser capaz de moverse a cualquier posición segura de este mapa. Para ello, utilizaremos algoritmos de planificación que generarán trayectorias seguras para el vehículo.

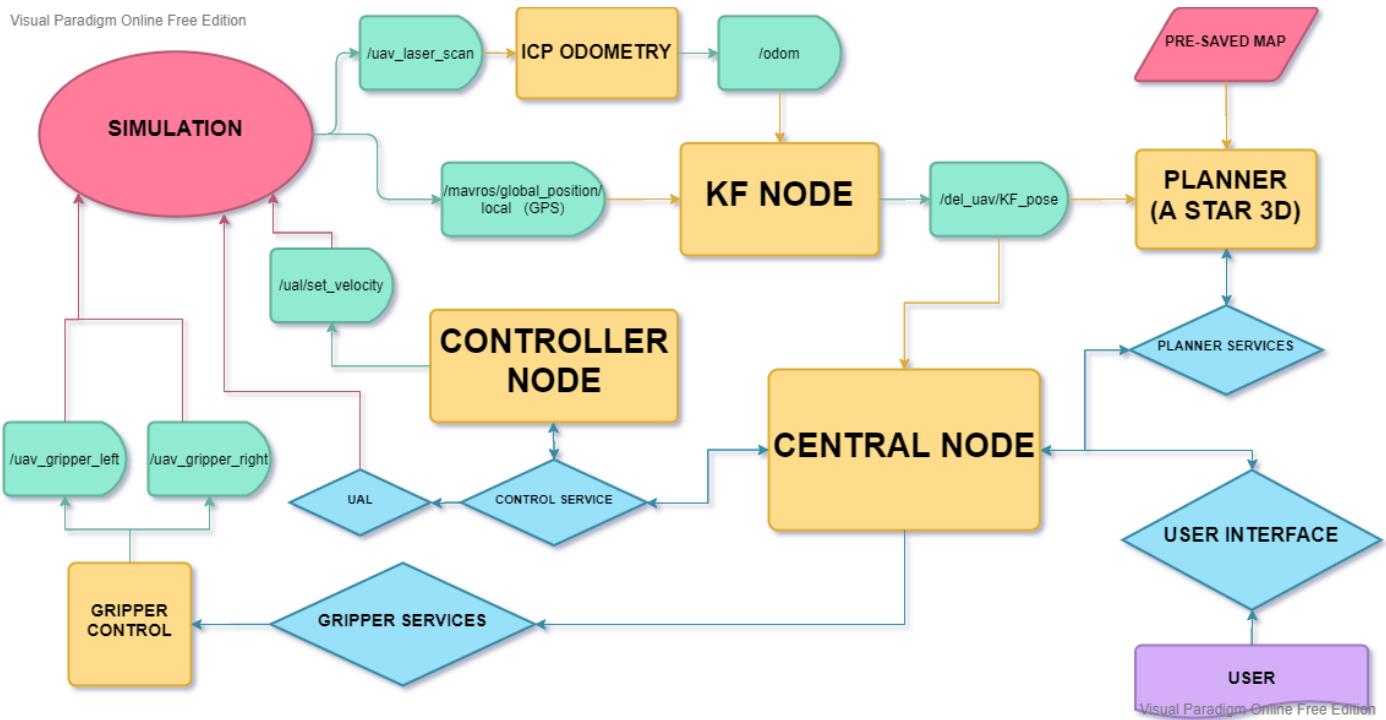
Todas estas tareas se implementarán en un mismo sistema de ROS, que conectará todos los nodos implicados en la operación del UAV y se encargará de regular los intercambios de información.

El UAV que utilizaremos es uno de los modelos incluidos con UAL. Este paquete nos permite implementar un UAV multirotor en Gazebo haciendo uso del autopiloto PX4 y del entorno MAVROS, y nos permite actuar sobre dicho robot. Se muestra a un lado una imagen del UAV tal y como viene en UAL.



Tareas realizadas

Los objetivos plasmados en el punto anterior se llevaron a cabo implementando un sistema de nodos y servicios relativamente complejo que permite al usuario (que simula a un servidor que accede remotamente al UAV) mandar diversas órdenes al vehículo. El esquema del sistema completo se muestra a continuación:



En el esquema se pueden observar como se relacionan los nodos (cuadrados amarillos) entre ellos y con el usuario. Hay dos formas de transferir información en ROS:

- Topics (figuras verdes) donde se publica información de forma síncrona para que cualquier nodo suscrito pueda leerlos.
- Servicios (rombos azules) donde se transfiere información de forma asíncrona entre dos nodos o entre un usuario y un nodo.

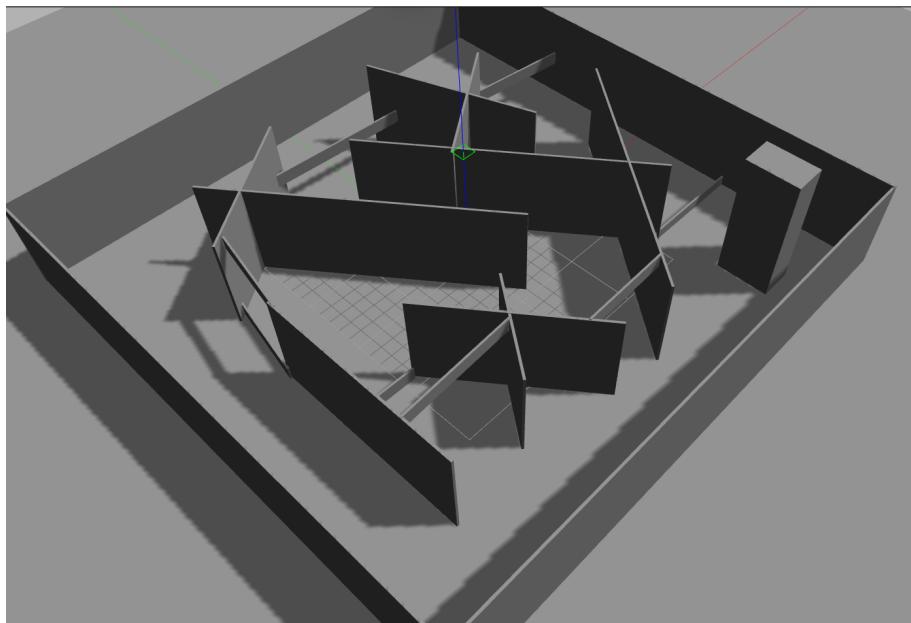


A continuación, se trata punto a punto como se han abordado los distintos objetivos y los resultados iniciales:

Preparación de la simulación:

La simulación se lleva a cabo utilizando Gazebo como entorno de pruebas y simulador de físicas y colisiones y los recursos del paquete UAL. Para ello, hemos creado una copia del UAV incluido en UAL en nuestro paquete, y hemos modificado los archivos para incluir los sensores y actuadores que necesitamos.

Además, Gazebo requiere un “mundo” (fichero .world) donde cargar los robots a simular. El mundo que utilizamos en este trabajo ha sido creado por el alumno Jorge Rodriguez Rubio y a continuación se adjunta una imagen del mismo:

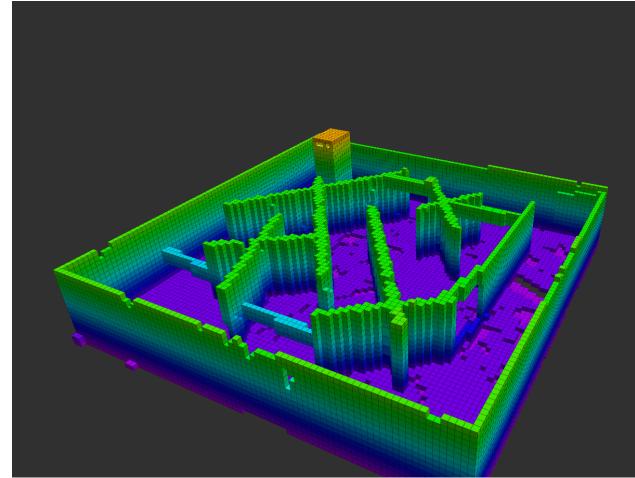




Mapeo del entorno

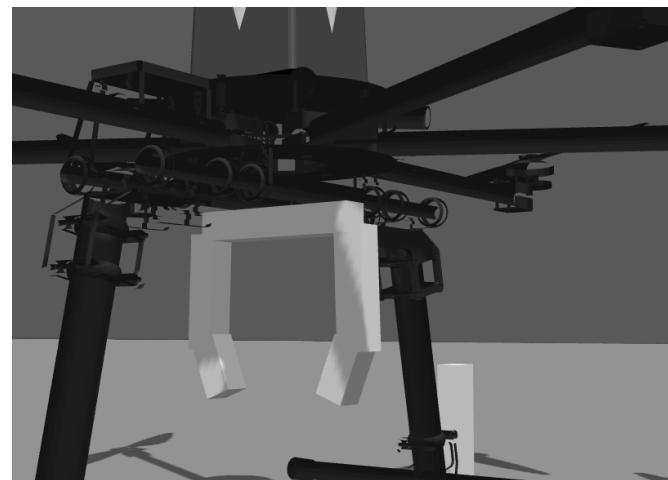
El mapeo lo hemos realizado utilizando un paquete llamado ROS_quadrotor_simulator junto con el de Octomap.

Pese al tiempo dedicado para la obtención del octree y pese a que estos son muchísimo más eficientes a la hora de buscar vecinos, no tuvimos tiempo suficiente para aprender a utilizarlos, por ello elegimos hacer nuestro propio voxelgrid en lugar de utilizar el que generamos con Octomap, ya que este es de 100x100x100 y como queda representado en la sección de experimentos el tiempo de planificación depende exponencialmente del número de voxels. Hemos acabado trabajando con tres mapas 2d, a 0, 3 y 6 metros de altura, con estos 3 mapas hemos creado un voxelgrid de 100x100x7.



Actuadores

Para agarrar los paquetes, el UAV llevará integrada una garra en su parte inferior. Para la simulación en Gazebo hemos creado un fichero URDF (*Consultar Anexo 2 para más información acerca de los ficheros URDF y xacro*) titulado simple_gripper.xacro, que se añade al modelo base del UAV en su parte inferior.





Esta garra, por limitaciones de Gazebo, no puede agarrar paquetes en la simulación, pero puede controlarse para abrirse y cerrarse sin mayores problemas, por lo que consideramos esto como un problema menor y nos centramos en poder controlarla de forma fiable.

La garra es controlada por medio de dos topics `/uav_gripper_right` y `/uav_gripper_left` que controlan el momento que se aplica sobre los dos “dedos” de la garra. Para simplificar este control y sincronizar el movimiento de la garra se implementa el nodo `“gripper_node”`.

Sensores

En el UAV vienen integrados varios sensores, entre ellos el GPS que utilizaremos para la localización. Por tanto, el único sensor que nos queda por implementar es el LIDAR. Este es implementado del mismo modo que la garra, programando su URDF.

Además del LIDAR 3D, está programado un LIDAR 2D, aunque este no se utilizó en la versión final del proyecto.

Dentro del URDF del LIDAR, se implementan tanto la posición física del sensor en el dron, como otros parámetros de funcionamiento, como pueden ser el número de muestras tomadas, el rango y la frecuencia de funcionamiento.



Localización

El problema de la localización es uno de los más complicados, ya que el UAV debe poder localizarse en un entorno tridimensional donde las perturbaciones son muy comunes y hay muy pocas opciones para obtener odometría.

Para poder solucionar este problema, dividimos este problema en dos partes: La obtención de odometría o “fase de predicción” y el filtrado de dicha odometría utilizando un GPS y un filtro de Kalman, la “fase de actualización”.

- Obtención de odometría: Obtener odometría en un UAV es complicado por lo general, dados los distintos efectos aerodinámicos y perturbaciones causadas por el viento. Por tanto, las soluciones adoptadas en el campo suelen pasar por el uso de odometría visual o el uso de LIDARs.

En nuestro caso, utilizaremos un LIDAR 3D para, aplicando un algoritmo ICP, obtener una estimación de la posición (*Consultar Anexo 3 para más información acerca del algoritmo ICP y su aplicación para obtener odometría por medio de un LIDAR*). Este proceso es realizado por un nodo “ICP odometry” sacado del paquete “*rtabmap_ros*” y adaptado a nuestro proyecto. El nodo se suscribe al topic “*/uav_laser_scan*” y publica su estimación de la posición y la covarianza de esta en el topic “*/odom*”.

Sin embargo, esta predicción tiene varios problemas. El primero es que es altamente dependiente de la geometría de su entorno, funcionando significativamente peor en entornos demasiado simples como el mapa de la simulación. Otro problema importante es el tiempo que tarda en ejecutarse cada estimación, que se publica con una frecuencia de aproximadamente 2 Hz.

Finalmente, al tratarse de un proceso iterativo que se apoya sobre la estimación anterior, se acumula el error a lo largo de las estimaciones y se hace necesario “actualizar” la estimación. De ahí la necesidad de implementar un filtro de Kalman.

- Filtrado de la estimación: Una vez obtenemos una predicción dada por el nodo “ICP odometry”, podemos filtrar esta señal con las medidas de nuestro GPS para obtener una localización robusta tanto a corto como a largo plazo. Para ello utilizaremos un filtro de

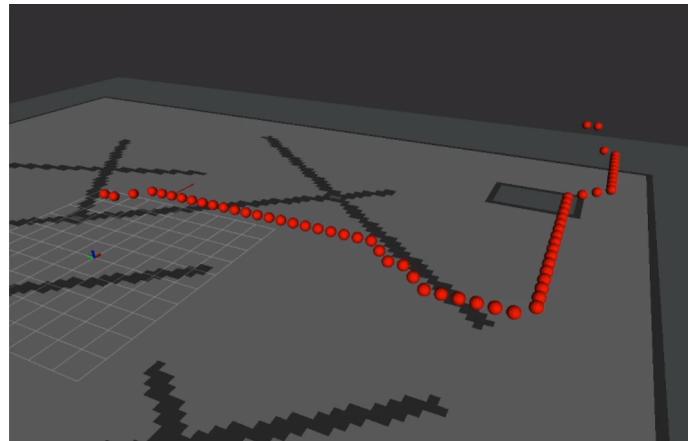


Kalman usando las estimaciones generadas por “ICP odometry” como posiciones predichas. Todo esto se implementa en el nodo “KF_node”, que además se asegura de detectar y solucionar posibles errores en la localización que se puedan dar por fallos del ICP o de UAL.

Finalmente, el producto de la localización se publicará en el topic “/del_uav/pose” para que otros nodos puedan leer el resultado.

Planificación

Para la planificación hemos decidido utilizar un algoritmo A*, uno de los algoritmos más conocido de pathfinding. Este es un algoritmo heurístico, es decir, un algoritmo en el que nunca se sobreestima el coste de alcanzar el objetivo, por lo que el coste de la posición actual siempre es el mínimo coste.



En cada iteración el algoritmo A* intenta elegir el siguiente nodo de forma que el camino tenga el menor coste posible, esto lo hace reduciendo la siguiente expresión:

$$f(n) = h(n) + g(n)$$

- n, nodo de estudio
- g, coste del camino del comienzo al nodo n
- h, estimación del coste del camino de n al nodo objetivo

El algoritmo itera hasta que el nodo “n” es el nodo objetivo o hasta que se ha explorado todo el mapa sin encontrar un camino válido. Podemos garantizar que si h es admisible*, el camino resultante será el de menor coste posible.



El algoritmo A* funciona de la siguiente forma:

Crear una clase “lista” que contiene el nodo actual, el padre y los valores de f, g y h

Crear dos listas, `lista_abierta=[]` y `lista_cerrada=[]`:

- `Lista_abierta` contiene todos los nodos que hemos encontrado pero no procesados.
- `Lista_cerrada` contiene los nodos que hemos visitado y procesado.

Introducimos la celda actual en la lista abierta: `lista_abierta.append(celda_actual)`

Mientras tengamos elemento en la lista abierta hacemos lo siguiente:

1- Pasamos el primer elemento de `lista_abierta` a `lista_cerrada` (el que tiene menor f)

2- Comprobamos si hemos llegado

3- Estudiamos los vecinos y eliminamos los obstáculos y los que ya hemos estudiado

4-Procesamos los vecinos restantes:

- Calculamos g_n :
- Si este coste fuera menor que el que ya tiene o bien este no está en la lista abierta hacemos:
 - Calculamos los parámetros h,g,f y padre para n, en caso de que no estuviera en la lista abierta lo meteríamos

5- Ordenamos la `lista_abierta` de menor a mayor en función de f y si hubiera empate elegimos el que tenga menor h

Se acabó el bucle, ahora tenemos que obtener los puntos de forma regresiva, esto lo haremos empezando por el final, comprobando el padre y buscándolo en la `lista_cerrada`, luego este será el siguiente y así hasta acabar

Por último, le doy la vuelta a path ya que este está ordenado del punto final al inicial

*Una función heurística se considera admisible si nunca sobreestima el coste de alcanzar el nodo objetivo

Control



Para interactuar con el UAV, utilizamos UAL. Concretamente, publicamos la velocidad deseada en el topic “/ual/set_velocity” para controlar directamente la velocidad del UAV. El valor de la velocidad deseada será calculado por un controlador PID a partir de la posición objetivo a la que queramos movernos y la posición actual obtenida de la localización.

Todo esto se lleva a cabo en el nodo “controller_node”. Este nodo, una vez el UAV ha despegado y UAL está listo para recibir comandos, tratará de mantener al UAV en una posición de referencia, actuando sobre la velocidad del robot en todo momento.

Para modificar esa posición de referencia y, por tanto, mover al UAV a la posición que deseemos, se dispone del servicio “/del_uav/goto”. Este servicio pide una posición en el espacio que, una vez el servicio sea recibido, parará a ser la nueva posición de referencia del UAV.

Nodo central

Finalmente, todos los sistemas implicados serán utilizados y coordinados por el nodo central o “central node”. Este nodo será el encargado de conectar todos los nodos entre sí y permitir al usuario actuar a alto nivel sobre el comportamiento del UAV. Es responsable de vigilar el estado de UAL, coordinar el despegue y aterrizaje, actuar sobre la garra, llamar al planificador para pedir trayectorias y utilizar el servicio del controlador para mover al robot donde corresponda.

Para comunicarse con el resto de nodos y sistemas se utilizarán servicios. Además, monitoriza el estado de UAL por medio del topic “/ual/state”.

Una vez el nodo está operativo, este recibirá órdenes del usuario o de un servidor, utilizando una serie de comandos que se le pasan al nodo central por medio del servicio “/del_uav/user_interface”, cuyo funcionamiento y uso se explicarán a continuación:

- Inicio del sistema: Activado con el comando “start”. Este es el primer comando que ha de usarse siempre, de modo que el sistema no recibirá órdenes hasta que no se haya ejecutado con éxito el inicio. Cualquier comando que se envíe antes de que el sistema haya iniciado bien se terminará a sí mismo tras un mensaje de aviso o se quedará a la

espera de que el sistema sea inicializado por otra terminal o usuario.

Una vez se llama al servicio, se cierra la garra para asegurar el supuesto paquete que cargamos y se despega a una altura que consideramos segura de 3 metros.

Tras el éxito de la inicialización, el “controller node” empieza a actuar y todos los comandos se desbloquean.

- Modo de viaje automático: Activado con el comando “auto” y una coordenada. Este comando ordena al UAV que se desplace a una coordenada dada siguiendo un camino seguro. Para lograr esto, el nodo central primero solicita al planificador una trayectoria segura y luego la sigue paso a paso, hasta finalmente estabilizarse en el punto final de la trayectoria.
- Cancelación de viaje: Se activa con el comando “idle” y puede activarse durante cualquier trayecto para cancelarlo en el siguiente waypoint.
- Soltar la carga: Se activa con el comando “drop”. Si se llama a este servicio, el UAV soltará su carga abriendo la garra.
- Vuelta a casa: Se activa con el comando “gohome”. Comando destinado a ser usado una vez se ha soltado la carga en un punto concreto, ordena al UAV que vuelva al punto inicial donde se ejecutó “start” al inicio de la operación. Una vez se llega al sitio, el servicio se encarga de ordenar el aterrizaje por medio del servicio “/ual/land”. Además, finaliza la operación, bloqueando el resto de comandos hasta que se vuelva a ejecutar “start”.

Resultados de los experimentos

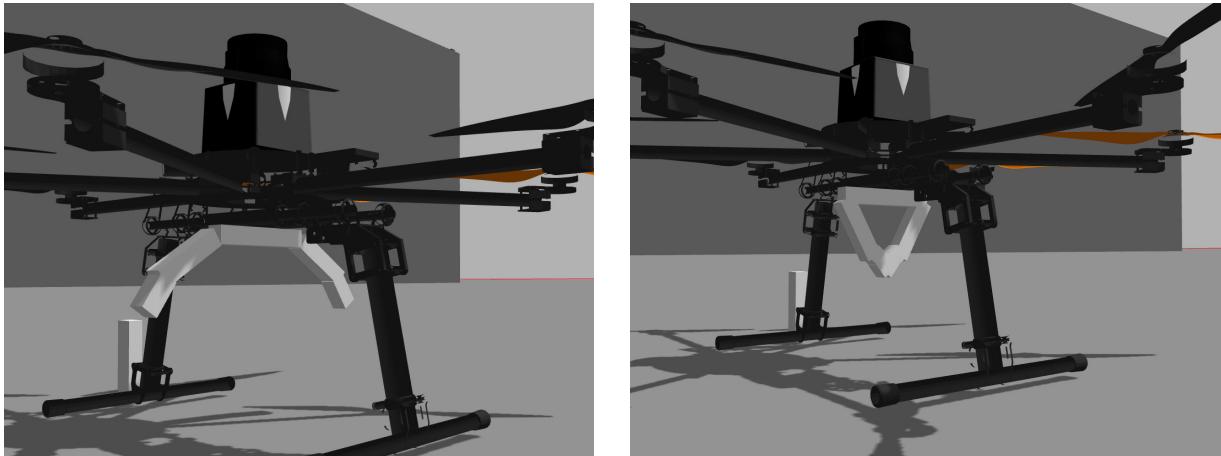


En este apartado mostraremos una serie de experimentos probando las distintas partes del sistema de forma individual y un último experimento con el sistema con todos sus nodos integrados y funcionando de forma conjunta.

Funcionamiento de la garra

En este experimento observaremos cómo se comporta la garra al usar su servicio.

En un inicio el gripper es relajado sin ningún tipo de fuerza aplicada, llamando al servicio podemos hacer que este se abra con el comando “open”, como vemos en la primera imagen, y que se cierre con el comando “close” como vemos en la segunda. También tenemos un tercer comando que nos permite volver al estado inicial eliminando la fuerza aplicada del gripper



Se muestran el estado de cerrado y de abierto del gripper.

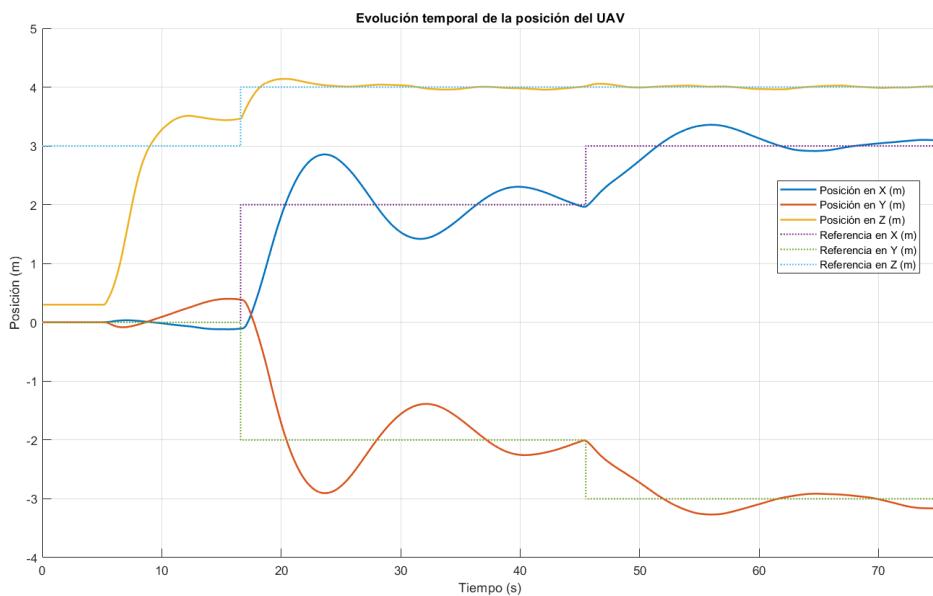
Funcionamiento del control

En este experimento comprobaremos cómo se comporta el control a la hora de moverse entre waypoints y seguir trayectorias predefinidas. Para la localización utilizaremos la proporcionada por UAL, a fin de independizar este experimento con el resto del sistema.



Los resultados han sido poco precisos, debido al menos en parte a la pésima estimación de velocidad que proporcionan los sensores de a bordo del UAV, que son los utilizados por UAL para cerrar el bucle del control en velocidad, base de nuestro control en posición. Por tanto, no vamos a exigirle demasiado, nos bastará con sobre oscilaciones de menos de un metro y medio, que es el factor de seguridad impuesto al planificador.

Se muestra a continuación la gráfica con la evolución temporal de la posición del UAV bajo la acción del control. En esta prueba se parte de [0, 0, 0], se despega a [0, 0, 3] y se mandan dos waypoints: uno a [2, -2, 4] y otro a [3, -3, 4]. En ella podemos observar como pese a que para desplazamientos de varios metros el sistema sobre oscila casi un metro, para desplazamientos pequeños como los que se dan entre los waypoints generados por el planificador no hay grandes problemas.



Funcionamiento de la localización



En estos experimentos mostraremos nuestra estimación de la posición usando el algoritmo ICP (visual odometry) y GPS durante un trayecto. Utilizaremos el control proporcionado por UAL a fin de independizar este experimento del resto del sistema. Los resultados serán comparados con la posición real del robot y la posición estimada por UAL.

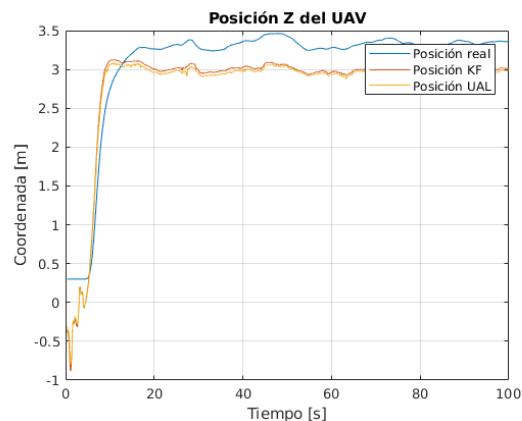
La localización se realiza con un Filtro de Kalman. Para todos los experimentos se lleva al UAV a la posición [-3,2,3] y luego a la posición [-4,6,3]

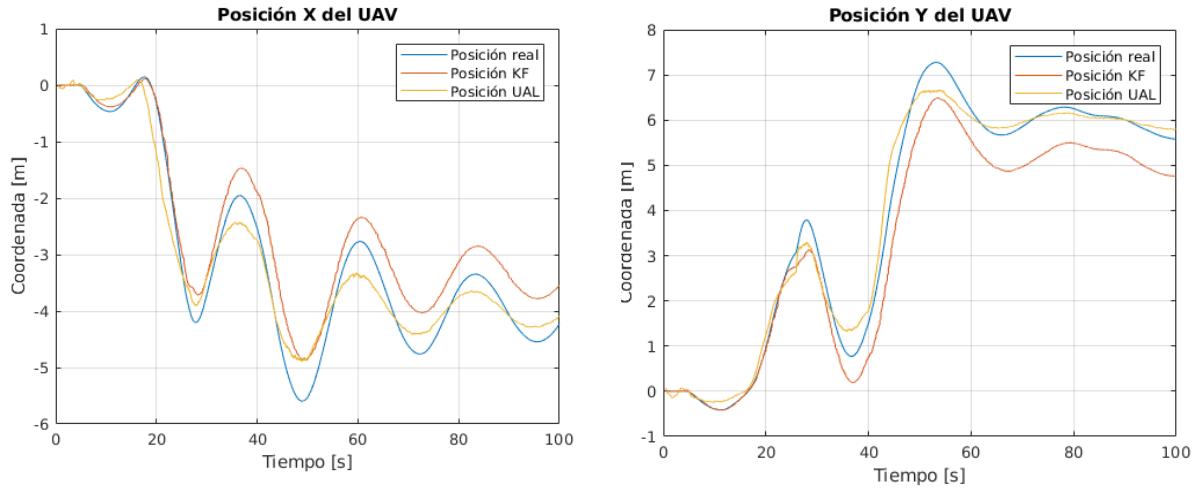
Los experimentos 1, 2 y 3 usan como punto predicho el proporcionado directamente por el algoritmo ICP, mientras que en los experimentos 1T, 2T y 3T el punto predicho se calcula multiplicando la matriz T proporcionada por ICP por el punto anterior (y sigma se estima mediante un modelo simple del sistema)

Los experimentos consisten en la variación de la covarianza del GPS (respecto a la covarianza de ICP), para así analizar los efectos que tiene en la estimación del filtro.

Dado que ICP estima muy mal la Z, en todos los experimentos la Z estimada estará muy cercana al valor dado por el altímetro.

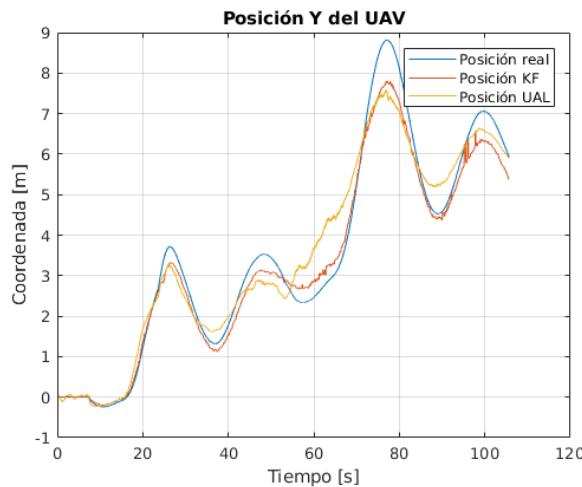
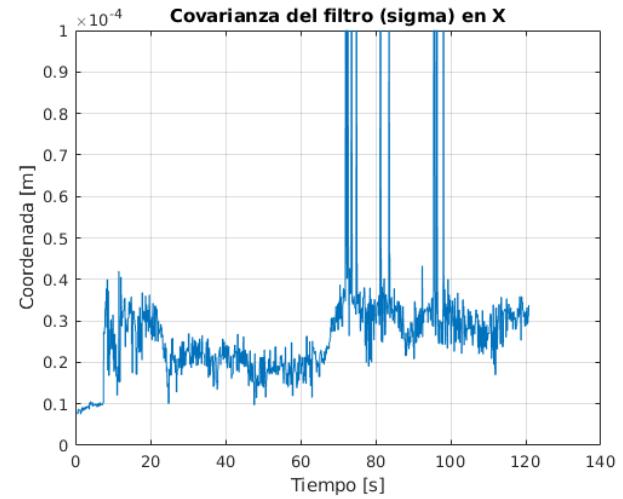
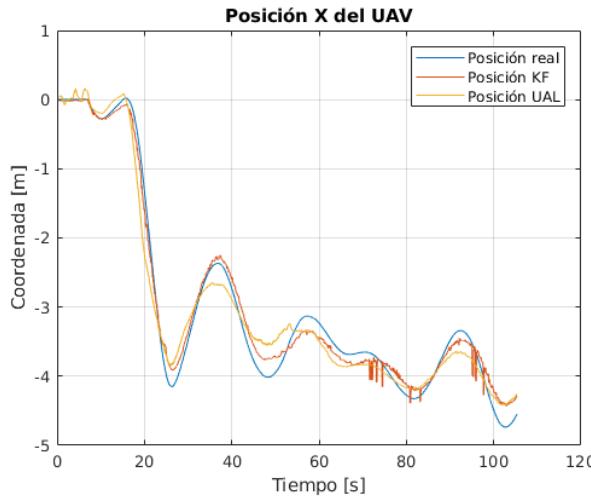
- Experimento 1: Covarianza GPS = 10^{-2} .
Tenemos una estimación muy buena al principio pero que luego se deriva y no se corrige. Las oscilaciones se deben a la dificultad que tiene el UAV con UAL para mantenerse estable en una posición.



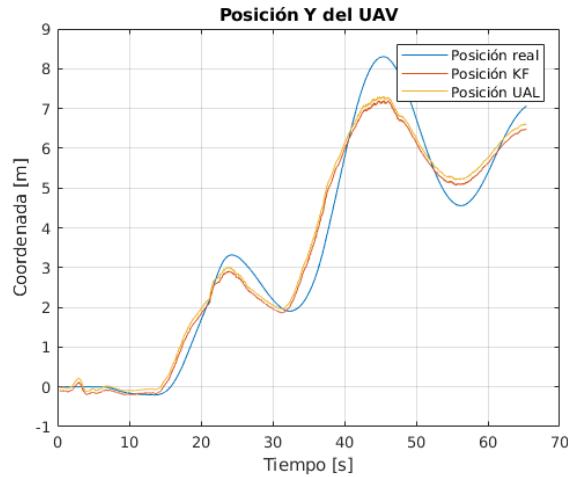


- Experimento 2: Covarianza GPS =

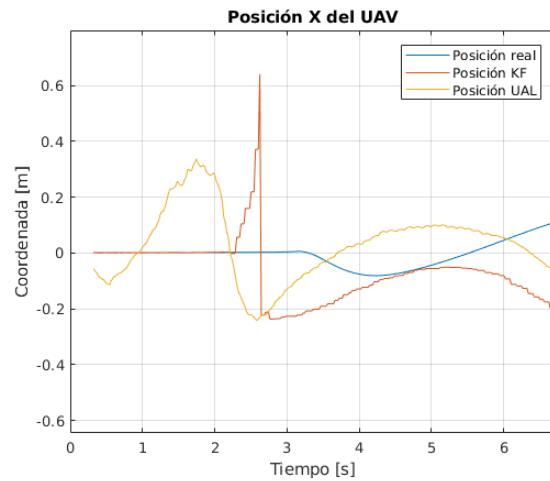
10^{-4} . Los resultados se parecen menos a la de la posición real, pero la deriva es menos notable. Es un valor que consideramos adecuado para el filtro y será el que usemos. Los picos en la posición estimada por KF se deben a instantes donde ha habido un error al calcular el ICP, y se puede ver que cuando esto sucede, tiene influencia en sigma, la credibilidad del filtro.



- Experimento 3: Covarianza GPS = 10^{-6} . La posición estimada es muy parecida a la posición estimada por UAL, que es parecida a la posición estimada del GPS.

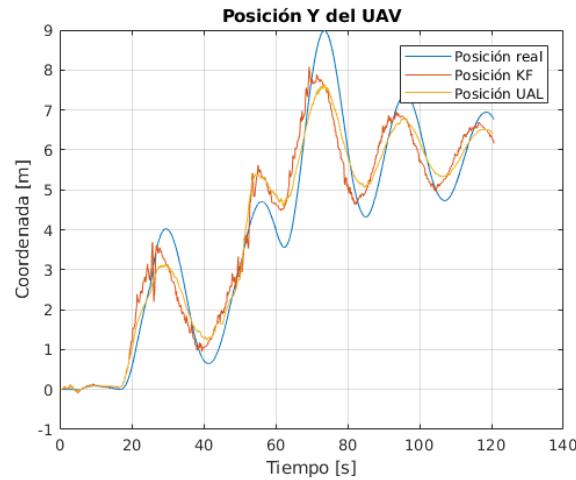


En ocasiones, ICP falla al iniciar y la posición estimada se deriva mucho de la real en el momento del despegue. El código es capaz de detectar este fallo y reiniciar ICP con la estimación del GPS cuando esto ocurre.

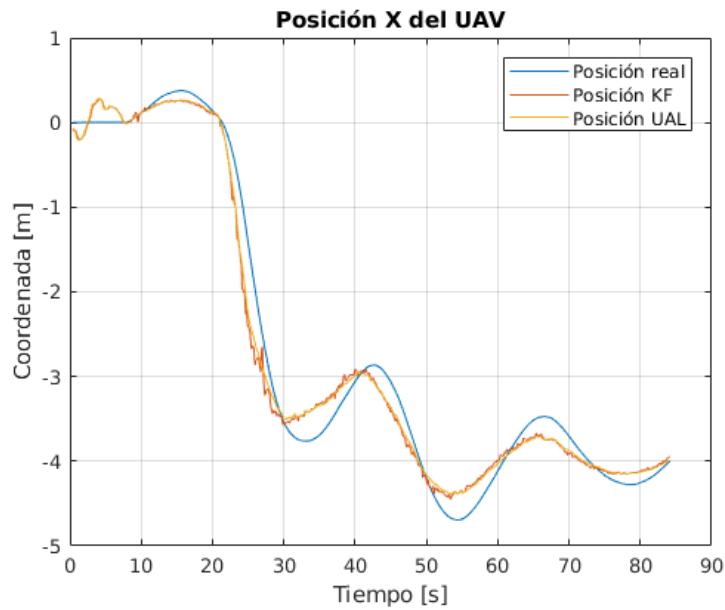




- Experimento 1T: Con una covarianza mayor para el GPS (10^{-3} frente a 10^{-5} del modelo), el filtro se fía más del modelo, obteniendo el siguiente resultado. Es una señal más ruidosa.



- Experimento 2T: Con un valor de 10^{-4} para la covarianza del GPS, la medida será más parecida a ésta.





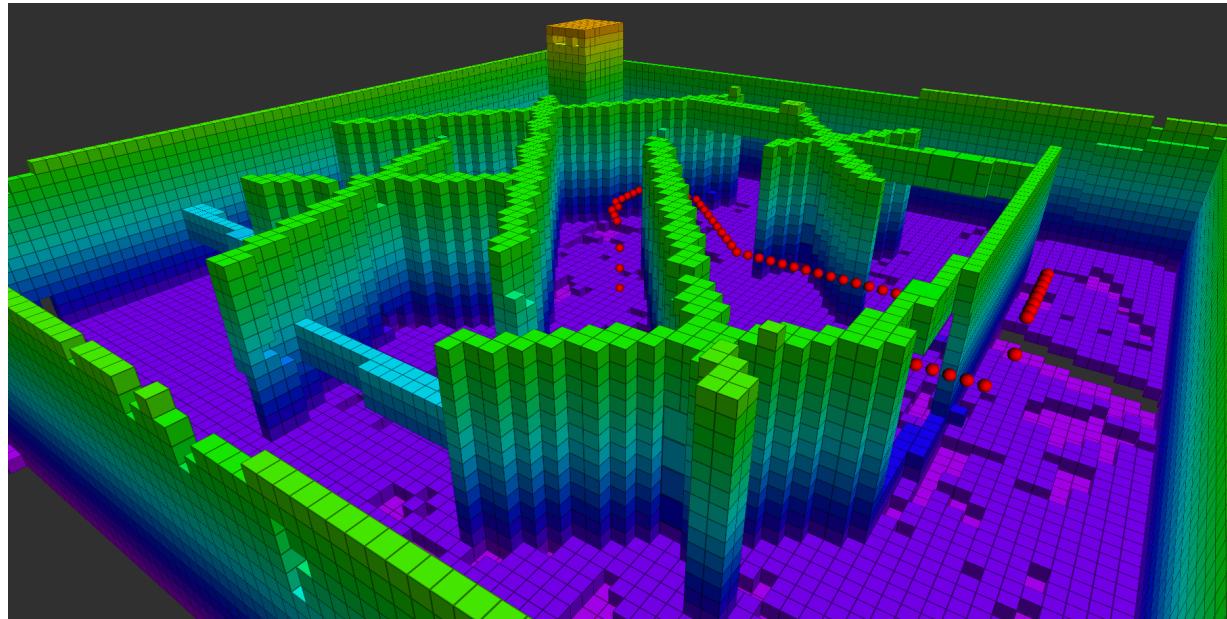
Funcionamiento de la planificación

El primer experimento a realizar va a ser el cálculo de una trayectoria, lo vamos a hacer con mapas de 2 resoluciones distintas, 0.3 y 0.5, el objetivo de esto es ilustrar el porqué hemos decidido utilizar 0.5 como resolución al trabajar con él A* en 3D.

Trayectoria	Resolución 0.3	Resolución 0.5
(0,0,0) a (-16,-1,-6)	295.64s	39.83s
(-16,-1,6) a (16,-1,3)	2916.39s	96.01s
(17,17,1) a (-3,-3,5)	3222.14s	117.68s

Hay que tener en cuenta que estos experimentos se han realizado solo con el planificador activo, estos tiempos aumentan mucho cuando tenemos todo el sistema funcionando

A continuación podemos ver la primera trayectoria representada en rviz, pese a ser una trayectoria no demasiado complicada la diferencia entre una resolución y otra es de más de 4 minutos.





Funcionamiento del sistema completo

En este experimento se integran todos los nodos y llevamos a cabo un ciclo completo de funcionamiento: el UAV deberá despegar, moverse a un punto interesante, abrir la garra, volver al punto inicial y aterrizar.

Este funcionamiento se muestra en el siguiente video:

[\[LINK AL VIDEO\]](#)

Distribución de trabajos

El trabajo se distribuyó entre los distintos miembros del grupo de la siguiente forma:

Tarea	Alumno/s encargado/s
Nodo Central	Eduardo Sotelo
KF node	Alejandro Rodríguez
ICP odometry	Eduardo Sotelo/Alejandro Rodríguez
Nodo Planificador	Jorge Rodríguez/Alejandro Rodríguez
Nodo de control	Eduardo Sotelo
Creación del mundo en Gazebo	Jorge Rodríguez
Mapeo del mundo	Jorge Rodríguez
Programación de la garra e implementación como servicio en ROS (Gripper node)	Eduardo Sotelo
Programación e implementación del LIDAR	Eduardo Sotelo/Alejandro Rodríguez
Investigación Octrees	Jorge Rodríguez
Investigación ICP	Eduardo Sotelo/Alejandro Rodríguez
Investigación UAL	Alejandro Rodríguez

Conclusiones

El objetivo inicial de este proyecto era el de abarcar todo lo aprendido durante la asignatura, objetivo el cual hemos cumplido satisfactoriamente, aunque no sin problemas. Haber tomado un objetivo tan amplio nos ha dado una gran visión de la robótica en general y de cómo los distintos procesos interaccionan entre sí. Hemos podido aprender muy bien cómo ROS maneja distintos procesos simultáneos y los comunica con topics y servicios, así como la útil modularidad que proporciona el sistema de nodos. Y, sobretodo, nos hemos demostrado ser capaces de implementar un sistema completo y funcional por sí solo. Por todo esto, podemos decir que estamos muy satisfechos con el resultado del proyecto.

Sin embargo, intentar integrar tantos sistemas totalmente diferentes nos ha traído varios inconvenientes. Aunque hemos aprendido muy bien las ideas generales y el uso de ROS, no hemos podido profundizar en tantos aspectos como nos gustaría, viéndonos obligados en más de una ocasión a renunciar a alguna idea interesante por falta de tiempo.

Además, que tantos nodos funcionaran de forma simultánea se cobra un gran coste en cómputo, coste que nuestros ordenadores han tenido problemas para afrontar. En general, hemos encontrado muchos problemas de rendimiento a la hora de simular el sistema completo con todos los nodos, empeorando los resultados de forma general en equipos menos potentes.

Por todo esto, muchas ideas se quedaron encima de la mesa, como por ejemplo el uso de octrees para la planificación, explorar más a fondo las posibilidades de ICP o mejorar el control.

En definitiva, pese a los obstáculos encontrados por el camino y la dificultad de montar todo un sistema robótico, podemos afirmar con convicción que este proyecto cumple con lo esperado y que puede expandirse hasta llegar a ser un proyecto fin de grado, fin de máster o incluso una tecnología útil para el mundo real.

ANEXO 1: GRVC UAL

En este anexo explicamos la funcionalidad de uno de los paquetes en los que se fundamenta nuestro proyecto. Se trata de [grvc_UAL](#), o UAV Abstraction Layer, un paquete que nos ofrece diversas herramientas para utilizar distintos autopilotos en ROS, sin necesidad de saber nada acerca de su funcionamiento. Además, proporciona varios modelos de UAV que podemos simular y usar para probar UAL, siendo uno de ellos el “*mbzirc*”, usado para este proyecto.

Concretamente, en nuestro proyecto hemos utilizado el backend concreto que conecta con MAVROS y el autopiloto PX4, copiando a nuestro el descriptor del robot “*mbzirc*” para modificarlo a nuestros fines. Gracias a la simplicidad de UAL no hemos tenido que configurar o cambiar nada de estos paquetes, sino que solo tuvimos que pasar como argumento los ficheros de nuestro descriptor de robot y nuestro mundo en Gazebo.

Un detalle importante es que UAL no está terminado, lo que da lugar a una gran inconsistencia en su funcionamiento, errores aparentemente aleatorios o funcionamientos extraños.

Herramientas de UAL

UAL ofrece diversas herramientas de muy alto nivel para controlar el robot e interactuar con él, tanto por medio de código en C++ como por medio de topics y servicios de ROS como se hace en este trabajo. A continuación, se detalla una explicación de los servicios y topics más importantes que han sido utilizados:

- Topic “/ual/state”: Según el valor de este estado, el UAV no podrá despegar o recibir comandos de movimiento.
- Topic “/ual/pose”: por medio de sensores integrados en el UAV, UAL puede estimar la posición en el espacio con cierta precisión. Esta información se publica en el topic y es utilizada por el servicio “/ual/go_to_waypoint” para cerrar el bucle de control.
- Topic “/ual/set_velocity”: Cuando publicamos unos valores de velocidad en este topic, el UAV intentará alcanzar esa velocidad.



Este es el método que utiliza el “*controller node*” de nuestro proyecto para controlar el movimiento del robot.

- Servicio “/ual/take_off”: servicio que permite hacer despegar al UAV a una altura deseada, siempre que el estado de UAL lo permita.
- Servicio “/ual/land”: servicio que permite hacer aterrizar al UAV, siempre que el estado de UAL lo permita.
- Servicio “/ual/go_to_waypoint”: servicio que permite controlar en posición al UAV, moviéndolo a una posición especificada en la llamada al servicio.

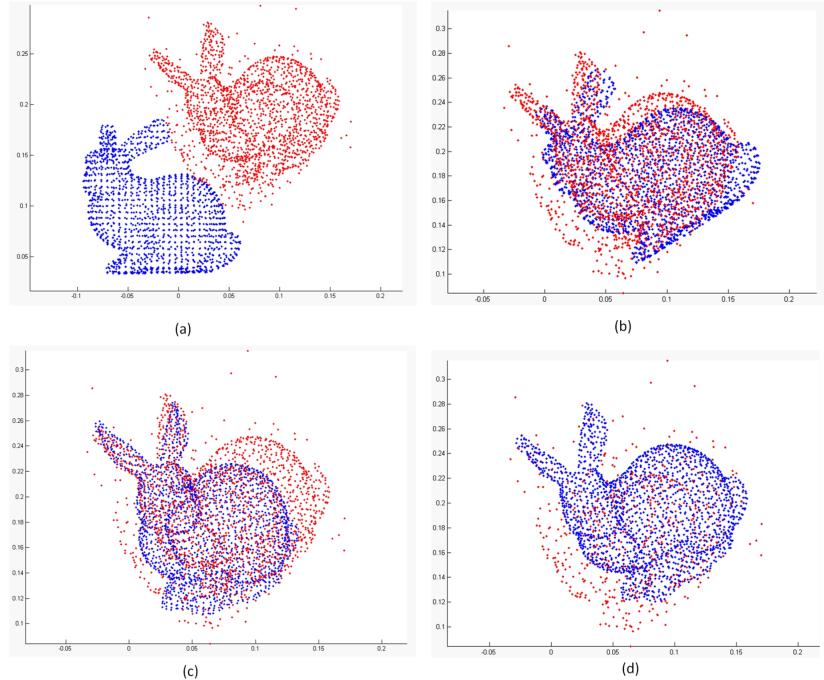


ANEXO 3: Algoritmo ICP y odometría

En este proyecto, la odometría se obtiene utilizando los datos de un sensor LIDAR 3D para calcular los desplazamientos entre escaneos. Para ello, compararemos escaneos en distintos instantes partiendo de la hipótesis de que los escaneos se hacen lo suficientemente rápido como para que ambas nubes de puntos generadas tengan la misma geometría pero desplazada. Partiendo de esa hipótesis, aplicamos un algoritmo ICP para obtener la matriz de transformación que definirá el movimiento ocurrido entre escaneos y que, integrada a la predicción de posición anterior, nos da la posición actual predicha.

Algoritmo ICP

Iterative Closest Point (ICP) es un algoritmo empleado para minimizar la diferencia entre dos nubes de puntos, "superponiéndolas" en el espacio. Para lograr esto, el algoritmo intenta definir una matriz de transformación que minimice una métrica del error (normalmente la distancia entre puntos), iterando hasta obtener un resultado que ponga esta métrica por debajo de un umbral dado. Además, el algoritmo puede funcionar aunque existan outliers, siempre y cuando la métrica del error se minimice.



Lamentablemente, por muy rápido que se hagan los escáneres, las nubes de puntos obtenidas por el sensor LIDAR en posiciones distintas del mapa tendrán diferencias que introducirán errores a la estimación.

Otro gran problema de este método es la falta de información. En un mapa con paredes planas y sin muchos elementos con geometrías fácilmente reconocibles, ICP carece de información para discernir los movimientos. Este es el caso, por ejemplo, cuando nuestro UAV asciende y deja de detectar el suelo con el LIDAR, de modo que está constantemente viendo las mismas nubes de puntos generadas por una misma pared plana a distintas alturas. En este caso podemos ver como la posición estimada por ICP no cambia en absoluto.

Además, nuestra forma de obtener la posición consiste en una predicción iterativa que se apoya en el valor anterior de la predicción para aplicar la nueva transformación. Por tanto, cuando aparece un error, este irá acumulándose hasta generar una deriva que hace inutil la predicción. He ahí la necesidad de una fase de actualización que filtre esta predicción con medidas de sensores.