Trabajo Fin de Grado
Ingeniería Aeroespacial

Robustez en sistemas de UAVs cooperativos. Simulaciones en ROS/Gazebo.

Autor: José Ángel Jiménez Doblado

Tutor: José Miguel Díaz Báñez

Dep. Matemática aplicada II
Escuela Técnica Superior de Ingeniería
Universidad de Sevilla

Sevilla, 2016

Trabajo Fin de Grado
Ingeniería Aeroespacial

Robustez en sistemas de UAVs cooperativos. Simulaciones en ROS/Gazebo.

Autor:
José Ángel Jiménez Doblado

Tutor:
José Miguel Díaz Báñez
Profesor Titular

Dep. Matemática aplicada II
Escuela Técnica Superior de Ingeniería
Universidad de Sevilla

Sevilla, 2016

Trabajo Fin de Grado: Robustez en sistemas de UAVs cooperativos. Simulaciones en ROS/Gazebo.

Autor: José Ángel Jiménez Doblado
Tutor: José Miguel Díaz Báñez

El tribunal nombrado para juzgar el trabajo arriba indicado, compuesto por los siguientes profesores:

Presidente:

Vocal/es:

Secretario:

acuerdan otorgarle la calificación de:

El Secretario del Tribunal

Fecha:

Agradecimientos

A mi familia, porque sin ellos nada de esto habría sido posible y para ellos todo agradecimiento es poco, a mis amigos, por haber sabido alegrarme cuando más lo he necesitado y, cómo no, a María, por haber sufrido conmigo esta etapa y, aún así, estar siempre ahí.

Asimismo, me gustaría agradecer a mi tutor, J.M.Díaz Báñez, por haberme dado la oportunidad de realizar este magnífico proyecto, a L.E.Caraballo, por su permanente e imprescindible ayuda y disponibilidad y, en definitiva, a todos los que estando presentes o no, de alguna manera han contribuido a que este sueño se haya hecho realidad.

A todos, mis más sinceros agradecimientos.

José Ángel Jiménez Doblado
Alumno de la Escuela Técnica Superior de Ingeniería de Sevilla

Sevilla, 2016

Resumen

En este proyecto se analiza la robustez de sistemas de UAVs que deben desarrollar tareas asignadas de forma cooperativa, tales como vigilancia o exploración de un área determinada y búsqueda de objetivos en entornos hostiles o peligrosos, por nombrar algunas. Para ello, se hace uso de un modelo que, no sólo permite la sincronización del conjunto de robots, sino que desarrolla una estrategia a seguir en caso de que uno o más componentes del equipo se vean obligados a abandonar. Con el objetivo de medir la tolerancia a fallos de un determinado sistema, se introduce el concepto de *resiliencia*, entendida esta en un sentido amplio como el máximo número de robots del sistema que pueden fallar sin que se comprometa la actuación global del conjunto. Para llevar a cabo el análisis de los sistemas considerados y, teniéndose en cuenta todo lo anterior, se desarrollan una serie de programas que permiten la simulación de distintas situaciones en una de las herramientas más empleadas en la actualidad en el campo de la robótica, ROS (siglas en inglés de Robot Operating System), un framework que permite el desarrollo de todo tipo de aplicaciones robóticas y su simulador 3D, Gazebo.

Abstract

In this project it is analyzed the robustness of systems of UAVs that must perform assigned tasks in a cooperative way, such as surveillance or exploration of a certain area and search for targets in hostile or dangerous environments, among others. For that, it is used a model that, not only allows the synchronization of a team of robots, but also developes a strategy to follow in case that one or more components of the team have to leave the system. In order to measure the fault tolerance of a particular system, it is introduced the concept of *resilience* which, in a broad sense, it can be understood as the largest number of robots that can leave the system without compromise the global performing of the team. In order to carry out the analysis of the considered systems and, taking into account the former explanations, it is developed a series of programs that allow the simulation of different situations in one of the most used tools in the robotic field nowadays, ROS (Robot Operating System), a framework that allow the development of any kind of robotic applications and its simulator, Gazebo.

Índice

Notación

β_{ij}	Ángulo respecto a la horizontal de la línea que conecta C_j con C_i, medido en sentido antihorario
ϕ_{ij}	Ángulo respecto a la horizontal del punto de cambio de la trayectoria *i* con respecto a la trayectoria *j*, medido en sentido antihorario
C_i	Trayectoria i-ésima en un determinado sistema de UAVs
E	Conjunto de aristas del grafo de comunicación $G = (V,E)$
$\mid E \mid$	N° de aristas en E
$F = (f,g)$	Programación de un sistema de UAVs
$G = (V,E)$	Grafo de comunicación
HW	Hardware
I_R	Isolation-resilience
m	N° de columnas de un sistema de UAVs en forma de *grid*
n	N° total de trayectorias de un sistema de UAVs en forma de árbol o n° de filas de un sistema en forma de *grid*
OSS	Open Source Software
r	Rango de comunicación
ROS	Robot Operating System
SCS	Synchronized Communication System
SW	Software
SO	Sistema Operativo
U	Conjunto de UAVs de un sistema
U_R	Uncovering-resilience
UAV	Unmanned Aerial Vehicle
V	Conjunto de nodos del grafo de comunicación $G = (V,E)$ y conjunto de trayectorias de un sistema de UAVs
WS	Workspace

1 Introducción.

Se entiende por Vehículo Aéreo No Tripulado (**UAV**, por sus siglas en inglés, Unmanned Aerial Vehicle) toda aeronave con capacidad para desarrollar tareas de forma totalmente autónoma o siendo controladas de forma remota, esto es, sin ningún piloto a bordo del vehículo.

Precisamente, el hecho de que no requieran piloto a bordo es una de sus mayores virtudes, dado que permiten desarrollar diversos tipos de misiones sin los inconvenientes habituales que supondría el uso de aeronaves convencionales, como el exponer a los pilotos a entornos hostiles o situaciones peligrosas. Ejemplo de ello pueden ser misiones de carácter militar, [1], o misiones en zonas de riesgo tras un determinado desastre, [2]. La otra gran ventaja de los UAVs es su extraordinaria capacidad para desarrollar tareas de forma cooperativa, [3], [4], con misiones que van desde la monitorización y la vigilancia, hasta la exploración y el ensamblaje de estructuras, por nombrar algunas. El uso conjunto de UAVs comporta ventajas como la posibilidad de paralelizar actividades, reduciendo así los tiempos de operación, o el reducir la incertidumbre y aumentar la tolerancia a fallos en el sistema, al tener una mayor redundancia y un mayor número de fuentes de información, [5]. Además de las dos grandes ventajas citadas anteriormente, los UAVs se caracterizan por su elevada maniobrabilidad y por presentar, generalmente, unos costes inferiores a los que implicaría un sistema tripulado.

Con las ventajas nombradas anteriormente, no es de extrañar el gran auge que ha tenido en los últimos años este tipo de vehículos, expandiéndose a todo tipo de mercados y entrando con fuerza, incluso, en el sector del ocio. Una de las consecuencias que ha provocado su uso cada vez más generalizado es el empleo de expresiones como "**drone**" o, simplemente, "**dron**", para referirse a los UAVs.

El trabajo aquí presentado se centra en aspectos relacionados con sistemas de UAVs, donde la coordinación entre sus integrantes es de vital importancia para completar correctamente las tareas asignadas y evitar colisiones entre los diferentes drones, [6]-[7]-[8]. Para ello, se hace imprescindible una fluida **comunicación** entre los distintos componentes del sistema. De la fiabilidad de dichas comunicaciones dependerá, en gran medida, la capacidad del sistema para hacer frente a contratiempos y fallos que pudieran producirse o, lo que es lo mismo, la **robustez** del sistema. No obstante, en muchas situaciones la comunicación directa entre los componentes del sistema no podrá ser garantizada (por ejemplo, situaciones en las que la zona de acción de los UAVs es mucho mayor que el alcance con el que pueden comunicarse) por lo que, en estos casos, la realización de tareas de forma cooperativa impone restricciones en las comunicaciones que pueden desarrollar los UAVs entre sí, siendo posible la comunicación entre robots vecinos únicamente cuando estos se encuentran dentro de un determinado rango de comunicaciones, [6], [9]. Todas las situaciones analizadas en el presente proyecto estarán caracterizadas por las consideraciones anteriores.

Así, considérese un conjunto de UAVs que viajan periódicamente siguiendo trayectorias cerradas al mismo tiempo que desarrollan las tareas asignadas. Cada UAV del sistema necesita informar al resto del estado de sus tareas, pero debido al alcance limitado con que pueden comunicarse, el intercambio de información sólo se produce cuando dos UAVs se encuentran dentro de un determinado rango de comunicaciones. De esta forma, cuando dos vecinos intercambian información de manera periódica se dice que están ***sincronizados***. Se entiende por sistema sincronizado aquel en el que cada pareja de vecinos está sincronizada.

La sincronización de las distintas configuraciones analizadas en el presente proyecto se ha realizado haciendo uso de un modelo presentado en [5], el cual asegura el máximo intercambio de información posible en un equipo de robots con un rango de comunicaciones limitado. En dicho modelo, la sincronización se produce de manera natural, sin necesidad de aceleraciones o cambios en la velocidad de los robots ni de

tiempos de espera entre ellos, aspectos estos que restarían realismo al modelo. En el siguiente enlace puede observarse un ejemplo de sistema sincronizado:

https://www.youtube.com/watch?v=1yFjFZQWF_M

Partiendo de un sistema sincronizado, es posible que la sincronización de este se vea afectada por el abandono de uno o varios UAVs (debido a que se agoten sus fuentes de energía, por ejemplo). En ese caso y, para reducir los perjuicios que ello pueda provocar en el funcionamiento global del equipo, cada UAV debe intercambiar su posición con la de su vecino (en caso de que haya abandonado el sistema) al llegar a la zona de comunicación con este. El punto de cambio de una trayectoria a otra será denominado en lo que sigue *punto de enlace* o bien *punto de comunicación*.

Sin embargo, debido a la estrategia adoptada, en algunos casos cuando un conjunto de robots abandona el sistema puede llegar a producirse un fenómeno indeseado: un robot pasa de una trayectoria a otra indefinidamente, al no encontrar vecinos en ninguno de los puntos de comunicación de las trayectorias que va ocupando. En tal situación, se dice que el dron en cuestión está **incomunicado**. Si dicho fenómeno se produce en todos los UAVs que en ese momento conforman el sistema, se dice que este *entra* o, simplemente, se encuentra, en un estado de **incomunicación**. Asociado a lo anterior, aparece el concepto de trayectoria *no cubierta*, esto es, cualquier trayectoria en la que existe al menos una sección que no es visitada por ninguno de los robots que en ese momento se encuentran en el sistema. En la figura 1.1 puede observarse un sistema que entra en estado de incomunicación cuando dos de los drones dejan el equipo. Nótese cómo los UAVs que permanecen en el sistema nunca llegan a encontrarse, así como que ninguna de las trayectorias es cubierta completamente.

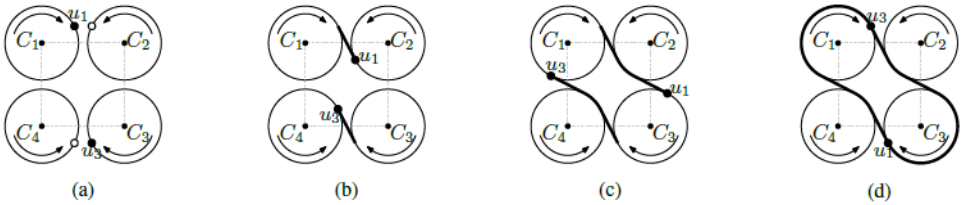


Figura 1.1 Cuando los UAVs representados por puntos blancos abandonan el sistema, los UAVs que permanecen en este, representados por puntos negros, siguen la trayectoria representada con trazo grueso. Imagen tomada de [10].

A la hora de analizar la tolerancia a fallos de sistemas de UAVs que desarrollan tareas de forma cooperativa, como es el objetivo del trabajo aquí presentado, los conceptos tanto de cobertura de trayectorias como de incomunicación deben ser, necesariamente, tenidos en cuenta. En este sentido, se define la ***resiliencia*** de un sistema de UAVs como el número máximo de robots que pueden dejar el sistema sin que se produzca incomunicación o haya trayectorias no cubiertas, [10]. Cuanto mayor sea la resiliencia de un determinado conjunto de UAVs, más robusto será el sistema y, por tanto, presentará una mayor tolerancia a los fallos.

1.1 Objetivo.

El objetivo principal del trabajo presentado es el de analizar la resiliencia de diferentes sistemas de UAVs, tanto desde el punto de vista de trayectorias no cubiertas, como desde el punto de vista de la incomunicación, lo cual dará lugar a dos conceptos de resiliencia: la ***uncovering-resilience***, o resiliencia asociada a trayectorias no cubiertas, y la ***isolation-resilience***, o resiliencia ligada a incomunicación, respectivamente. Ambos conceptos serán desarrollados en profundidad en el capítulo 3. En particular, se considerarán dos configuraciones: una en forma de árbol y otra en forma de malla o cuadrícula.

El estudio será llevado a cabo mediante el desarrollo de diferentes algoritmos que permitan la simulación 3D de los sistemas considerados, con lo que se facilite su análisis. Dichos algoritmos serán implementados en el sistema operativo robótico **ROS** (Robot Operating System), haciéndose uso de su simulador 3D **Gazebo**, para las simulaciones pertinentes. Las razones que han llevado a la elección de este software serán expuestas en profundidad en el capítulo 4. El lenguaje de programación del que se hará uso para la generación de los diferentes algoritmos es Python.

Cabe destacarse que los algoritmos desarrollados tienen un carácter mucho más general que el que aquí se muestra, permitiendo la simulación de cualquier configuración de UAVs y el ajuste de todo tipo de valores como la altitud de vuelo de los UAVs, el radio de sus trayectorias, el sentido de giro y la velocidad, etc.

Para conseguir los objetivos que se pretenden, se hará un uso intensivo de los principales resultados presentados en [5], para la sincronización de los diferentes sistemas objeto de análisis, y de [10], para el estudio de los distintos conceptos de resiliencia.

Nótese que, si bien todo el proyecto gira entorno a grupos de UAVs, los resultados obtenidos serán totalmente extrapolables a cualquier sistema de robots que operen de forma conjunta, sean aéreos, como aquí es el caso, terrestres o marítimos.

Así, el trabajo presentado está organizado como sigue: en la sección 2 se presenta el modelo teórico del que se hace uso para generar sistemas de UAVs sincronizados, así como la estrategia a seguir en caso de abandono de algún miembro del equipo para mantener la sincronización de este. En el capítulo 3, se presentan los conceptos de resiliencia que permiten cuantificar la robustez de un determinado sistema de UAVs y las expresiones que permiten analizar diferentes configuraciones. En el capítulo 4, se introduce el software del que se hará uso en las simulaciones, esto es, ROS y Gazebo, así como las razones que han llevado a la elección de estas herramientas. En la sección 5, se comenta el código desarrollado y la forma de proceder para llevar a cabo las simulaciones pertinentes. En el capítulo 6, se expone el análisis en sí de los sistemas considerados haciéndose uso de todo lo anterior para, finalmente, en el capítulo 7, exponer una serie de conclusiones y posibles líneas de investigación futuras.

2 Aspectos previos.

En este capítulo se presentan las consideraciones básicas que permiten definir y caracterizar los sistemas de UAVs, de acuerdo al modelo seguido. Asimismo, se establecen las condiciones que hacen que dichos sistemas estén sincronizados.

2.1 Modelo del sistema.

En todas las configuraciones particulares que se analizarán en el presente proyecto se sigue un modelo simplificado, en el que se considera que todos los UAVs son iguales. Además, se supone que las trayectorias de cada uno de ellos, así como su velocidad (la cual será constante) y rango de comunicación, también son iguales.

Las trayectorias se han tomado circulares y no tienen ningún punto en común entre ellas, lo cual permite evitar colisiones entre UAVs y aprovechar mejor las capacidades del sistema, impidiendo que se produzca cualquier tipo de solapamiento entre las tareas de los distintos drones. Por simplicidad, se consideran inicialmente círculos de radio unidad.

Según las consideraciones anteriores, un sistema queda completamente definido por una terna de valores (V,U,r), donde $V = \{C_1,...,C_n\}$ representa el conjunto de trayectorias, $U = \{u_1,...,u_n\}$ el conjunto de UAVs y r el rango de comunicaciones de estos. Por otra parte, todos los UAVs tardarán el mismo tiempo en completar una vuelta, dicho tiempo será denominado en lo que sigue ***periodo del sistema***.

En cualquier sistema, se define el ***grafo de comunicación*** del sistema como el grafo $G = (V,E)$, cuyos nodos son los elementos de V y dos nodos están conectados si, y sólo si, la distancia entre sus centros es menor o igual a 2+r (con $r < 0.5$). La línea que une dos nodos adyacentes, C_i y C_j, en caso de que la hubiera, se denota por (i,j). Nótese que la condición que determina si dos nodos están o no conectados es la condición mínima necesaria que permite que haya comunicación en, al menos, un punto de la trayectoria entre dos robots, de acuerdo al rango de comunicaciones que los caracteriza, r. En la figura 2.1, puede observarse el grafo de comunicación asociado a un determinado sistema de UAVs.

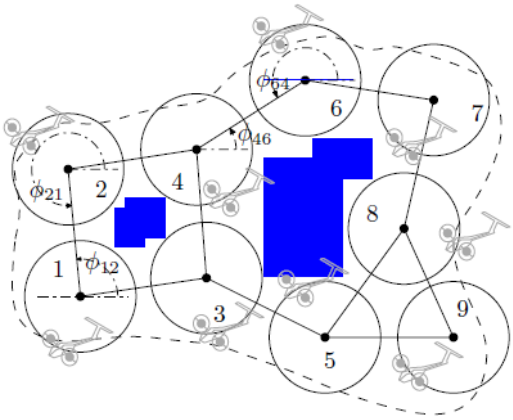


Figura 2.1 Sistema de UAVs y su grafo de comunicación asociado. Imagen tomada de [5].

Para calcular la posición de cada UAV, se hace uso del ángulo que forma éste con el eje horizontal positivo cuyo origen es el centro de su trayectoria, medido en sentido contrario a las agujas del reloj. Este valor por tanto se encontrará dentro del rango $[0,2\pi)$. De esta forma, el punto de enlace entre dos nodos conectados, C_i con respecto a C_j, denominado como ϕ_{ij}, estará definido por el punto de C_i más cercano a C_j. Es inmediato comprobar que $\phi_{ji} = \phi_{ij} \pm \pi$.

En un sistema de UAVs con grafo de comunicación $G = (V,E)$ se dice que dos robots, volando en C_i y C_j respectivamente, son **vecinos** si $(i,j) \in E$, esto es, si están conectados. Se dice que dos vecinos están **sincronizados** si ambos llegan al mismo tiempo al punto de enlace común, ϕ_{ij} y ϕ_{ji}, respectivamente. Necesariamente, el número máximo de vecinos sincronizados en un determinado sistema estará limitado por $|E|$. Se dice que un sistema de UAVs está sincronizado si el número de vecinos sincronizados es exactamente $|E|$.

Nótese que, debido a las consideraciones anteriores, basta con conocer la posición inicial y el sentido de movimiento (horario o antihorario) de un dron para conocer su posición en cualquier momento.

Se entiende por ***programación*** de un sistema o, simplemente, *programa*, a la pareja de funciones $F = (f,g)$:

$$f : V \to [0,2\pi)$$

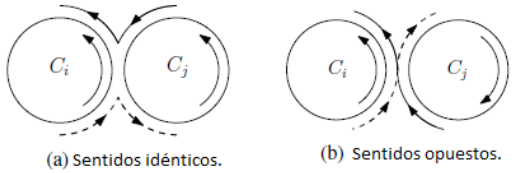
$$g : V \to \{1,-1\}$$

tal que $\forall C_i \in V$ el valor $f(C_i) \in [0,2\pi)$ representa la posición inicial para el UAV que inicia en C_i y $g(C_i) \in \{1,-1\}$ representa el sentido de movimiento de este. Se denomina ***programa de sincronización*** a aquel programa que hace que el sistema se encuentre sincronizado.

2.2 Sincronización del sistema.

Teniendo en cuenta los aspectos comentados en la sección anterior, se trata ahora de conseguir un método eficaz que permita obtener un programa de sincronización para cualquier sistema de UAVs, en particular, un programa que permita sincronizar los sistemas que serán objeto de estudio en el capítulo 6.

En primer lugar y, como se comentó en el capítulo 1, el proyecto se centra en sistemas de UAVs, esto es, robots aéreos. Por este motivo, es necesario tener en cuenta ciertas características inherentes a este tipo de vehículos y que influyen notablemente en la robustez que puede llegar a alcanzar el conjunto. En efecto, para llevar a cabo la estrategia de intercambio de trayectorias que se ha decidido seguir en caso de abandono de algún componente del equipo con un suficiente grado de robustez, el paso de un camino a otro debe ser suficientemente *suave* como para que las importantes **restricciones cinemáticas** asociadas a todo vehículo aéreo (problemas de entrada en pérdidas, limitaciones en la carga alar, mínimo radio de giro...) no supongan un impedimento. Como es inmediato observar en la figura 2.2, dicha maniobra de intercambio es mucho más suave en sistemas donde los vecinos vuelan en **sentidos opuestos**, que en aquellos conjuntos en los que los vecinos vuelan en el mismo sentido. En otras palabras, los sistemas en los que los intercambios se producen entre trayectorias con sentidos opuestos son mucho más robustos que aquellos en los que los intercambios se producen entre trayectorias con sentidos idénticos. Por esta razón, en todos los sistemas que serán analizados en el presente proyecto, los vecinos volarán en sentidos opuestos, favoreciendo así las prestaciones del sistema. Es fácil deducir que, de acuerdo a las consideraciones anteriores, la función g vendrá dada por: $g(C_i) = -g(C_i)\ \forall (i,j) \in E$.



(a) Sentidos idénticos. (b) Sentidos opuestos.

Figura 2.2 Intercambio de trayectorias entre UAVs. Nótese cómo el intercambio en el caso b) es mucho más suave que en a). Imagen tomada de [5].

Además, se debe tener en cuenta que la sincronización completa de un conjunto de UAVs, cuyos miembros de cada pareja de vecinos vuelan en direcciones opuestas, sólo es posible si el grafo de comunicación asociado

es ***bipartito***, esto es, se pueden dividir los UAVs del sistema en dos grupos de acuerdo a sus sentidos de giro sin violar la relación de sentidos opuestos entre vecinos, [5].

Por otra parte, en lo que respecta a la función f, de [5] se tiene que dos vecinos, C_i y C_j, bajo las condiciones expuestas están sincronizados si, y sólo si:

$$f(C_j) = 2\beta_{ij} - f(C_i) \pm \pi \tag{2.1}$$

donde β_{ij} es el ángulo que forma el eje horizontal positivo con origen en el centro de C_j y la línea (i,j) perteneciente al grafo del sistema, medido en sentido antihorario. En la figura 2.3 se observa dicho ángulo.

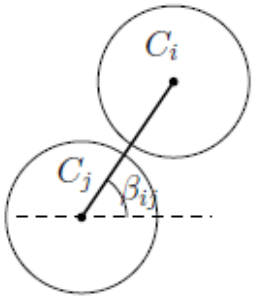


Figura 2.3 Definición del ángulo β_{ij}. Imagen tomada de [5].

De la ecuación 2.1 se deduce pues la condición que debe cumplir todo sistema de UAVs para estar sincronizado:

$$f(C_j) = 2\beta_{ij} - f(C_i) \pm \pi \; \forall (i,j) \in E$$

Finalmente, cabe preguntarse para un sistema determinado, cuándo dicha sincronización es posible. En este sentido, el teorema 4.2 de [5] establece las condiciones necesarias y suficientes para que exista un programa de sincronización en un conjunto de robots en el que cada pareja de vecinos vuelan en direcciones opuestas.

Teorema 2.2.1 (Teorema 4.2 de [5]) *Sea $G = (V,E)$ el grafo de comunicación de un sistema de UAVs cuyos vecinos vuelan en sentidos opuestos. Existe un programa de sincronización para dicho sistema si, y sólo si, G es bipartito y cada ciclo $\langle C_{i_1}, C_{i_2}, ..., C_{i_{2k}}, C_{i_1} \rangle$ en G satisface:*

$$\beta_{i_1 i_2} - \beta_{i_2 i_3} + \beta_{i_3 i_4} - \beta_{i_4 i_5} + ... + \beta_{i_{2k-1} i_{2k}} - \beta_{i_{2k} i_1} = 2m\pi, \;\; m \in \mathbb{N}$$

Adicionalmente, de acuerdo al teorema 4.3 de [5] se tiene que en un sistema sincronizado de acuerdo a las consideraciones anteriores, cada trayectoria está ocupada, como mucho, por un único UAV.

Se entenderá por ***sistema de comunicación sincronizado (SCS)*** a todo sistema de UAV sincronizado en el que los vecinos vuelan en direcciones opuestas y en el que el intercambio de trayectorias se supone realizado de forma instantánea. La consideración de que el intercambio de trayectorias se produce instantánemante se realiza para llevar a cabo el estudio teórico puesto que, en la práctica, el robot que debe intercambiar su trayectoria incrementará su velocidad (acelerará) durante dicha maniobra para permitir mantener la sincronización del sistema. Esta será, por tanto, la forma en que se implementen los cambios en las simulaciones que se desarrollarán posteriormente. Nótese que, en caso de abandono de algún componente del equipo y, para mantener la sincronización del sistema, el UAV que cambia de trayectoria debe seguir volando (una vez efectuada la maniobra de intercambio) de acuerdo a la programación que seguía el UAV al que sustituye.

Todos los conjuntos de UAVs que se analizarán en el presente proyecto se considerarán SCS.

2.3 Generalización del modelo.

Aunque el modelo seguido en este proyecto es un modelo simplificado, presentado anteriormente, en [5] puede encontrarse la generalización de dicho modelo para el caso en que las trayectorias no sean circulares y no todos los robots sean iguales entre sí, situaciones estas más realistas.

Así, considérese un equipo de *n* robots con diferentes características y desarrollando sus respectivas tareas a lo largo de un conjunto de trayectorias cerradas no necesariamente circulares. En la realidad, se tendrá un cierto control sobre las velocidades de dichos robots (en el modelo simplificado se consideraba la velocidad

de cada UAV constante e igual para todos los componentes), pudiéndose acelerar o frenar en diferentes tramos de la trayectoria. En este sentido, la clave para generalizar el modelo simplificado seguido es forzar a todos los componentes del equipo a que tarden (aproximadamente) el mismo tiempo en completar una vuelta a lo largo de su trayectoria. En el modelo simplificado, dos vecinos llegaban al mismo tiempo al punto de comunicación, mientras que en un modelo más realista se debe permitir un cierto margen de error para garantizar el correcto funcionamiento del sistema. Esto puede conseguirse sin más que permitir a cada UAV del equipo comparar, en intervalos regulares, su posición actual con la posición en la que debería encontrarse para mantener la sincronización con sus vecinos, ajustando en función de ello su velocidad en cada tramo de la trayectoria.

3 Resiliencia.

Como se comentó en el capítulo 1, una forma de conocer la robustez de un determinado sistema de UAVs es a través del concepto de resiliencia, el cual mide de alguna forma la tolerancia a fallos del conjunto.

En este capítulo, se definen formalmente los conceptos que permitirán analizar los sistemas de UAVs considerados, además de establecerse las relaciones que permiten obtener la resiliencia de un determinado conjunto en función de su grafo de comunicación.

3.1 Conceptos básicos.

De acuerdo a lo expuesto en el capítulo 2, los sistemas de UAVs analizados serán SCS. Para el estudio de la robustez de estos conjuntos, se considerará que algunos componentes se ven obligados a abandonar sus tareas, por ejemplo, para recargar sus baterías. Dichos sistemas serán denominados **SCS parciales**, esto es, un SCS en el que falta alguno de sus integrantes.

En un SCS parcial, se dice que un robot u está **incomunicado** si $\forall(i,j) \in E$, cada vez que u llega a ϕ_{ij}, la trayectoria C_j está vacía. Si todos los UAVs que conforman el sistema en un momento dado se encuentran en dicha situación, se dice que el sistema se encuentra en un estado de **incomunicación**. En la figura 3.1 se pueden observar diversos sistemas que entrarían en incomunicación si los robots representados por puntos blancos abandonan el equipo.

En un SCS parcial, se dice que una trayectoria es ***cubierta*** si todos los puntos que la forman son recorridos periódicamente por algún robot. Análogamente, una trayectoria es ***no cubierta*** si existe en ella algún punto no visitado por ningún robot. Nótese que el hecho de que el sistema se encuentre en estado de incomunicación no va ligado necesariamente a la existencia en él de trayectorias no cubiertas, como puede observarse en los tres primeros casos de la figura 3.1, en los cuales no existen trayectorias no cubiertas aún estando el conjunto incomunicado.

En un SCS, se define la ***uncovering-resilience***, o resiliencia ligada a trayectorias no cubiertas, como el número máximo de UAVs que pueden abandonar el equipo de forma que todas las trayectorias están cubiertas por los robots que permanecen en él.

De forma similar, se define para un SCS la ***isolation-resilience*** o resiliencia asociada a incomunicación, como el número máximo de UAVs que pueden abandonar el equipo de forma que se asegure que el sistema no pasa a estar en estado de incomunicación.

Cabe destacar que **la resiliencia de un determinado sistema**, en sus dos vertientes, no depende del momento en que los UAVs abandonan el equipo, ni del sentido de movimiento de estos, ni de sus posiciones de inicio, **depende única y exclusivamente del grafo de comunicación que caracteriza al sistema**, [10].

3.2 Uncovering-resilience.

Como se ha comentado en la sección anterior, la uncovering-resilience, U_R, o resiliencia asociada a trayectorias no cubiertas de un determinado equipo de UAVs depende únicamente del grafo de comunicación de dicho sistema, $G = (V,E)$, siendo pues independiente, entre otras cosas, del programa de sincronización empleado.

De acuerdo a lo expuesto en el capítulo 1, el análisis de la resiliencia se va a desarrollar sobre sistemas cuyos grafos de comunicación son, o bien un árbol, o bien una cuadrícula, por lo que siguiendo el trabajo desarrollado en [10], se tienen los siguientes resultados:

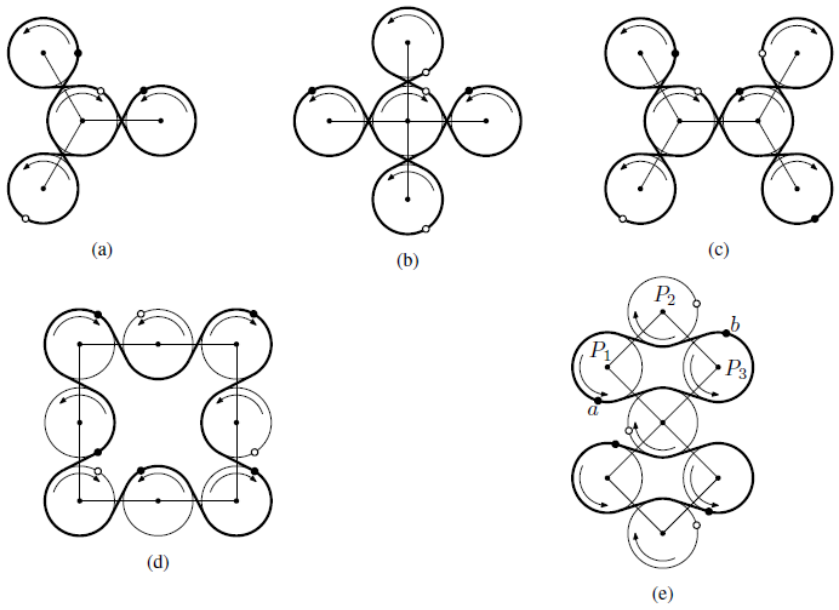


Figura 3.1 Si los UAVs representados por puntos blancos abandonan el sistema, el sistema pasa a encontrarse en incomunicación. En tal caso, los robots restantes, ilustrados como puntos negros, recorrerían la trayectoria representada por trazo grueso. Imagen tomada de [10].

Uncovering-resilience de árboles. La resiliencia asociada a trayectorias no cubiertas de un sistema con n trayectorias, cuyo grafo de comunicación es un árbol, viene dada por:

$$U_R = n - 1 \tag{3.1}$$

En la figura 3.2 puede observarse un ejemplo de tal sistema.

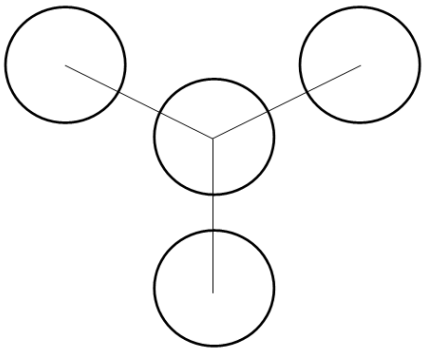


Figura 3.2 Ejemplo de sistema con $n = 4$ cuyo grafo de comunicación es un árbol.

Uncovering-resilience de mallas. Un sistema cuyo grafo de comunicación tiene forma de malla o cuadrícula es aquel que presenta una disposición en que hay $n \times m$ trayectorias, distribuidas en n filas y m columnas. En dicha configuración, cada trayectoria se designa como (i,j), donde i y j indican, respectivamente, la fila y la columna en que se encuentra. En un grafo de comunicación en forma de malla, la trayectoria (i,j) se encuentra conectada con las trayectorias $(i-1,j)$, $(i,j-1)$, $(i+1,j)$ e $(i,j+1)$, si existen. En la figura 3.3 puede observarse un ejemplo de tal sistema.

La resiliencia asociada a trayectorias no cubiertas en sistemas cuyo grafo de comunicación tiene forma de malla $n \times m$ viene dada por la expresión:

$$U_R = \frac{n \cdot m}{MCD(n,m)} - 1 \tag{3.2}$$

donde $MCD(n,m)$ representa el máximo común divisor de n y m.

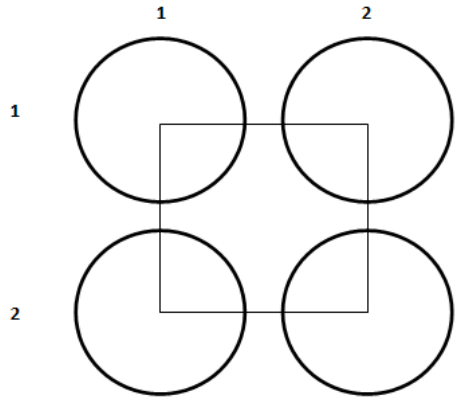


Figura 3.3 Ejemplo de sistema con grado de comunicación en forma de malla 2×2.

3.3 Isolation-resilience.

Al igual que ocurre con la resiliencia asociada a trayectorias no cubiertas, la resiliencia ligada a incomunicación o, lo que es lo mismo, la isolation-resilience, I_R, sólo depende del grafo de comunicación del sistema de UAVs en cuestión.

De forma análoga a como se ha procedido en la sección anterior y, siguiendo los resultados expuestos en [10], se tiene:

Isolation-resilience de árboles. En un sistema cuyo grafo de comunicación, $G = (V,E)$, es un árbol, la resiliencia asociada a incomunicación debe ser tal que:

$$I_R \geq [n/2] - 1 \tag{3.3}$$

Nótese que, en esta ocasión, sólo ha sido posible obtener una condición a cumplir por I_R, debido a la mayor dificultad que entraña este problema.

Isolation-resilience de mallas. En un sistema cuyo grafo de comunicación, $G = (V,E)$, tiene forma de cuadrícula $n \cdot m$, entendiéndose por ello lo ya expuesto en la sección anterior, se tiene que la isolation-resilience es:

$$I_R = n \cdot m - min(n,m) - 1 \tag{3.4}$$

4 ROS.

En este capítulo se pretende dar una visión global de qué es ROS y las motivaciones que han llevado a usarlo en este proyecto, así como del simulador Gazebo. Se describen, además, los principales elementos que caracterizan ROS y de los que se ha hecho uso a la hora de llevar a cabo la implementación, con el fin de facilitar su comprensión.

4.1 Introducción a ROS.

ROS, siglas en inglés de Robot Operating System, es una plataforma software (SW), también denominado *framework*, que proporciona todo tipo de librerías y herramientas para el desarrollo de aplicaciones específicamente robóticas.

Tanto ROS, como todos los recursos que este ofrece, son de **código abierto (OSS)**, siendo su uso totalmente gratuito. Esta fue, de hecho, la filosofía que impulsó su creación: una plataforma que permitiera el desarrollo de sistemas robóticos de forma mucho más óptima gracias a la aportación de todos sus usuarios, compartiéndose todo tipo de herramientas y programas, de forma libre, reutilizables, adaptables a las necesidades de cada uno e integrables con todo el software robótico existente y futuro, [12]. Muestra de ello es la existencia de *paquetes* que permiten el uso en ROS de aplicaciones robóticas ya existentes, de código abierto, como Gazebo, OpenCV o Stage. Esta ha sido, sin duda, una característica fundamental que ha permitido el desarrollo del presente proyecto, al haberse hecho uso de diversas herramientas y aplicaciones proporcionadas por diversas fuentes que han simplificado notablemente el trabajo a realizar, como se verá más adelante.

A pesar de su nombre, ROS no es un sistema operativo (SO) en sí, como demuestra el hecho de que requiere ser instalado sobre otro SO (preferentemente sobre sistemas Linux, como Ubuntu), si bien sí que proporciona servicios estándares como abstracción del hardware (HW), comunicación a través de mensajes y gestión de archivos, por nombrar algunos, siendo por ello también denominado Meta-Sistema Operativo, [13]. En la figura 4.1 puede observarse, a grandes rasgos, la relación ROS-SO.

A grosso modo, podría decirse que ROS ofrece, [11]:

1. Un conjunto de drivers que permiten la lectura de datos y el envío de órdenes a los diferentes motores y actuadores del robot, soportando una gran variedad de HW como, por ejemplo, la cámara Kinect de Microsoft.
2. Una gran cantidad de algoritmos que permiten la construcción de mapas, la navegación en estos, representar e interpretar dados tomados por los sensores, manipulación de objetos, etc.
3. Toda la infraestructura computacional que permite conectar varios componentes de un robot e incorporar los algoritmos pertinentes.
4. Un gran conjunto de herramientas que facilitan la labor de depuración de errores.
5. En un nivel superior, podría destacarse la existencia de recursos importantísimos como una *wiki* dedicada expresamente a ROS, [14], donde pueden encontrarse, entre otras cosas, todo tipo de tutoriales, y un foro oficial de preguntas, [15], soportado en gran parte gracias a la colaboración de investigadores, expertos y usuarios de todo el mundo. Estos recursos han sido una pieza clave en el desarrollo de este proyecto.

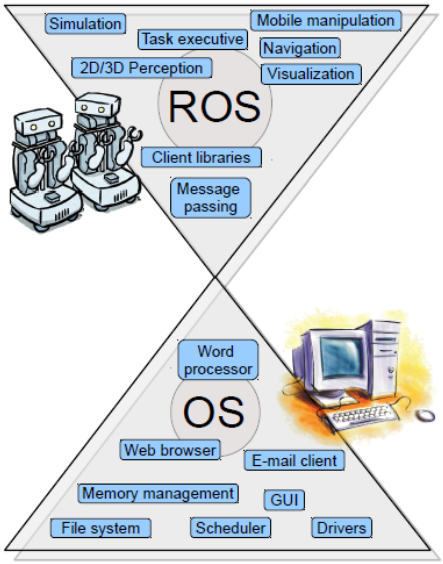


Figura 4.1 Relación, a grandes rasgos, entre ROS y un SO habitual. Imagen tomada de [13].

Debido a su gran desarrollo, existen diferentes versiones de ROS. En este proyecto, se ha hecho uso de la versión **ROS Indigo**, por ser la más estable al momento de comenzarse.

Gazebo Gazebo es un entorno de simulación 3*D* para aplicaciones robóticas. Su integración con ROS se realiza a través de la instalación de un paquete, *gazebo_ros* [1], el cual permite una comunicación bidireccional entre ROS y Gazebo.

Gazebo permite la creación y simulación de todo tipo de modelos robóticos, ya sea un único robot o conjuntos de ellos, como es el caso del trabajo aquí presentado. Permite, además, la creación de todo tipo de escenarios donde desarrollar la simulación. Como muestra de ello, en la figura 4.2, se observa la simulación de un entorno con diversos elementos así como un modelo robótico completo, en concreto, un modelo del *Fecth robot*, mientras que en la figura 4.3 se observa una captura tomada de una de las simulaciones desarrolladas para este proyecto de un sistema formado por 4 UAVs.

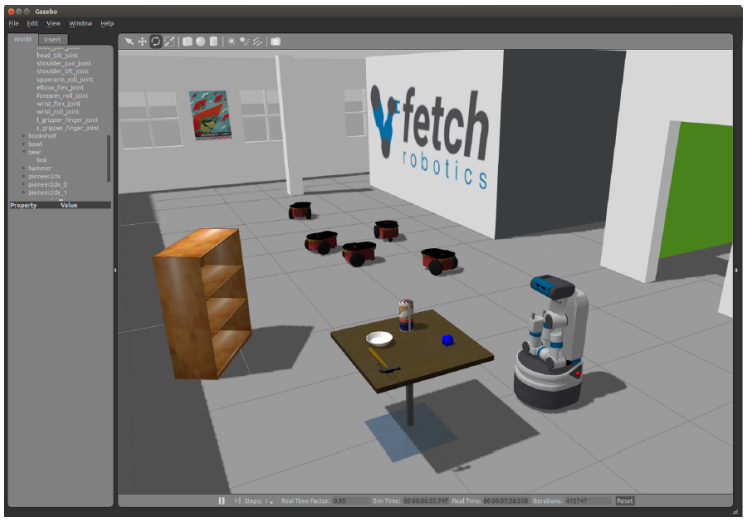


Figura 4.2 Simulación en Gazebo de un entorno de diversos elementos y un modelo del Fecth robot. Imagen tomada de [11].

Dadas sus grandes prestaciones, ha sido el simulador empleado para el desarrollo del presente proyecto.

[1] *http://wiki.ros.org/gazebo_ros_pkgs*

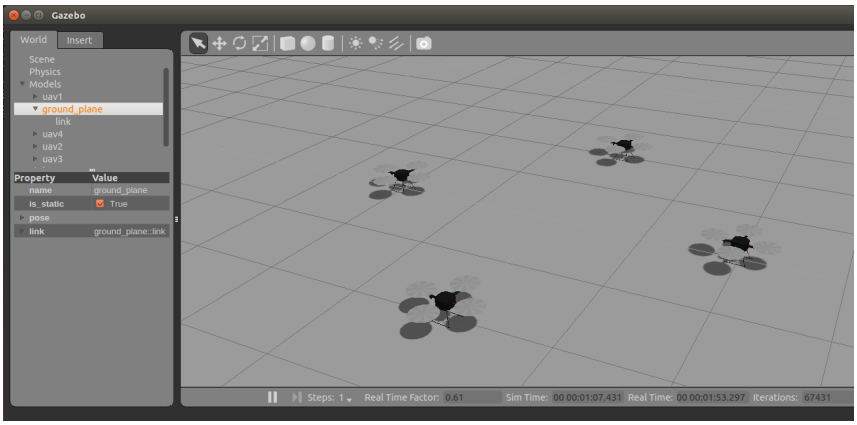


Figura 4.3 Simulación en Gazebo de un sistema de 4 UAVs.

4.2 ¿Por qué ROS/Gazebo?

Tras las consideraciones anteriores, cabe preguntarse ¿Por qué elegir ROS/Gazebo? ¿Por qué desarrollar simulaciones en ROS/Gazebo?

En primer lugar y, como razón de peso, porque tanto ROS como todo el material asociado es código libre, con lo que los gastos respecto a otro tipo de SW se reducen sensiblemente.

En segundo lugar y, sin ser menos importante, porque proporciona un entorno ideal para el desarrollo de todo tipo de aplicaciones robóticas reales. Si bien lo desarrollado para este proyecto han sido una serie de algoritmos que permiten la simulación de diferentes sistemas de UAVs siguiendo unas determinadas estrategias expuestas en los capítulos anteriores, gracias a haberlo desarrollado en ROS sólo se requerirían unos mínimos cambios en dichos algoritmos para posibilitar la experimentación real y posterior implantación de los sistemas simulados. Es decir, tiene una clara orientación al mundo real, lejos, por tanto, de simples simulaciones en las que sólo se busque algún resultado de carácter gráfico.

Además, gracias a su filosofía, el hecho de que exista una gran comunidad de expertos e investigadores detrás de él aportando todo tipo de herramientas y programas adaptables a las necesidades de cada usuario, hace que se puedan conseguir los objetivos propuestos de forma mucho más simple, rápida y eficiente.

Otro aspecto muy a tener en cuenta de ROS es que admite diferentes lenguajes de programación, como $C++$ ó Python, siendo este último el que se ha usado en el presente proyecto.

En definitiva, gracias a sus enormes cualidades, ROS se ha establecido en los últimos años como el SW de programación robótica por excelencia, siendo la herramienta común de todos los avances realizados en dicho campo y permitiendo la puesta en marcha de infinidad de aplicaciones orientadas al mundo real, razón de más para desarrollar el presente proyecto haciendo uso de él.

4.3 Conceptos básicos.

ROS presenta una estructura modular, en la que los programas se crean a partir de paquetes que contienen los códigos necesarios para desarrollar las aplicaciones que requiere el usuario.

Sin ánimo de ser exhaustivos ni de realizar un manual detallado de ROS, se presentan a continuación los principales elementos de los que se ha hecho uso para el desarrollo del presente trabajo y que ayudarán a comprender el código desarrollado.

Nodos Los nodos son módulos ejecutables que llevan a cabo una determinada acción y que permiten publicar y/o suscribirse a distintos *topics* y proporcionar o usar *servicios*. Cada nodo del sistema debe tener un nombre único, para permitir su identificación inequívocamente.

Los nodos en ROS se escriben haciéndose uso de bibliotecas específicas como *Rospy* si se escribe en código Python, como aquí es el caso, o Roscpp, si se escribe en código $C++$.

Master Es el nodo principal o nodo maestro. Proporciona el registro de nombres, permitiendo así al resto de nodos del sistema comunicarse entre sí, intercambiando mensajes, solicitando servicios, etc. En otras palabras, al iniciarse un nodo, este se conecta al nodo maestro, dándole detalles de los mensajes

que está publicando así como información acerca de los mensajes a los que desea suscribirse. Sin el nodo maestro, por tanto, no puede funcionar ningún sistema.

Es importante destacar que la comunicación se produce entre nodos, de forma que el nodo maestro sólo es usado por estos para saber dónde encontrar al resto de nodos con los que comunicarse.

El nodo maestro, además, actúa de **servidor de parámetros**, almacenando diferentes datos y permitiendo su lectura por parte de los nodos que lo requieran.

Generalmente se ejecuta mediante la orden **roscore**, siendo la primera instrucción que debe introducirse al abrirse ROS.

Mensajes Estructura de datos que permite la comunicación entre distintos nodos, enviadas a través de los *topics*. Ejemplos de mensajes son int8 (entero de 8 bits) o float64 (flotante de 64 bits).

Topics o temas Un *topic* no es más que un nombre para un determinado flujo de mensajes con un tipo definido. Los topics permiten llevar a cabo el mecanismo de comunicación publicar/suscribir. Así, un nodo envía mensajes publicándolos en un determinado topic y, de la misma forma, un nodo recibe mensajes suscribiéndose al topic correspondiente.

Nótese que un mismo topic puede tener múltiples suscriptores, y un mismo nodo puede publicar o suscribirse a varios topics.

En ROS, todos los mensajes enviados a través del mismo topic deben ser del mismo tipo.

A modo de ejemplo, en la figura 4.4, pueden observarse algunos topics empleados durante una simulación de un sistema de UAVs, concretamente, son un extracto de topics pertenecientes al UAV número 1 del sistema.

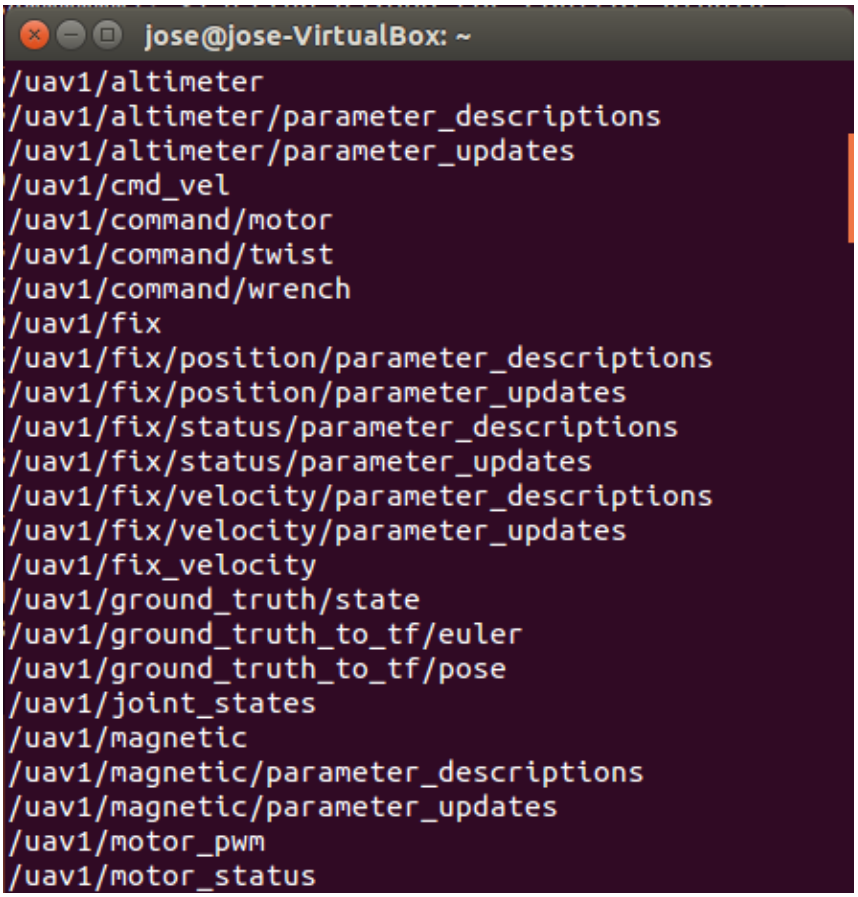


Figura 4.4 Ejemplo de topics empleados durante una simulación de un sistema de UAVs, en particular, son topics pertenecientes al UAV número 1 del equipo.

4.4 Sistema de archivos.

Antes de escribir cualquier código en ROS se necesita configurar un *workspace* (WS) o espacio de trabajo, en el que residirá dicho código. Un WS es, simplemente, un conjunto de directorios o carpetas en las cuales se encuentra un determinado conjunto de códigos ROS. Puede haber múltiples WS, pero sólo se puede trabajar en uno de ellos simultáneamente. Para crear un WS, se deben seguir los siguientes pasos:

1. Abrir un terminal ROS e introducir **source /opt/ros/indigo/setup.bash**. Dicha orden permite el uso de comandos básicos de ROS que serán necesarios a continuación.
2. Introducir: **mkdir -p ∼/wanderbot_ws/src**
3. Introducir: **cd ∼/wanderbot_ws/src**
4. Introducir: **catkin_init_workspace**

Las últimas tres sentencias se encargan de crear e inicializar el WS deseado. En ellas, se ha llamado al WS a crear *wanderbot_ws*, siendo totalmente indiferente el nombre que se le dé. Además de crear el WS, las órdenes anteriores crean, entre otras cosas, un directorio dentro del WS llamado *src*.

Tras la creación del WS, introducir las siguientes dos órdenes para terminar de configurarlo:

5. **cd ∼/wanderbot_ws**
6. **catkin_make**

Para albergar el código desarrollado para este proyecto, se va a crear una carpeta o *package* (al cual se ha decidido llamar *wanderbot*) dentro del directorio *src* del WS recién creado. Con este fin, introducir los siguientes comandos:

7. **cd ∼/wanderbot_ws/src**
8. **catkin_create_pkg wanderbot rospy**

La primera línea, permite situarse en el directorio en el que se quiere crear el *package* (en este caso el directorio *src* del WS *wanderbot_ws*). Tras ello, la segunda línea crea el *package* en sí. Nótese que, al crear un *package*, se introduce también la librería de cliente de la que se va a hacer uso, *rospy* en el caso del presente proyecto, al haberse hecho uso del lenguaje de programación Python, como ya se ha comentado anteriormente. Al crearse el package anterior, también se crea automáticamente dentro de este una carpeta llamada *src* (no confundir con la otra carpeta homónima nombrada anteriormente). En esta carpeta se guardarán algunos de los archivos creados en el presente proyecto, como se expondrá en el siguiente capítulo. Otros archivos sin embargo, serán guardados en una carpeta llamada *launch*, dentro del *package* recién creado. Para crear dicha carpeta, proceder como sigue:

9. **cd ∼/wanderbot_ws/src/wanderbot**
10. **mkdir launch**

Una vez desarrollado lo anterior, se debe tener, entre otros archivos auxiliares que crea automáticamente ROS al seguir el proceso enunciado: un WS llamado *wanderbot_ws*, dentro del WS un directorio *src*, dentro de dicho directorio una carpeta llamada *wanderbot* y, finalmente, dentro de esta, dos carpetas, *src* y *launch*, respectivamente. En estas dos últimas carpetas se guardan los archivos desarrollados de acuerdo a la distribución que se expondrá en el capítulo siguiente. En la figura 4.5, se muestra la disposición de dichas carpetas, además de ciertos archivos auxiliares creados por ROS.

4.5 Hector quadrotor

Como se ha comentado anteriormente, una de las ventajas principales de ROS es la posibilidad de obtener todo tipo de programas y aplicaciones de la comunidad de forma fácil y gratuita, facilitando así la labor de desarrollar toda una aplicación robótica completa, como la que aquí se pretende. En este sentido y, poniéndose de manifiesto dicha ventaja, a la hora de desarrollar el código que permite las simulaciones se ha tomado un

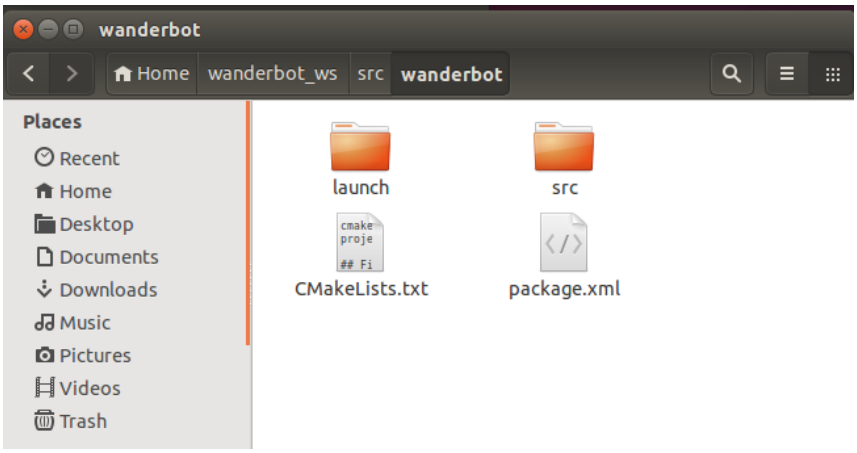


Figura 4.5 Carpetas *src* y *launch*, en las que se guardan los archivos desarrollados en el presente proyecto. Aparecen, además, otros archivos auxiliares creados por ROS para su correcto funcionamiento.

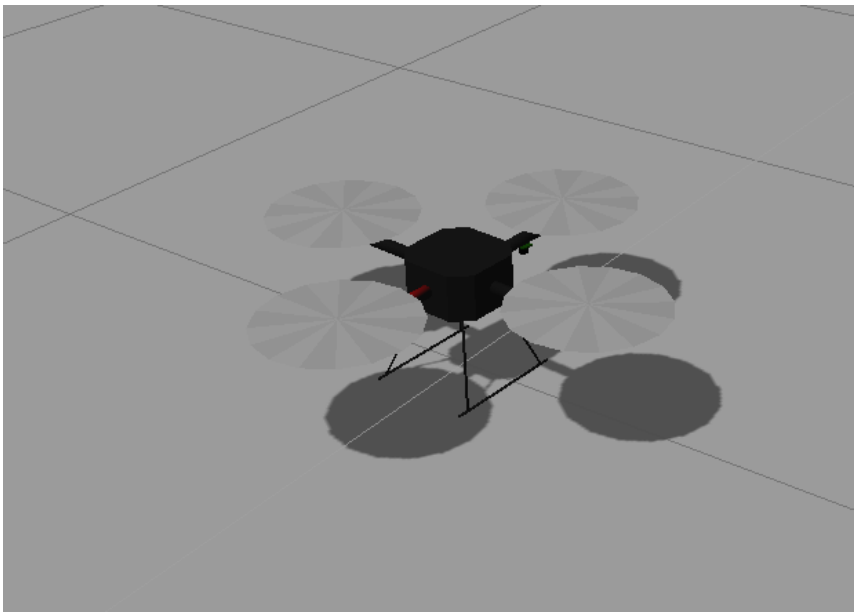


Figura 4.6 Modelo de UAV proporcionado por el paquete *hector_quadrotor*.

modelo completo de UAV desarrollado por los expertos de la Universidad Técnica de Darmstadt, Johannes Meyer y Stefan Kohlbrecher, llamado *hector_quadrotor*[2]. En la figura 4.6 se muestra dicho modelo de UAV.

Así, instalando en ROS el paquete proporcionado por dichos autores, se tiene completamente definido el modelo de UAV con el que trabajar, de forma que todo el esfuerzo a realizar se ha centrado en la implementación del modelo expuesto en los capítulos 2 y 3 y las diferentes estrategias adoptadas en caso de fallo de algún componente del equipo, como será descrito en el siguiente capítulo.

[2] *https://github.com/tu-darmstadt-ros-pkg/hector_quadrotor*

5 Implementación en ROS/Gazebo.

En este capítulo se expone cómo se ha procedido a la hora de desarrollar el código que permite el análisis en ROS/Gazebo de los diferentes sistemas de UAVs considerados, el cual se adjunta al final de este trabajo para su consulta. De acuerdo a lo comentado en la sección 4.5, la implementación se enfocará en la puesta en marcha del sistema según el modelo expuesto en los capítulos 2 y 3 así como de las estrategias a adoptar en caso de fallo de algún componente del equipo.

El código que permite la simulación y análisis de los diferentes sistemas de UAVs está recogido en cuatro archivos, a saber:

1. simulacion_subida_y_orientacion.py
2. simulacion_navegacion_sincronizada.py
3. ausentes_publisher.py
4. simulacion_generador_sistema_uavs.py

5.1 simulacion_subida_y_orientacion.py

Este archivo, alojado en la carpeta *src* del directorio *wanderbot*, se encarga de iniciar un nodo al ejecutarse que hace subir al UAV hasta una determinada altura inicial y, tras ello, lo orienta de forma aproximadamente tangente a la trayectoria circular que debe recorrer. Hecho esto, el nodo detiene al UAV en dicha posición y con dicha orientación.

Nótese, por tanto, que dicho archivo deberá ejecutarse para cada UAV que conforme el sistema, de forma que todos suban y se orienten de acuerdo al código anterior. Para lograr ejecutar simultáneamente este archivo para todos los UAVs que componen el sistema, de forma que todos suban y se orienten al mismo tiempo, se hace uso de archivos .xml, como se explicará en la sección 5.4.

En lo que respecta al código en sí, se comienza importando diversas librerías y funciones, así como ciertos tipos de mensajes que serán utilizados por el nodo (líneas 1-8). Los tipos de mensajes que utilizará este nodo son *Twist*, empleados para publicar las velocidades tanto lineales como angulares que debe adquirir el UAV, y mensajes de tipo *Odometry*, que son utilizados por el nodo para conocer la posición y la orientación en la que se encuentra el UAV. En la figura 5.1 se puede observar un extracto de un mensaje de tipo Odometry en el que se observa la información comentada. Tras ello, en las líneas (9-12) se fijan algunos parámetros para esta primera subida y orientación inicial. En la línea 13, se inicia el nodo en sí, con el nombre *simulacion_subida_y_orientacion*. Acto seguido, se define la función que permite al nodo la lectura de los datos de odometría correspondientes (cuyos mensajes son de tipo *Odometry* por lo comentado)(14-29) y se suscribe para ello al topic *ground_truth/state*, en el que es publicada dicha información (30-32). En la línea 33, se establece el topic en el que va a publicar información el nodo, en este caso lo hará a través del topic *cmd_vel*, y el tipo de mensaje en que lo hará, *Twist*. La información que publicará este nodo a través de dicho topic es, como se ha comentado anteriormente, las velocidades angulares y lineales que debe seguir el UAV en cuestión para realizar correctamente la subida y orientación iniciales previa a la navegación en sí, al ser este el objetivo de este nodo. Una vez establecida la publicación, se fijan algunos parámetros necesarios (34-39) y se inicia un bucle (40-78) el cual, usando permanentemente las lecturas de odometría, realiza las operaciones necesarias para calcular la velocidad a seguir por el robot para conseguir el propósito deseado. Finalmente, una vez calculadas las velocidades, las publica en la línea 76 para que el UAV se mueva de acuerdo a ellas.

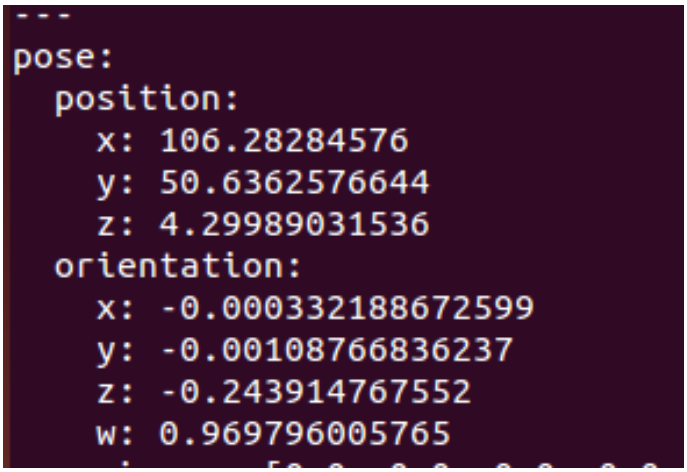


Figura 5.1 Extracto de un mensaje de tipo *Odometry* en el que se puede ver información relativa a la posición y orientación de un determinado UAV.

5.2 simulacion_navegacion_sincronizada.py

Este archivo también se aloja en la carpeta *src* del directorio *wanderbot*. Es el archivo fundamental de la simulación, dado que, al ejecutarse, inicia un nodo que se encarga de subir el UAV hasta la altura deseada y, una vez alcanzada dicha altura, se encarga de hacer que el UAV recorra la trayectoria circular que le corresponde. Además, de acuerdo al modelo expuesto en los capítulos 2 y 3, este nodo se encarga de realizar las maniobras de cambio de trayectorias entre vecinos siempre que sea necesario.

Como es inmediato pensar y, al igual que el archivo simulacion_subida_y_orientacion.py, el archivo simulacion_navegacion_sincronizada.py debe ejecutarse para cada UAV que compone el sistema a simular. Para lograr que dicha ejecución se produzca al mismo tiempo y que, por tanto, todos los UAVs viajen de forma sincronizada según el modelo expuesto, se hace uso nuevamente de archivos .xml como se comentará en la sección 5.4.

La estructura que sigue el código de este archivo es análoga al descrito en la sección anterior, diferenciándose, fundamentalmente, en los cálculos realizados dentro del bucle para obtener las velocidades que debe seguir el UAV, ya que el propósito de este nodo es el de llevar a cabo la navegación y los cambios de trayectoria que sean necesarios. Destaca, además, la definición de una función y la suscripción a un topic, *ausentes* (líneas 56-62), que permiten a este nodo conocer qué UAVs del sistema han abandonado. Esta información es usada dentro del bucle para saber si el UAV debe realizar la maniobra de cambio o no al llegar a una posición de cambio con algún vecino, calculando así unas velocidades u otras. El nodo que publica el topic *ausentes* será descrito en la siguiente sección.

Como aspectos a resaltar, cabe decirse que el nodo que inicia este archivo se ha denominado *simulacion_navegacion_sincronizada* y la publicación de las velocidades se realiza, como era de esperar, a través del mismo topic en el que publicaba el nodo de la sección anterior, dado que es el topic que indica al modelo de UAV tomado las velocidades que debe adquirir.

5.3 ausentes_publisher.py

Este archivo se encuentra también dentro de la carpeta *src* del directorio *wanderbot*. Es el encargado, al ejecutarse, de iniciar un nodo, *ausentes_publisher*, que informa permanentemente a todos los UAVs del sistema de qué componentes de este son los que han abandonado. Dicha información es usada por los nodos anteriores para realizar correctamente las maniobras de cambio entre vecinos. Es decir, los nodos encargados de controlar la navegación de los UAVs, descritos en la sección anterior, se suscriben a este nodo para saber qué UAVs han abandonado el sistema y realizar así los cambios de trayectoria, si aplica, con el vecino oportuno.

A diferencia de los dos archivos anteriores, los cuales debían ejecutarse para cada UAV que forma parte del equipo, este archivo sólo debe ejecutarse una vez para cada sistema de UAVs, dado que informa a todos los nodos que se suscriban a la información que publica.

En el código, destaca la importación inicial de mensajes tipo *Int32MultiArray* (línea 5), que será el tipo de mensajes publicado por este nodo. Se ha seleccionado este tipo de mensaje puesto que la información publicada por el nodo, esto es, los UAVs que han abandonado el sistema, se expresa fácilmente a través de un vector, lo cual se puede realizar de forma sencilla con el tipo de mensaje seleccionado. El topic en el que publica este nodo se ha denominado *ausentes*, al cual se suscriben los nodos de la sección anterior como ya se ha comentado. Finalmente, cabe resaltarse que dentro del bucle de este nodo, lo que se realiza de forma permanente es, simplemente, la actualización de la información proporcionada (UAVs que faltan) cada vez que se produce un intercambio de trayectorias.

5.4 simulacion_generador_sistema_uavs.py

Este archivo se aloja en la carpeta *launch* del directorio *wanderbot* y contiene el código básico que permite generar los archivos necesarios para la simulación de un sistema de UAVs cualquiera, requiriendo para ello el valor de diferentes parámetros como entrada así como del resto de archivos expuestos anteriormente. En la figura 5.2 se observa la entrada de algunos de dichos parámetros.

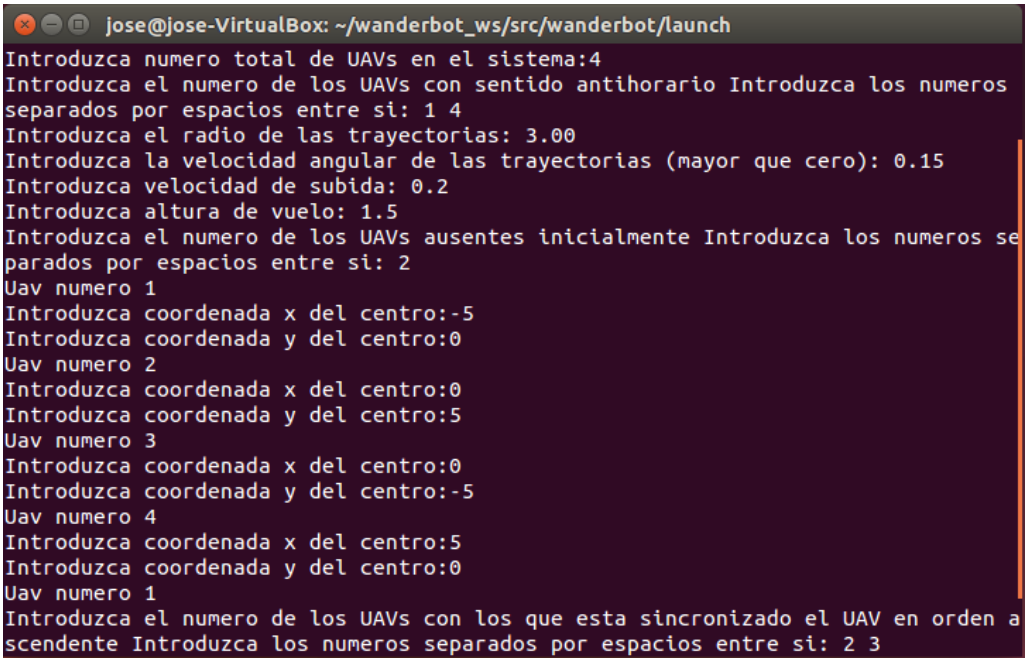


Figura 5.2 Introducción de parámetros requeridos al ejecutar el archivo simulacion_generador_sistema_-uavs.py.

El código comienza con la importación de diferentes librerías y funciones (líneas 1-6) que serán requeridas a lo largo de este, destacando la librería *lxml* la cual permitirá el tratamiento y generación de ficheros .xml. Tras ello, le siguen una serie de líneas que permiten la **entrada de parámetros y la caracterización del sistema de UAVs que se quiere simular** (7-204)(véase la figura 5.2). Destacan en este sentido las líneas (72-204) en las que, haciéndose uso del modelo expuesto en el capítulo 2, se establecen las posiciones de inicio de cada UAV del conjunto de forma que el sistema esté sincronizado, así como las diversas relaciones existentes entre los miembros del equipo: vecinos con los que está sincronizado cada UAV, puntos de cambio con dichos vecinos en los que habría que realizarse la maniobra de cambio de trayectorias si fuera necesaria, etc. En las líneas (206-216) se establecen una serie de parámetros en función de lo anterior que serán requeridos por los otros archivos. Las líneas restantes, (217-420), permiten la creación, en función de toda la información anterior, de cuatro archivos .xml, propios para cada sistema de UAVs:

1. spawn_quadrotors.xml
2. system_uavs_empty_world.xml
3. orientacion_prevuelo.xml

4. patrulla_circular.xml

Los archivos .xml son archivos que permiten, entre otras cosas, el lanzamiento de diferentes nodos de forma simultánea, por lo que su uso en los sistemas que se pretenden simular, en los que los UAVs deben volar de forma sincronizada, está más que justificado. A modo de ejemplo, en la figura 5.3 puede observarse como, tras ejecutar un archivo .xml (en particular, la imagen corresponde a la ejecución de un archivo patrulla_circular.xml) se inician al mismo tiempo cuatro nodos, tres correspondientes a la navegación de los UAVs que conforman el sistema (simulacion_navegacion_sincronizada_uav1 y análogos) y el nodo que informa de los UAVs que han abandonado el equipo (ausentes_publisher).

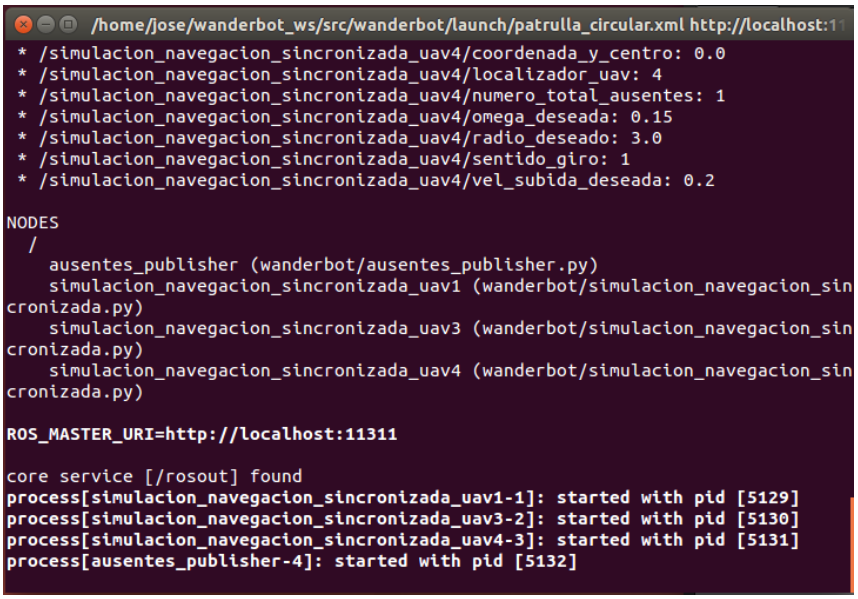


Figura 5.3 Imagen que muestra cómo, tras ejecutarse un archivo .xml, se inician de forma simultánea varios nodos.

spawn_quadrotors.xml Este archivo se encarga de generar los UAVs de los que consta el sistema que se quiere simular, haciendo uso del modelo tomado de quadrotor. Es por tanto un archivo auxiliar de la simulación.

system_uavs_empty_world.xml Este archivo se encarga de iniciar la simulación en Gazebo, poniendo en marcha el archivo anterior sobre el escenario que se le indique. En el caso del presente proyecto, el escenario elegido para todas las simulaciones es un escenario vacío que proporciona por defecto el simulador Gazebo.

orientacion_prevuelo.xml Este archivo pone en marcha, por cada UAV del sistema, el nodo encargado de realizar la subida inicial y la orientación previa a iniciar la navegación de cada componente. Es decir, a grosso modo, ejecuta el archivo simulacion_subida_y_orientacion.py de forma simultánea tantas veces como UAVs hay en el sistema.

patrulla_circular.xml Una vez los UAVs están orientados correctamente y listos para iniciar la navegación gracias al archivo anterior, se ejecuta este otro archivo, el cual pone en marcha simultáneamente por cada UAV del sistema el nodo que controla la navegación y los intercambios de cada uno (archivo simulacion_navegacion_sincronizada.py). Al mismo tiempo que inicia dichos nodos, inicia también el nodo que publica la información de los UAVs ausentes y que es requerida por estos como se ha comentado anteriormente. Un ejemplo de ejecución de este archivo puede verse en la figura 5.3.

A modo de ejemplo, en el apéndice B pueden encontrarse los cuatro archivos .xml de un sistema de UAVs concreto. Nótese que dichos archivos variarán de un sistema a otro en función de sus características.

Como se comentó anteriormente, cada nodo del sistema debe tener un nombre único para evitar cualquier tipo de conflicto en las comunicaciones. De acuerdo a esto, ¿por qué tras ejecutar un archivo .xml de los comentados, los cuales inician varios nodos del mismo tipo, no hay ningún problema? La respuesta está

en que, al generar los diferentes archivos .xml, si existe posibilidad de solapamiento en el nombre de dos o más nodos de los que lanzan, se renombran estos para que sean diferentes entre sí, añadiéndoles al nombre original del nodo el sufijo *_uav*$*$, donde $*$ representa el número del robot correspondiente. Por ejemplo, si se ejecuta el archivo patrulla_circular.xml para un sistema formado por tres robots, deben iniciarse, además del nodo de ausentes, tres nodos *simulacion_navegacion_sincronizada*, uno por cada UAV presente en el sistema. Para evitar el solapamiento que ello supondría, en lugar de iniciar los nodos con dicho nombre, los iniciaría llamándolos *simulacion_navegacion_sincronizada_uav1*, *simulacion_navegacion_sincronizada_uav2* y *simulacion_navegacion_sincronizada_uav3* respectivamente, con lo que quedaría totalmente resuelto dicho problema. Un ejemplo de esta situación puede observarse en la figura 5.3.

Además de solventar los problemas asociados a la nomenclatura de los diferentes nodos que lanzan, los archivos .xml también proporcionan a cada nodo que lanzan la información que necesitan. Por ejemplo, al lanzar un nodo simulacion_navegacion_sincronizada, para su correcto funcionamiento, este requiere conocer parámetros como la posición del centro de la trayectoria o el sentido del movimiento, por nombrar algunos. Dicha información es proporcionada por el archivo .xml correspondiente a la hora de iniciar el nodo en cuestión.

5.5 Procedimiento para llevar a cabo una simulación.

De acuerdo a todo lo anterior y, una vez alojados los cuatro archivos .py descritos en las secciones anteriores en las carpetas indicadas, los pasos a seguir para llevar a cabo una simulación de un sistema de UAVs cualquiera, sería como sigue:

1. Si es la primera vez que se van a utilizar los programas anteriores en un determinado ordenador, abrir una terminal de ROS y ejecutar las siguientes órdenes:
   - **cd ∼/wanderbot_ws**
   - **source devel/setup.bash**
   - **cd ∼/wanderbot_ws/src/wanderbot/src**
   - **chmod u+x ausentes_publisher.py**
   - **chmod u+x simulacion_navegacion_sincronizada.py**
   - **chmod u+x simulacion_subida_y_orientacion.py**

   donde *wanderbot_ws* es el nombre del *workspace* o directorio en el que se han guardado dichos programas y *wanderbot* la carpeta concreta dentro de dicho directorio en la que se encuentran. El nombre que se le dé tanto al directorio como a dicha carpeta es totalmente indiferente. De las líneas anteriores, las tres primeras son de uso habitual, dado que posibilitan para la terminal actual, el uso de una serie de órdenes y comandos básicos de ROS y sitúan al usuario en la carpeta deseada del directorio, en este caso, la carpeta *src*. Las líneas restantes, se encargan de hacer ejecutables desde ROS los archivos indicados.
2. Abrir un terminal y ejecutar **roscore**. Como se comentó en el capítulo 5, esta orden inicia la ejecución de un nodo maestro, el cual se encarga, entre otras funciones, de permitir la comunicación entre los diferentes nodos del sistema. Esta terminal se dejará por ello ejecutándose durante el transcurso de la simulación y no se hará más uso de ella. En la figura 5.4 se puede observar como queda dicha terminal una vez iniciado el nodo maestro.
3. Abrir una nueva terminal ROS, e introducir los comandos siguientes:
   - **cd ∼/wanderbot_ws**
   - **source devel/setup.bash**
   - **cd ∼/wanderbot_ws/src/wanderbot/launch**

   De la misma forma que antes, estas líneas sirven para permitir el uso de comandos básicos de ROS en dicha terminal y para situar al usuario en la carpeta deseada del directorio, en este caso la carpeta*launch*.
4. Introducir el siguiente comando: **python simulacion_generador_sistema_uavs.py**, lo cual inicia la ejecución de dicho archivo que, como se ha comentado anteriormente, **requerirá la introducción por parte del usuario de diversos parámetros que caractericen el sistema de UAVs** que se quiere

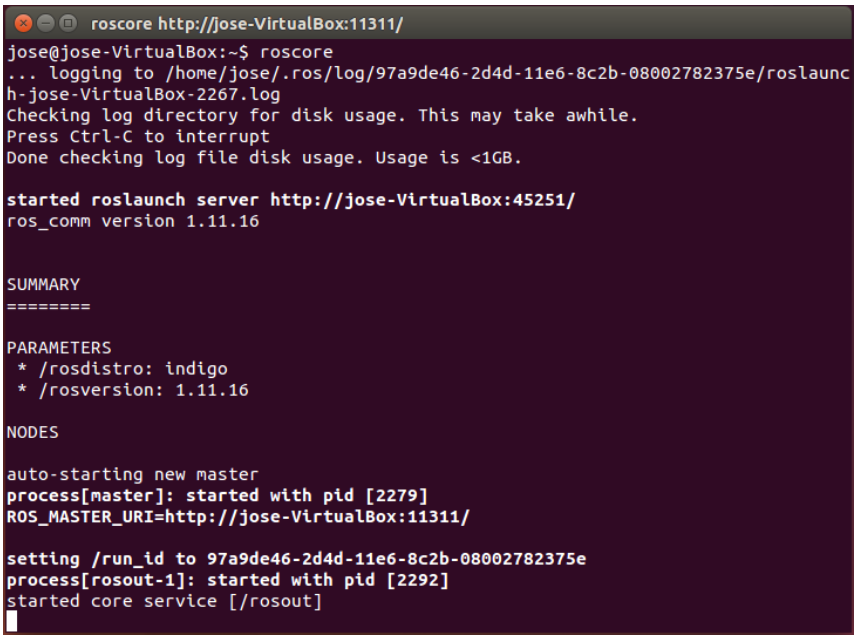


Figura 5.4 Terminal ejecutando el nodo maestro a partir de la orden roscore.

simular (véase la figura 5.2). Una vez finalizada la ejecución, se habrán generado los cuatro archivos .xml necesarios para llevar a cabo la simulación del sistema deseado.

5. Ejecutar el comando **roslaunch wanderbot system_uavs_empty_world.xml** para ejecutar el archivo .xml correspondiente, el cual lanzará la simulación del sistema en cuestión en el escenario pertinente. Tras esta orden deberá iniciarse el simulador Gazebo mostrando el equipo de UAVs de acuerdo a las características deseadas. En la figura 4.3 se observa una captura de dicha pantalla para un sistema concreto.
6. Abrir un nuevo terminal ROS, y ejecutar nuevamente las líneas que permiten el uso de comandos básicos y la situación en la carpeta requerida, esto es:
   - **cd ∼/wanderbot_ws**
   - **source devel/setup.bash**
   - **cd ∼/wanderbot_ws/src/wanderbot/launch**
7. Hecho esto, ejecutar en esta misma terminal: **roslaunch wanderbot orientacion_prevuelo.xml**, lo cual, como cabe esperar, provoca la subida y la orientación inicial de los UAVs de acuerdo a lo expuesto en las secciones anteriores. Una vez que se detengan todos los UAVs, esto es, dejen de girar al haber completado su orientación previa a la navegación, introducir **Ctrl+C** para detener la ejecución de este archivo.
8. Sin cambiar de terminal, introducir **roslaunch wanderbot patrulla_circular.xml** para ejecutar ahora el archivo que inicia la navegación en sí de los UAVs del sistema y que implementa, además, las estrategias de cambio vistas en capítulos anteriores. Con ello se habrá finalizado el lanzamiento de la simulación en cuestión, debiendo introducirse nuevamente **Ctrl+C** cuando se quiera detener.

Nótese que se han supuesto instalados tanto ROS como los paquetes relativos a Gazebo y al modelo de UAV tomado, descritos a lo largo de los capítulos anteriores.

6 Análisis.

En el presente capítulo se desarrolla el análisis de las dos configuraciones de UAVs comentadas anteriormente: árbol y malla, tanto desde el punto de vista de la isolation-resilience como desde el punto de vista de la uncovering-resilience, medidas ambas, de la robustez del sistema. Para ello y, haciéndose uso de los códigos desarrollados, se llevan a cabo las simulaciones pertinentes. Una vez realizado el análisis de ambos tipos de conjuntos, se expone una somera comparación entre ambos en la sección 6.3.

6.1 Sistema de UAVs en forma de árbol.

Se trata en esta sección de analizar un sistema de UAVs cuyo grafo de comunicación, $G = (V,E)$, presenta forma de árbol. La configuración particular seleccionada puede observarse en la figura 6.1, así como la numeración seguida. Como se puede ver, el sistema consta inicialmente de cuatro componentes, siendo por tanto $n = 4$.

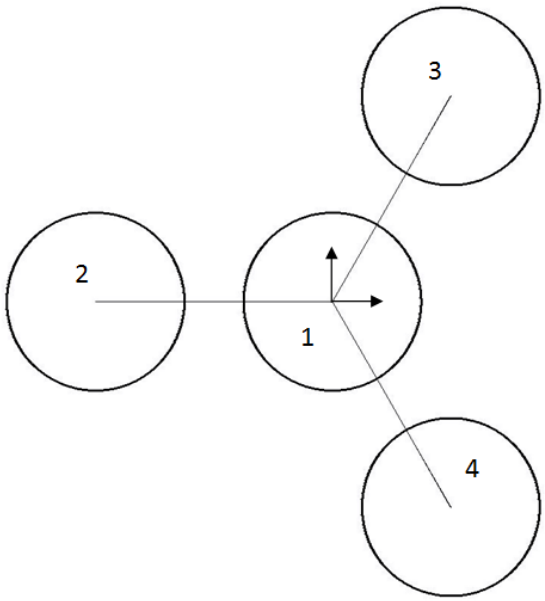


Figura 6.1 Sistema de UAVs analizado para el caso de grafo con forma de árbol y numeración seguida.

Uncovering-resilience. De acuerdo a lo expuesto en el capítulo 3, en particular, la expresión 3.1, se tiene que la U_R del sistema considerado es igual a 3. Es decir, pueden eliminarse hasta tres UAVs del sistema sin que ninguna trayectoria de este quede no cubierta. Puesto que el sistema inicial considerado consta de cuatro UAVs en total, todas las trayectorias estarán cubiertas abandonen los robots que abandonen, a excepción, claro está, de que abandonen todos los componentes del equipo. Para ilustrar lo anterior,

se simula el caso más crítico, esto es, que abandonen el equipo tres de los cuatro robots, debiendo quedar cubierto todo el territorio según lo expuesto. Los parámetros introducidos con el fin de llevar a cabo dicha simulación se recogen en la tabla 6.1 y el resultado puede verse en el siguiente vídeo:

1. **Simulación**: *https://www.youtube.com/watch?v=FKnVgL27GDw*

Tabla 6.1 Parámetros para la simulación del sistema de UAVs cuyo grafo de comunicación tiene forma de árbol y en el que han abandonado todos los robots del sistema salvo el número 1.

Parámetro	Valor
N° total de UAVs	4
N° de los UAVs con sentido de vuelo antihorario	1
Radio de las trayectorias	3 m
Velocidad angular	0.15 rad/s
Velocidad de subida	0.15 m/s
Altura de vuelo	1.2 m
N° de los UAVs ausentes	2,3,4
Coordenada x del centro de la trayectoria del UAV n° 1	0
Coordenada y del centro de la trayectoria del UAV n° 1	0
Coordenada x del centro de la trayectoria del UAV n° 2	-7.071
Coordenada y del centro de la trayectoria del UAV n° 2	0
Coordenada x del centro de la trayectoria del UAV n° 3	3.536
Coordenada y del centro de la trayectoria del UAV n° 3	6.124
Coordenada x del centro de la trayectoria del UAV n° 4	3.536
Coordenada y del centro de la trayectoria del UAV n° 4	-6.124
N° de los UAVs con los que está sincronizado el UAV n° 1	2,3,4
N° de los UAVs con los que está sincronizado el UAV n° 2	1
N° de los UAVs con los que está sincronizado el UAV n° 3	1
N° de los UAVs con los que está sincronizado el UAV n° 4	1

Nótese que este comportamiento del sistema es independiente, entre otras cosas, del número de los UAVs que abandonen (en este caso han abandonado el sistema los números 2,3 y 4).

Isolation-resilience. Considerando ahora la resiliencia asociada a incomunicación se tiene, de acuerdo a la expresión 3.3, que $I_R \geq 1$. Es decir, es posible asegurar que si sólo abandona el equipo uno de los cuatro componentes (sea cual sea), el sistema no entrará en estado de incomunicacion. Nótese que, debido a la mayor complejidad que conllevan este tipo de sistemas, no ha sido posible obtener una expresión que permita calcular la I_R del sistema, sino solamente una cota inferior para esta. Con objeto de visualizar la validez de la expresión anterior, se simula dicho sistema considerando que abandona uno de los robots (se ha considerado que abandona el número 3, siendo indiferente que abandone el sistema un robot u otro, de los cuatro que conforman el equipo como ya se ha expuesto). Los datos de la simulación llevada a cabo son los mismos que los recogidos en la tabla 6.1 salvo que, en este caso, sólo está ausente el UAV número 3, como ya se ha dicho. Así, los resultados de la simulación pueden observarse en el siguiente enlace:

2. **Simulación**: *https://www.youtube.com/watch?v=NoPojLfWUcg*

Como era de esperar, no se produce incomunicación en el sistema eliminando un solo UAV de los cuatro iniciales. Simulando ahora el caso en que abandonan dos UAVS, por ejemplo los número 1 y 4, se tiene:

3. **Simulación**: *https://www.youtube.com/watch?v=go2m2UtQoRQ*

De acuerdo al vídeo anterior puede verse que, para esta configuración particular, el sistema entra en estado de incomunicación al eliminar dos de los cuatro componentes iniciales (UAVs número 1 y 4), por lo que puede asegurarse que $I_R = 1$. El hecho de que $I_R = 1$ no implica que, quitando dos UAVs cualesquiera del sistema este entre en incomunicación, sino que existe, al menos, un par de robots tales que al abandonar provocan el estado incomunicación del sistema.

Nótese además que, el comportamiento observado en el vídeo, no podía ser asegurado a priori, dado que el valor de I_R dependerá del árbol que caracterice el sistema. Adicionalmente, resulta evidente que, si se eliminan tres robots del sistema considerado, queda un único UAV navegando que, necesariamente, debe estar incomunicado y con él el sistema, puesto que no queda en tal configuración ningún robot con que encontrarse.

Cabe destacarse que, como se dijo en el capítulo 3, el hecho de que el sistema entre en estado de **incomunicación no implica, necesariamente, la aparición** en este **de trayectorias no cubiertas**. Sin ir más lejos, para los casos simulados anteriormente, el sistema entra en incomunicación al abandonar dos UAVs, mientras que, aún abandonando tres robots, todas las trayectorias siguen cubiertas.

6.2 Sistema de UAVs en forma de malla.

En esta sección, se analiza un sistema de UAVs cuyo grafo de comunicación, $G = (V,E)$, tiene forma de cuadrícula o malla, entendiéndose por ello que presenta una configuración acorde a lo comentado en la sección 3.2. En particular, se analiza la configuración representada en la figura 6.2 cuyo sistema, inicialmente, está formado por cuatro robots, estando caracterizado por $n = m = 2$, donde n representa las filas de la cuadrícula y m las columnas. Nótese la numeración asignada a cada UAV.

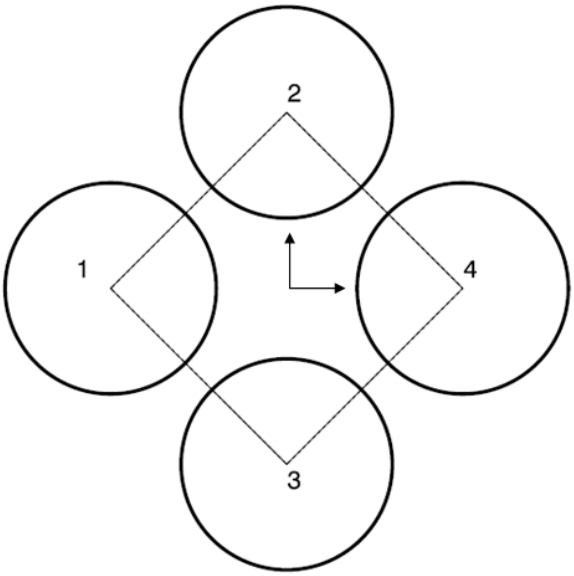


Figura 6.2 Sistema de UAVs analizado para el caso de grafo con forma de malla y numeración seguida.

Uncovering-resilience. En lo que respecta a la uncovering-resilience, sin más que hacer uso de la expresión 3.2, se tiene que el máximo número de UAVs que pueden abandonar el sistema de forma que se asegure que no aparecen trayectorias no cubiertas es uno. En otras palabras, eliminando dos o tres de los componentes del equipo, puede quedar algún tramo del recorrido total de los robots sin cubrir por ninguno de ellos. Para ilustrar dicho comportamiento y, haciendo uso del código desarrollado para ello, se procede a simular ambas situaciones.

En primer lugar, para ilustrar que no existe ninguna trayectoria no cubierta eliminando un componente del sistema, se desarrolla la simulación introduciendo los datos mostrados en la tabla 6.2 (el procedimiento para llevar a cabo una simulación fue descrito en la sección 5.5) y cuyo resultado puede observarse en el siguiente enlace:

4. **Simulación**: *https://www.youtube.com/watch?v=_gD7akuEKb0*

Nótese que el comportamiento del sistema es independiente de cual de los cuatro UAVs iniciales se haya eliminado. En este caso en particular, se ha eliminado el UAV número 2.

Tabla 6.2 Parámetros para la simulación del sistema de UAVs cuyo grafo de comunicación tiene forma de malla y en el que ha abandonado el robot número 2.

Parámetro	Valor
N° total de UAVs	4
N° de los UAVs con sentido de vuelo antihorario	1,4
Radio de las trayectorias	3 m
Velocidad angular	0.15 rad/s
Velocidad de subida	0.15 m/s
Altura de vuelo	1.2 m
N° de los UAVs ausentes	2
Coordenada x del centro de la trayectoria del UAV n° 1	-5
Coordenada y del centro de la trayectoria del UAV n° 1	0
Coordenada x del centro de la trayectoria del UAV n° 2	0
Coordenada y del centro de la trayectoria del UAV n° 2	5
Coordenada x del centro de la trayectoria del UAV n° 3	0
Coordenada y del centro de la trayectoria del UAV n° 3	-5
Coordenada x del centro de la trayectoria del UAV n° 4	5
Coordenada y del centro de la trayectoria del UAV n° 4	0
N° de los UAVs con los que está sincronizado el UAV n° 1	2,3
N° de los UAVs con los que está sincronizado el UAV n° 2	1,4
N° de los UAVs con los que está sincronizado el UAV n° 3	1,4
N° de los UAVs con los que está sincronizado el UAV n° 4	2,3

Por otra parte, eliminando dos componentes del sistema, se tiene que pueden aparecer trayectorias no cubiertas, lo cual puede verse eliminando, por ejemplo, los UAVs número 2 y 3 para este caso particular. Los datos correspondientes a esta simulación son iguales que los de la tabla 6.2 a excepción del número de los UAVs ausentes, que en este caso pasa a ser (2,3), observándose el comportamiento del sistema en el siguiente vídeo:

5. **Simulación**: *https://www.youtube.com/watch?v=e8kmikBmZds*

La situación si abandonasen tres componentes del sistema sería análoga.

Isolation-resilience. En lo que a isolation-resilience se refiere, se tiene que, haciéndose uso de la expresión 3.4, el máximo número de UAVs que pueden abandonar el sistema de forma que se asegure que este no entre en estado de incomunicación es de uno. Es decir, si abandonan el equipo dos robots puede producirse la incomunicación del conjunto.

La situación en la que se elimina un UAV y no hay incomunicación (así como tampoco aparecen trayectorias no cubiertas, de acuerdo a lo expuesto anteriormente para este mismo sistema) puede comprobarse en la 4° simulación, cuyos parámetros se recogen, como se dijo, en la tabla 6.2. El UAV que abandone el sistema es, nuevamente, indiferente.

Por contra, si el sistema lo abandonan dos UAVs, puede producirse la incomunicación, observándose un ejemplo de dicha situación en la 5° simulación, en la que también aparecen trayectorias no cubiertas como se ha comentado anteriormente para este mismo conjunto y donde se han eliminado nuevamente los UAVs números 2 y 3. Resulta evidente que, si se eliminan tres robots del sistema considerado, queda un único UAV navegando que, necesariamente, debe estar incomunicado y con él el sistema, puesto que no queda en tal configuración ningún robot con que encontrarse.

6.3 Comparación de ambas configuraciones.

Como análisis final de los sistemas considerados, se va a realizar una comparativa entre ambas configuraciones: árbol y malla.

En primer lugar, cabe destacarse que ambas se han considerado inicialmente formadas por 4 UAVs, por lo que la elección de una u otra configuración podría responder a cuestiones de cobertura de los diferentes

equipos embarcados en los robots, por la mayor o menor adecuación de una u otra configuración al terreno que se quiera explorar (o cualquier otra tarea a la que sea destinado el conjunto), etc, dado que los costes en infraestructuras serían prácticamente los mismos en ambas (mismo número de UAVs).

Un criterio adicional a los anteriores para la elección podría ser la robustez de una configuración u otra, aspecto este analizado en las dos secciones anteriores. Tanto la uncovering-resilience como la isolation-resilience son fenómenos totalmente indeseables, puesto que reducen el rendimiento global del conjunto.

De esta forma, en lo que a U_R se refiere, se ha visto que el sistema en forma de árbol es totalmente robusto desde este punto de vista, puesto que todas las trayectorias permanecen cubiertas aún abandonando 3 robots el sistema, no ocurre así en el sistema en forma de *grid*, en el que cuando abandonan dos (o más) UAVs el equipo aparecen tramos no visitados por estos. En diversas misiones, como por ejemplo de monitorización de una determinada zona de conflicto, este efecto podría tener consecuencias devastadoras en la consecución de los objetivos fijados, por lo que parece aconsejable optar por el sistema cuyo grafo de comunicación es en forma de árbol en estos casos.

Por otra parte, desde el punto de vista de la isolation-resilience, se ha comprobado que tanto el sistema en forma de malla como el sistema en forma de árbol pueden entrar en estado de incomunicación (esto es, quedan totalmente aislados los UAVs que permanecen en el sistema) al eliminar el mismo número de UAVs: dos, por lo que, de acuerdo a este criterio, no puede decirse a priori que una configuración posea una ventaja sobre la otra. En otras palabras, la tolerancia a fallos de ambos sistemas en relación a la incomunicación es la misma. El problema de la incomunicación puede resultar fundamentalmente crítico en misiones, por ejemplo, en las que el transporte de información de una parte del sistema a otra sea especialmente importante.

7 Conclusiones y trabajos futuros.

En este proyecto se ha presentado un modelo simplificado para la sincronización de un equipo de UAVs que trabajan de forma coordinada, así como la estrategia a seguir para mantener dicha sincronización en el caso en que uno o varios de los componentes del equipo se vean obligados a abandonar sus tareas, por ejemplo, para recargar baterías. Con ello, se busca mantener el rendimiento global del equipo en la medida en que sea posible. Una forma de cuantificar cómo de robusto es un determinado sistema a la pérdida de componentes es a través del concepto presentado de *resiliencia*, más concretamente, se ha definido la *uncovering-resilience* y la *isolation-resilience*, asociadas a la existencia en el sistema de trayectorias no cubiertas y al fenómeno de la incomunicación, respectivamente. El fenómeno de la incomunicación puede aparecer en sistemas sincronizados en el que han abandonado alguno de sus miembros y se caracteriza por una pérdida de comunicación permanente de uno o más miembros del equipo. Así, la resiliencia de un sistema es el máximo número de robots que pueden abandonar el equipo sin que deje de haber una cobertura total de todas las trayectorias, o bien sin que aparezca el fenómeno de incomunicación.

Haciéndose uso tanto del modelo expuesto como de los resultados desarrollados en [10] relativos a resiliencia de diferentes sistemas, se han analizado diferentes configuraciones de UAVs, implementando además una serie de códigos que permiten la simulación de dichos sistemas. Estas simulaciones se han llevado a cabo en ROS gracias al simulador Gazebo, por ser este el software más destacado y utilizado en los últimos años en lo que a robótica se refiere. Entre sus muchas ventajas, destaca la cercanía con la experimentación real que permite conseguir.

Como punto final al trabajo desarrollado, se proponen a continuación posibles líneas futuras de investigación orientadas, sobre todo, al desarrollo de situaciones más realistas que las aquí expuestas:

1. Empleo de un modelo más próximo a la realidad. Ejemplos de ello podrían ser:
    - Considerar el sistema formado por diferentes tipos de UAVs o, incluso, sistemas formados por vehículos aéreos y terrestres, con diferentes rangos de comunicación para cada uno de ellos.
    - Considerar diferentes alturas para cada UAV del equipo.
    - Contemplar la posibilidad de que, aun sin abandonar el sistema, un robot pueda ser incapaz de comunicarse con el resto y llevar a cabo sus tareas. Esto podría implicar, por ejemplo, la necesidad de implementar en las simulaciones de ROS un determinado sensor de proximidad para evitar colisiones.
2. Considerar efectos externos como el efecto del viento, de especial importancia en los drones debido a que suelen ser bastante ligeros.
3. Desarrollar las simulaciones en escenarios más realistas. Es decir, en las simulaciones llevadas a cabo en el presente trabajo, siempre se ha hecho uso de un escenario vacío que incluye Gazebo por defecto. Una posible mejora sería por tanto modelar en Gazebo un escenario concreto, por ejemplo, el interior de una nave industrial, y desarrollar las simulaciones de acuerdo a dicho escenario. Un ejemplo de escenario más realista puede verse en la figura **??**, donde se observa un terreno montañoso. Nótese que a la derecha de la foto pueden observarse las imágenes que capta la cámara que lleva incorporada el UAV así como datos tomados por sus sensores.
4. Desarrollar un modelo propio de UAV, acorde al robot del que se disponga, en lugar de hacer uso de uno ya existente, como aquí ha sido el caso.

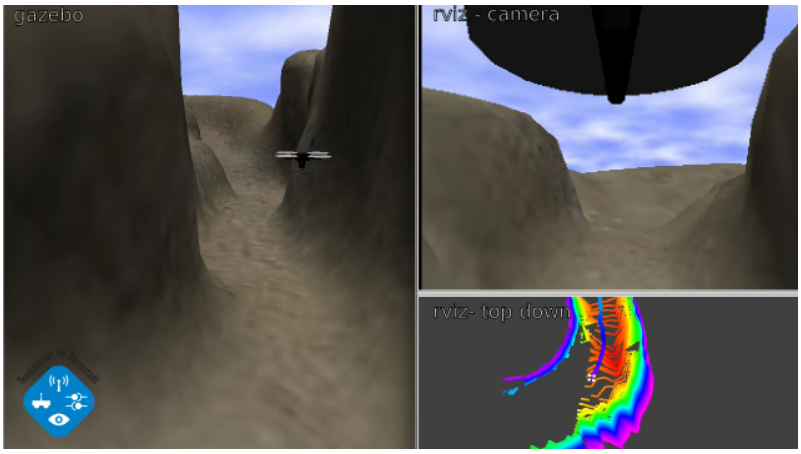


Figura 7.1 Desarrollo de una simulación en Gazebo donde el escenario representa un terreno montañoso. A la derecha pueden observarse las imágenes captadas por la cámara incorporada por el UAV así como datos procedentes de sus diferentes sensores. Imagen tomada de *https://www.youtube.com/watch?v=9CGIcc0jeuI*.

5. Orientar la simulación al análisis de las medidas, datos e imágenes captadas por los equipo de a bordo del UAV (los cuales pueden ser modelados libremente), comprobando así si la cobertura instrumental proporcionada por el equipo es la deseada, etc.
6. Extender el análisis aquí desarrollado a otras configuraciones particulares de sistemas o, incluso, a otro tipo de sistemas que no tengan forma ni de árbol ni de cuadrícula. Un ejemplo de esto último serían sistemas cuyo grafo de comunicación tiene forma de *ciclo*. La resiliencia de este tipo de sistemas también es objeto de estudio en [10]. Otra posibilidad sería el caso de combinaciones de estos tipos de sistemas.

Apéndice A
Código ROS.

Código A.1 simulacion_subida_y_orientacion.py.

```
1
2
3  #!/usr/bin/env python
4  import rospy
5  from geometry_msgs.msg import Twist
6  from nav_msgs.msg import Odometry
7  from math import sqrt, acos, cos, sin, atan, pi
8
9  omega=0.15
10 vel_subida=0.15
11 altura_especificada=0.5
12
13 rospy.init_node('simulacion_subida_y_orientacion')
14
15 def odometry_callback(msg):
16   global altura
17   global coordenada_x
18   global coordenada_y
19   global orientacion
20   altura=msg.pose.pose.position.z
21   coordenada_x=msg.pose.pose.position.x
22   coordenada_y=msg.pose.pose.position.y
23   orientacion=msg.pose.pose.orientation.w
24
25 altura=0 #Valor para inicializar.
26 coordenada_x=0
27 coordenada_y=0
28 orientacion=0
29
30 posicion_sub=rospy.Subscriber('ground_truth/state', Odometry, odometry_callback
31     )
32
33 cmd_vel_pub = rospy.Publisher('cmd_vel', Twist, queue_size=1)
34 rate = rospy.Rate(100000)
35
36 xc = rospy.get_param('~coordenada_x_centro')
37 yc = rospy.get_param('~coordenada_y_centro')
38 sentido = rospy.get_param('~sentido_giro')
39
```

```
while not rospy.is_shutdown():
  simulacion_subida_y_orientacion=Twist()
  simulacion_subida_y_orientacion.linear.z=vel_subida
  if abs(altura-altura_especificada)<=0.1: #Tolerancia inicial de 0.1
    simulacion_subida_y_orientacion.linear.z=0.0
    alpha=2*acos(orientacion)
    simulacion_subida_y_orientacion.angular.z=0.0
    if (coordenada_y-yc)==0:
      if (coordenada_x-xc)>=0:
        angulo=0
      if (coordenada_x-xc)<0:
        angulo=pi
    elif (coordenada_x-xc)==0:
      if (coordenada_y-yc)>=0:
        angulo=pi/2
      if (coordenada_y-yc)<0:
        angulo=3*pi/2
    else:
      incremento=(coordenada_y-yc)/(coordenada_x-xc)
      angulo=atan(incremento)
      if (coordenada_y-yc)/abs(coordenada_y-yc)>=0 and (coordenada_x-xc)/abs(
          coordenada_x-xc)<=0:
        angulo=angulo+pi
      if (coordenada_y-yc)/abs(coordenada_y-yc)<=0 and (coordenada_x-xc)/abs(
          coordenada_x-xc)<=0:
        angulo=angulo+pi
      if angulo<=0:
        angulo=angulo+2*pi
    if sentido==-1:
      angulo=angulo-pi
      if angulo<0:
        angulo=angulo+2*pi
    if (abs(angulo-alpha)>=0.05) and (sentido==1):
      simulacion_subida_y_orientacion.angular.z=omega
    elif (abs(2*pi-angulo-alpha)>=0.05) and (sentido==-1):
      simulacion_subida_y_orientacion.angular.z=-omega
  cmd_vel_pub.publish(simulacion_subida_y_orientacion)
  rate.sleep()
```

Código A.2 simulacion_navegacion_sincronizada.py.

```

#!/usr/bin/env python
################################################################################
import rospy
from geometry_msgs.msg import Twist
from nav_msgs.msg import Odometry
from std_msgs.msg import Int32MultiArray
from math import sqrt, acos, cos, asin, sin, atan, pi
################################################################################
rospy.init_node('simulacion_navegacion_sincronizada')

def odometry_callback(msg):
  global altura
  global coordenada_x
  global coordenada_y
```

```
  global orientacion
  altura=msg.pose.pose.position.z
  coordenada_x=msg.pose.pose.position.x
  coordenada_y=msg.pose.pose.position.y
  orientacion=msg.pose.pose.orientation.w
#Valores para inicializar.
altura=0
coordenada_x=0
coordenada_y=0
orientacion=0
#############################################################################
posicion_sub=rospy.Subscriber('ground_truth/state', Odometry, odometry_callback
    )

cmd_vel_pub = rospy.Publisher('cmd_vel', Twist, queue_size=1)

rate = rospy.Rate(1000000)
vueltas=2
#############################################################################
localizador = rospy.get_param('~localizador_uav')
dic_centros = rospy.get_param('diccionario_centros')
dic_emparejados = rospy.get_param('diccionario_emparejados')
dic_pos_cambio = rospy.get_param('diccionario_posiciones_cambio')
dic_dis_centros = rospy.get_param('diccionario_distancia_centros')
dic_betas = rospy.get_param('diccionario_betas')
r = rospy.get_param('~numero_total_ausentes')

radio = rospy.get_param('~radio_deseado')
omega = rospy.get_param('~omega_deseada')
vel_subida = rospy.get_param('~vel_subida_deseada')
altura_especificada = rospy.get_param('~altura_deseada')

xc = rospy.get_param('~coordenada_x_centro')
yc = rospy.get_param('~coordenada_y_centro')

sentido = rospy.get_param('~sentido_giro')
#Si el sentido de giro es horario, pasar como parametro sentido=-1. Si es
    antihorario, sentido=1.
#############################################################################
def ausencias_callback(msg):
  global ausencias
  ausencias=msg.data

ausencias=[]

ausencias_sub=rospy.Subscriber('ausentes', Int32MultiArray, ausencias_callback)
#############################################################################
cambio=False
maniobras=False
maniobra_insercion=False

vel_cambio=1.15*omega*radio
sustitucion=[]
#############################################################################
while not rospy.is_shutdown():
  simulacion_navegacion_sincronizada=Twist()
  simulacion_navegacion_sincronizada.linear.z=vel_subida
```

```
  if abs(altura-altura_especificada)<=0.1: #Tolerancia inicial de 0.1
    simulacion_navegacion_sincronizada.linear.z=0.0
################################################################################

    distancia=sqrt((coordenada_x-xc)**2+(coordenada_y-yc)**2)
    alpha=2*acos(orientacion)
    if sentido==1:
      vx=-omega*radio*(coordenada_y-yc)/distancia
      vy=omega*radio*(coordenada_x-xc)/distancia
      if vueltas%2==1:
        alpha=2*pi-alpha
      if abs(alpha-359.147583699*pi/180)<=0.1:
        vueltas=vueltas+1
      simulacion_navegacion_sincronizada.linear.x=vx*cos(alpha)+vy*sin(alpha)
      if (distancia-radio)>=0.05:
        simulacion_navegacion_sincronizada.linear.x=vx*cos(alpha)+vy*sin(alpha)
            -2*(distancia-radio)
      if (radio-distancia)>0.05:
        simulacion_navegacion_sincronizada.linear.x=vx*cos(alpha)+vy*sin(alpha)
            +2*(distancia-radio)
      simulacion_navegacion_sincronizada.linear.y=-vx*sin(alpha)+vy*cos(alpha)
      simulacion_navegacion_sincronizada.angular.z=omega
    if sentido==-1:
      vx=omega*radio*(coordenada_y-yc)/distancia
      vy=-omega*radio*(coordenada_x-xc)/distancia
      if vueltas%2==1:
        alpha=2*pi-alpha
      if abs(alpha-359.147583699*pi/180)<=0.1:
        vueltas=vueltas+1
      alpha=alpha-pi
      if alpha<0:
        alpha=alpha+2*pi
      simulacion_navegacion_sincronizada.linear.x=-(vx*cos(alpha)-vy*sin(alpha)
          )
      if (distancia-radio)>=0.05:
        simulacion_navegacion_sincronizada.linear.x=-(vx*cos(alpha)-vy*sin(
            alpha)-2*(distancia-radio))
      if (radio-distancia)>0.05:
        simulacion_navegacion_sincronizada.linear.x=-(vx*cos(alpha)-vy*sin(
            alpha)+2*(distancia-radio))
      simulacion_navegacion_sincronizada.linear.y=-(vx*sin(alpha)+vy*cos(alpha)
          )
      simulacion_navegacion_sincronizada.angular.z=-omega
################################################################################

    n="uav%d" %(localizador)
    emparejados=dic_emparejados[n]
    sustitucion=[]
    for a in range(0,len(emparejados)):
      sustitucion.append(0)
    if maniobras==False:
      for j in range(0,len(emparejados)):
        for x in ausencias:
          if x==emparejados[j]:
            sustitucion[j]=-1
            break
      decision=False
```

```
      cambio=False
      for p in range(0,len(sustitucion)):
        if sustitucion[p]==-1 and decision==True:
          beta_decision=dic_betas[n][p]
          if sentido==1:
            alpha_decision=alpha
          elif sentido==-1:
            alpha_decision=2*pi-alpha

          if sentido==1:
            incremento1=beta_decidido-alpha_decision
            if incremento1<0:
              incremento1=2*pi-incremento1
            incremento2=beta_decision-alpha_decision
            if incremento2<0:
              incremento2=2*pi-incremento2
            if incremento2<=incremento1:
              sustituir=p
              nn="uav%d" %(emparejados[p])
              beta_decidido=beta_decision
          elif sentido==-1:
            incremento1=alpha_decision-beta_decidido
            if incremento1<0:
              incremento1=2*pi-incremento1
            incremento2=alpha_decision-beta_decision
            if incremento2<0:
              incremento2=2*pi-incremento2
            if incremento2<=incremento1:
              sustituir=p
              nn="uav%d" %(emparejados[p])
              beta_decidido=beta_decision
        if sustitucion[p]==-1 and decision==False:
          sustituir=p
          nn="uav%d" %(emparejados[p])
          beta_decidido=dic_betas[n][p]
          decision=True
          cambio=True
##############################################################################

    if cambio==True:
      argumento=radio/(dic_dis_centros[n][sustituir]/2)
      theta=acos(argumento)
      if sentido==1:
        angulo_cambio=dic_betas[n][sustituir]-theta
        angulo_reinsercion=dic_betas[n][sustituir]+pi-theta
        if angulo_cambio<0:
          angulo_cambio=angulo_cambio+2*pi
        if angulo_reinsercion<0:
          angulo_reinsercion=angulo_reinsercion+2*pi
      elif sentido==-1:
        angulo_cambio=dic_betas[n][sustituir]+theta
        angulo_reinsercion=dic_betas[n][sustituir]-pi+theta
        if angulo_cambio>2*pi:
          angulo_cambio=angulo_cambio-2*pi
        if angulo_reinsercion<0:
          angulo_reinsercion=angulo_reinsercion+2*pi
      x_cambio=xc+radio*cos(angulo_cambio)
```

```
      y_cambio=yc+radio*sin(angulo_cambio)
      x_reinsercion=dic_centros[nn][0]+radio*cos(angulo_reinsercion)
      y_reinsercion=dic_centros[nn][1]+radio*sin(angulo_reinsercion)
      if abs(coordenada_x-x_cambio)<=0.05 and abs(coordenada_y-y_cambio)<=0.05:
        maniobras=True
        maniobra_insercion=True
        cambio==False
        actualizacion_ausentes=rospy.set_param('cambios_ausentes', [localizador,
            emparejados[sustituir]])

##############################################################################

    if maniobra_insercion==True:
      simulacion_navegacion_sincronizada.linear.x=0
      simulacion_navegacion_sincronizada.linear.y=0
      simulacion_navegacion_sincronizada.angular.z=0.0
      if (y_reinsercion-coordenada_y)==0:
        if (x_reinsercion-coordenada_x)>=0:
          xi=0
        if (x_reinsercion-coordenada_x)<0:
          xi=pi
      elif (x_reinsercion-coordenada_x)==0:
        if (y_reinsercion-coordenada_y)>=0:
          xi=pi/2
        if (y_reinsercion-coordenada_y)<0:
          xi=3*pi/2
      else:
        incremento=(y_reinsercion-coordenada_y)/(x_reinsercion-coordenada_x)
        xi=atan(incremento)
        if (y_reinsercion-coordenada_y)/abs(y_reinsercion-coordenada_y)>=0 and
            (x_reinsercion-coordenada_x)/abs(x_reinsercion-coordenada_x)<=0:
          xi=xi+pi
        if (y_reinsercion-coordenada_y)/abs(y_reinsercion-coordenada_y)<=0 and
            (x_reinsercion-coordenada_x)/abs(x_reinsercion-coordenada_x)<=0:
          xi=xi+pi
        if xi<=0:
          xi=xi+2*pi
##############################################################################
      if sentido==1:
        simulacion_navegacion_sincronizada.linear.x=vel_cambio*cos(xi-alpha)
        simulacion_navegacion_sincronizada.linear.y=vel_cambio*sin(xi-alpha)
      elif sentido==-1:
        simulacion_navegacion_sincronizada.linear.x=vel_cambio*cos(xi-(pi-alpha)
            )
        simulacion_navegacion_sincronizada.linear.y=vel_cambio*sin(xi-(pi-alpha)
            )
##############################################################################
      if sentido==1:
        if (angulo_cambio-alpha)>=0.05:
          simulacion_navegacion_sincronizada.angular.z=omega/1.3
        if (alpha-angulo_cambio)>0.05:
          simulacion_navegacion_sincronizada.angular.z=-omega/1.3
      elif sentido==-1:
        if (angulo_cambio-(2*pi-alpha))>=0.05:
          simulacion_navegacion_sincronizada.angular.z=omega/1.3
        if ((2*pi-alpha)-angulo_cambio)>0.05:
          simulacion_navegacion_sincronizada.angular.z=-omega/1.3
```

```
################################################################################
      if abs(coordenada_x-x_reinsercion)<=0.05 and abs(coordenada_y-
          y_reinsercion)<=0.05:
        sentido=-sentido
        localizador=emparejados[sustituir]
        xc=dic_centros[nn][0]
        yc=dic_centros[nn][1]
        cambio=False
        maniobras=False
        maniobra_insercion=False
################################################################################
        if vueltas%2==1:
          vueltas=2
        elif vueltas%2==0:
          vueltas=3
################################################################################
#       parada=True
#   if parada==True:
#     simulacion_navegacion_sincronizada.linear.x=0
#     simulacion_navegacion_sincronizada.linear.y=0
#     simulacion_navegacion_sincronizada.angular.z=0
################################################################################
  cmd_vel_pub.publish(simulacion_navegacion_sincronizada)
  rate.sleep()
################################################################################
```

Código A.3 ausentes_publisher.py.

```

#!/usr/bin/env python
import rospy
from std_msgs.msg import Int32MultiArray

rospy.init_node('ausentes_publisher')

pub = rospy.Publisher('ausentes', Int32MultiArray, queue_size=1)

rate = rospy.Rate(1000000)

ausentes_iniciales=[]
ausentes_iniciales= rospy.get_param('ausentes')

uavs_ausentes=Int32MultiArray()
ausentes=ausentes_iniciales

while not rospy.is_shutdown():
  if rospy.has_param('cambios_ausentes'): ################
    intercambio=rospy.get_param('cambios_ausentes') ###################
    for i in range(0,len(ausentes)):
      if ausentes[i]==intercambio[1]:
        ausentes[i]=intercambio[0]
  uavs_ausentes.data=ausentes
  pub.publish(uavs_ausentes)
  rate.sleep()

```

Código A.4 simulacion_generador_sistema_uavs.py.

```
import rospy
from lxml import etree
from math import atan, cos, sin, pi, sqrt

numero_total_uavs=int(raw_input("Introduzca numero total de UAVs en el sistema
    :"))
vector_uavs=range(1,numero_total_uavs+1)

s=raw_input("Introduzca el numero de los UAVs con sentido antihorario
    Introduzca los numeros separados por espacios entre si: ")
vector_emparejados=map(int, s.split())


radio=float(raw_input("Introduzca el radio de las trayectorias: "))
omega=float(raw_input("Introduzca la velocidad angular de las trayectorias (
    mayor que cero): "))
vel_subida=float(raw_input("Introduzca velocidad de subida: "))
altura_especificada=float(raw_input("Introduzca altura de vuelo: "))

##############################################################################
s=raw_input("Introduzca el numero de los UAVs ausentes inicialmente Introduzca
    los numeros separados por espacios entre si: ")
ausentes=map(int, s.split())
r=len(ausentes)
vector_presentes=[]
for l in vector_uavs:
  eliminar=False
  for t in ausentes:
    if l==t:
      eliminar=True
      break
  if eliminar==False:
    vector_presentes.append(l)
##############################################################################

launch=etree.Element("launch")
doc=etree.ElementTree(launch)

arg_element=etree.SubElement(launch,'arg')
launch[0].attrib['name']='model'
launch[0].attrib['default']="$(find hector_quadrotor_description)/urdf/
    quadrotor.gazebo.xacro"

dic_centros={} #Diccionario que almacenara los centros de las trayectorias de
    cada UAV.
dic_emparejados={} #Diccionario que almacenara los UAVs sincronizados entre si.

coordenadas_xc=[]
coordenadas_yc=[]
coordenadas_x=[]
coordenadas_y=[]
for i in vector_uavs:
  print ("Uav numero %d") %(i)
```

```
  coordenadas_xc.append(float(raw_input("Introduzca coordenada x del centro:"))
      )
  coordenadas_yc.append(float(raw_input("Introduzca coordenada y del centro:"))
      )
  coordenadas_x.append(0)
  coordenadas_y.append(0)

  p="uav%d" %(i)
  dic_centros[p]=[coordenadas_xc[i-1], coordenadas_yc[i-1]]


coordenadas_x[0]=coordenadas_xc[0]+radio
coordenadas_y[0]=coordenadas_yc[0]



for i in vector_uavs:
  print ("Uav numero %d") %(i)
  s=raw_input("Introduzca el numero de los UAVs con los que esta sincronizado
      el UAV en orden ascendente Introduzca los numeros separados por espacios
      entre si: ")
  sincronizados=map(int, s.split())

  p="uav%d" %(i)
  dic_emparejados[p]=sincronizados

  if (coordenadas_y[i-1]-coordenadas_yc[i-1])==0:
    if (coordenadas_x[i-1]-coordenadas_xc[i-1])>=0:
      alpha_anterior=0
    if (coordenadas_x[i-1]-coordenadas_xc[i-1])<0:
      alpha_anterior=pi
  elif (coordenadas_x[i-1]-coordenadas_xc[i-1])==0:
    if (coordenadas_y[i-1]-coordenadas_yc[i-1])>=0:
      alpha_anterior=pi/2
    if (coordenadas_y[i-1]-coordenadas_yc[i-1])<0:
      alpha_anterior=3*pi/2
  else:
    incremento=(coordenadas_y[i-1]-coordenadas_yc[i-1])/(coordenadas_x[i-1]-
        coordenadas_xc[i-1])
    alpha_anterior=atan(incremento)
    if (coordenadas_y[i-1]-coordenadas_yc[i-1])/abs(coordenadas_y[i-1]-
        coordenadas_yc[i-1])>=0 and (coordenadas_x[i-1]-coordenadas_xc[i-1])/
        abs(coordenadas_x[i-1]-coordenadas_xc[i-1])<=0:
      alpha_anterior=alpha_anterior+pi
    if (coordenadas_y[i-1]-coordenadas_yc[i-1])/abs(coordenadas_y[i-1]-
        coordenadas_yc[i-1])<=0 and (coordenadas_x[i-1]-coordenadas_xc[i-1])/
        abs(coordenadas_x[i-1]-coordenadas_xc[i-1])<=0:
      alpha_anterior=alpha_anterior+pi
    if alpha_anterior<=0:
      alpha_anterior=alpha_anterior+2*pi
  for j in sincronizados:
    if coordenadas_x[j-1]==0 and coordenadas_y[j-1]==0:
      if (coordenadas_yc[j-1]-coordenadas_yc[i-1])==0:
        if (coordenadas_xc[j-1]-coordenadas_xc[i-1])>=0:
          beta=0
        if (coordenadas_xc[j-1]-coordenadas_xc[i-1])<0:
          beta=pi
```

```
      elif (coordenadas_xc[j-1]-coordenadas_xc[i-1])==0:
        if (coordenadas_yc[j-1]-coordenadas_yc[i-1])>=0:
          beta=pi/2
        if (coordenadas_yc[j-1]-coordenadas_yc[i-1])<0:
          beta=3*pi/2
      else:
        incremento=(coordenadas_yc[j-1]-coordenadas_yc[i-1])/(coordenadas_xc[j
            -1]-coordenadas_xc[i-1])
        beta=atan(incremento)
        if (coordenadas_yc[j-1]-coordenadas_yc[i-1])/abs(coordenadas_yc[j-1]-
            coordenadas_yc[i-1])>=0 and (coordenadas_xc[j-1]-coordenadas_xc[i
            -1])/abs(coordenadas_xc[j-1]-coordenadas_xc[i-1])<=0:
          beta=beta+pi
        if (coordenadas_yc[j-1]-coordenadas_yc[i-1])/abs(coordenadas_yc[j-1]-
            coordenadas_yc[i-1])<=0 and (coordenadas_xc[j-1]-coordenadas_xc[i
            -1])/abs(coordenadas_xc[j-1]-coordenadas_xc[i-1])<=0:
          beta=beta+pi
        if beta<=0:
          beta=beta+2*pi
      alpha=2*beta-alpha_anterior+pi
      coordenadas_x[j-1]=coordenadas_xc[j-1]+radio*cos(alpha)
      coordenadas_y[j-1]=coordenadas_yc[j-1]+radio*sin(alpha)

  coordenada_x=coordenadas_x[i-1]
  coordenada_y=coordenadas_y[i-1]

  group_element = etree.SubElement(launch, 'group')
  launch[i].attrib["ns"]="uav%d" %(i)
  group_element.append(etree.Element("include"))
  group_element[0].attrib['file']="$(find hector_quadrotor_gazebo)/launch/
      spawn_quadrotor.launch"

  group_element[0].append(etree.Element('arg'))
  group_element[0][0].attrib['name']='name'
  group_element[0][0].attrib['value']='uav%d' %(i)
  group_element[0].append(etree.Element('arg'))
  group_element[0][1].attrib['name']='tf_prefix'
  group_element[0][1].attrib['value']='uav%d' %(i)
  group_element[0].append(etree.Element('arg'))
  group_element[0][2].attrib['name']='model'
  group_element[0][2].attrib['value']='$(arg model)'
  group_element[0].append(etree.Element('arg'))
  group_element[0][3].attrib['name']='x'
  group_element[0][3].attrib['value']='%f' %(coordenada_x)
  group_element[0].append(etree.Element('arg'))
  group_element[0][4].attrib['name']='y'
  group_element[0][4].attrib['value']='%f' %(coordenada_y)

################
dic_betas={}
dic_pos_cambio={}
dic_dis_centros={}
for i in vector_uavs:
  p="uav%d" %(i)
  emparejados=dic_emparejados[p]
  vector_betas=[]
  vector_pos_cambio=[]
```

```
  vector_dis_centros=[]
  for j in emparejados:
    if (coordenadas_yc[j-1]-coordenadas_yc[i-1])==0:
      if (coordenadas_xc[j-1]-coordenadas_xc[i-1])>=0:
        beta=0
      if (coordenadas_xc[j-1]-coordenadas_xc[i-1])<0:
        beta=pi
    elif (coordenadas_xc[j-1]-coordenadas_xc[i-1])==0:
      if (coordenadas_yc[j-1]-coordenadas_yc[i-1])>=0:
        beta=pi/2
      if (coordenadas_yc[j-1]-coordenadas_yc[i-1])<0:
        beta=3*pi/2
    else:
      incremento=(coordenadas_yc[j-1]-coordenadas_yc[i-1])/(coordenadas_xc[j-1]-
          coordenadas_xc[i-1])
      beta=atan(incremento)
      if (coordenadas_yc[j-1]-coordenadas_yc[i-1])/abs(coordenadas_yc[j-1]-
          coordenadas_yc[i-1])>=0 and (coordenadas_xc[j-1]-coordenadas_xc[i-1])/
          abs(coordenadas_xc[j-1]-coordenadas_xc[i-1])<=0:
        beta=beta+pi
      if (coordenadas_yc[j-1]-coordenadas_yc[i-1])/abs(coordenadas_yc[j-1]-
          coordenadas_yc[i-1])<=0 and (coordenadas_xc[j-1]-coordenadas_xc[i-1])/
          abs(coordenadas_xc[j-1]-coordenadas_xc[i-1])<=0:
        beta=beta+pi
      if beta<=0:
        beta=beta+2*pi
    dis_centros=sqrt((coordenadas_xc[j-1]-coordenadas_xc[i-1])**2+(
        coordenadas_yc[j-1]-coordenadas_yc[i-1])**2)
    vector_betas.append(beta)
    vector_pos_cambio.append([coordenadas_xc[i-1]+radio*cos(beta),
        coordenadas_yc[i-1]+radio*sin(beta)])
    vector_dis_centros.append(dis_centros)
  dic_betas[p]=vector_betas
  dic_pos_cambio[p]=vector_pos_cambio
  dic_dis_centros[p]=vector_dis_centros

rospy.set_param('diccionario_centros', dic_centros)
rospy.set_param('diccionario_emparejados', dic_emparejados)
rospy.set_param('diccionario_posiciones_cambio', dic_pos_cambio)
rospy.set_param('diccionario_distancia_centros', dic_dis_centros)
rospy.set_param('diccionario_betas', dic_betas)

rospy.set_param('ausentes', ausentes)

outFile = open('spawn_quadrotors.xml', 'w')
doc.write(outFile)

#############################################################################
launch=etree.Element("launch")
doc=etree.ElementTree(launch)

arg_element=etree.SubElement(launch,'arg')
launch[0].attrib['name']='paused'
launch[0].attrib['default']="false"

arg_element=etree.SubElement(launch,'arg')
launch[1].attrib['name']='use_sim_time'
```

```
launch[1].attrib['default']="true"

arg_element=etree.SubElement(launch,'arg')
launch[2].attrib['name']='gui'
launch[2].attrib['default']="true"

arg_element=etree.SubElement(launch,'arg')
launch[3].attrib['name']='headless'
launch[3].attrib['default']="false"

arg_element=etree.SubElement(launch,'arg')
launch[4].attrib['name']='debug'
launch[4].attrib['default']="false"

include_element=etree.SubElement(launch,'include')
launch[5].attrib['file']='$(find gazebo_ros)/launch/empty_world.launch'

include_element.append(etree.Element("arg"))
include_element[0].attrib['name']='paused'
include_element[0].attrib['value']='$(arg paused)'

include_element.append(etree.Element("arg"))
include_element[1].attrib['name']='use_sim_time'
include_element[1].attrib['value']='$(arg use_sim_time)'

include_element.append(etree.Element("arg"))
include_element[2].attrib['name']='gui'
include_element[2].attrib['value']='$(arg gui)'

include_element.append(etree.Element("arg"))
include_element[3].attrib['name']='headless'
include_element[3].attrib['value']='$(arg headless)'

include_element.append(etree.Element("arg"))
include_element[4].attrib['name']='debug'
include_element[4].attrib['value']='$(arg debug)'

include_element=etree.SubElement(launch,'include')
launch[6].attrib['file']='$(find wanderbot)/launch/spawn_quadrotors.xml'

outFile = open('system_uavs_empty_world.xml', 'w')
doc.write(outFile)

########################################################################
launch=etree.Element("launch")
doc=etree.ElementTree(launch)

for k in range(1,len(vector_presentes)+1):
  sentido=-1
  for x in vector_emparejados:
    if x==vector_presentes[k-1]:
      sentido=1

  node_element = etree.SubElement(launch, 'node')
  launch[k-1].attrib["name"]="simulacion_navegacion_sincronizada_uav%d" %(
      vector_presentes[k-1])
  launch[k-1].attrib["pkg"]="wanderbot"
```

```
    launch[k-1].attrib["type"]="simulacion_navegacion_sincronizada.py"
285
    node_element.append(etree.Element("remap"))
    node_element[0].attrib['from']="ground_truth/state"
288 node_element[0].attrib['to']="uav%d/ground_truth/state" %(vector_presentes[k
        -1])
    node_element.append(etree.Element("remap"))
291 node_element[1].attrib['from']="cmd_vel"
    node_element[1].attrib['to']="uav%d/cmd_vel" %(vector_presentes[k-1])

294 node_element.append(etree.Element("remap"))
    node_element[2].attrib['from']="radio_deseado"
    node_element[2].attrib['to']="simulacion_navegacion_sincronizada_uav%d/
297     radio_deseado" %(vector_presentes[k-1])
    node_element.append(etree.Element("param"))
    node_element[3].attrib['name']="radio_deseado"
300 node_element[3].attrib['value']="%f" %(radio)
    node_element.append(etree.Element("remap"))
    node_element[4].attrib['from']="omega_deseada"
303 node_element[4].attrib['to']="simulacion_navegacion_sincronizada_uav%d/
        omega_deseada" %(vector_presentes[k-1])
    node_element.append(etree.Element("param"))
306 node_element[5].attrib['name']="omega_deseada"
    node_element[5].attrib['value']="%f" %(omega)
    node_element.append(etree.Element("remap"))
309 node_element[6].attrib['from']="vel_subida_deseada"
    node_element[6].attrib['to']="simulacion_navegacion_sincronizada_uav%d/
        vel_subida_deseada" %(vector_presentes[k-1])
312 node_element.append(etree.Element("param"))
    node_element[7].attrib['name']="vel_subida_deseada"
    node_element[7].attrib['value']="%f" %(vel_subida)
315 node_element.append(etree.Element("remap"))
    node_element[8].attrib['from']="altura_deseada"
    node_element[8].attrib['to']="simulacion_navegacion_sincronizada_uav%d/
318     altura_deseada" %(vector_presentes[k-1])
    node_element.append(etree.Element("param"))
    node_element[9].attrib['name']="altura_deseada"
321 node_element[9].attrib['value']="%f" %(altura_especificada)
    node_element.append(etree.Element("remap"))
    node_element[10].attrib['from']="sentido_giro"
324 node_element[10].attrib['to']="simulacion_navegacion_sincronizada_uav%d/
        sentido_giro" %(vector_presentes[k-1])
    node_element.append(etree.Element("param"))
327 node_element[11].attrib['name']="sentido_giro"
    node_element[11].attrib['value']="%d" %(sentido)

330 node_element.append(etree.Element("remap"))
    node_element[12].attrib['from']="coordenada_x_centro"
    node_element[12].attrib['to']="simulacion_navegacion_sincronizada_uav%d/
333     coordenada_x_centro" %(vector_presentes[k-1])
    node_element.append(etree.Element("param"))
    node_element[13].attrib['name']="coordenada_x_centro"
336 node_element[13].attrib['value']="%f" %(coordenadas_xc[vector_presentes[k
        -1]-1])
    node_element.append(etree.Element("remap"))
339 node_element[14].attrib['from']="coordenada_y_centro"
```

```
  node_element[14].attrib['to']="simulacion_navegacion_sincronizada_uav%d/
      coordenada_y_centro" %(vector_presentes[k-1])
  node_element.append(etree.Element("param"))
  node_element[15].attrib['name']="coordenada_y_centro"
  node_element[15].attrib['value']="%f" %(coordenadas_yc[vector_presentes[k
      -1]-1])

  node_element.append(etree.Element("remap"))
  node_element[16].attrib['from']="localizador_uav"
  node_element[16].attrib['to']="simulacion_navegacion_sincronizada_uav%d/
      localizador_uav" %(vector_presentes[k-1])
  node_element.append(etree.Element("param"))
  node_element[17].attrib['name']="localizador_uav"
  node_element[17].attrib['value']="%d" %(vector_presentes[k-1])
  node_element.append(etree.Element("remap"))
  node_element[18].attrib['from']="numero_total_ausentes"
  node_element[18].attrib['to']="simulacion_navegacion_sincronizada_uav%d/
      numero_total_ausentes" %(vector_presentes[k-1])
  node_element.append(etree.Element("param"))
  node_element[19].attrib['name']="numero_total_ausentes"
  node_element[19].attrib['value']="%d" %(r)

node_element = etree.SubElement(launch, 'node')
launch[k].attrib["name"]="ausentes_publisher"
launch[k].attrib["pkg"]="wanderbot"
launch[k].attrib["type"]="ausentes_publisher.py"

outFile = open('patrulla_circular.xml', 'w')
doc.write(outFile)

#############################################################################
launch=etree.Element("launch")
doc=etree.ElementTree(launch)

for k in range(1,len(vector_presentes)+1):
  sentido=-1
  for x in vector_emparejados:
    if x==vector_presentes[k-1]:
      sentido=1

  node_element = etree.SubElement(launch, 'node')
  launch[k-1].attrib["name"]="simulacion_subida_y_orientacion_uav%d" %(
      vector_presentes[k-1])
  launch[k-1].attrib["pkg"]="wanderbot"
  launch[k-1].attrib["type"]="simulacion_subida_y_orientacion.py"

  node_element.append(etree.Element("remap"))
  node_element[0].attrib['from']="ground_truth/state"
  node_element[0].attrib['to']="uav%d/ground_truth/state" %(vector_presentes[k
      -1])
  node_element.append(etree.Element("remap"))
  node_element[1].attrib['from']="cmd_vel"
  node_element[1].attrib['to']="uav%d/cmd_vel" %(vector_presentes[k-1])

  node_element.append(etree.Element("remap"))
  node_element[2].attrib['from']="coordenada_x_centro"
```

```
396   node_element[2].attrib['to']="simulacion_subida_y_orientacion_uav%d/
          coordenada_x_centro" %(vector_presentes[k-1])
      node_element.append(etree.Element("param"))
399   node_element[3].attrib['name']="coordenada_x_centro"
      node_element[3].attrib['value']="%f" %(coordenadas_xc[vector_presentes[k
          -1]-1])
402   node_element.append(etree.Element("remap"))
      node_element[4].attrib['from']="coordenada_y_centro"
      node_element[4].attrib['to']="simulacion_subida_y_orientacion_uav%d/
405       coordenada_y_centro" %(vector_presentes[k-1])
      node_element.append(etree.Element("param"))
      node_element[5].attrib['name']="coordenada_y_centro"
408   node_element[5].attrib['value']="%f" %(coordenadas_yc[vector_presentes[k
          -1]-1])
      node_element.append(etree.Element("remap"))
411   node_element[6].attrib['from']="sentido_giro"
      node_element[6].attrib['to']="simulacion_subida_y_orientacion_uav%d/
          sentido_giro" %(vector_presentes[k-1])
414   node_element.append(etree.Element("param"))
      node_element[7].attrib['name']="sentido_giro"
      node_element[7].attrib['value']="%d" %(sentido)
417
  outFile = open('orientacion_prevuelo.xml', 'w')
420 doc.write(outFile)
```

Apéndice B

Ejemplos de archivos .xml generados para un determinado sistema de UAVs.

Código B.1 spawn_quadrotors.xml.

```
<?xml version="1.0"?>
-<launch>
  <arg default="$(find hector_quadrotor_description)/urdf/quadrotor.gazebo.
      xacro" name="model"/>
  -<group ns="uav1">
    -<include file="$(find hector_quadrotor_gazebo)/launch/spawn_quadrotor.
        launch">
      <arg name="name" value="uav1"/>
      <arg name="tf_prefix" value="uav1"/>
      <arg name="model" value="$(arg model)"/>
      <arg name="x" value="-2.000000"/>
      <arg name="y" value="0.000000"/>
    </include>
  </group>
  -<group ns="uav2">
    -<include file="$(find hector_quadrotor_gazebo)/launch/spawn_quadrotor.
        launch">
      <arg name="name" value="uav2"/>
      <arg name="tf_prefix" value="uav2"/>
      <arg name="model" value="$(arg model)"/>
      <arg name="x" value="-0.000000"/>
      <arg name="y" value="2.000000"/>
    </include>
  </group>
  -<group ns="uav3">
    -<include file="$(find hector_quadrotor_gazebo)/launch/spawn_quadrotor.
        launch">
      <arg name="name" value="uav3"/>
      <arg name="tf_prefix" value="uav3"/>
      <arg name="model" value="$(arg model)"/>
      <arg name="x" value="0.000000"/>
      <arg name="y" value="-2.000000"/>
    </include>
  </group>
  -<group ns="uav4">
```

```
    -<include file="$(find hector_quadrotor_gazebo)/launch/spawn_quadrotor.
        launch">
      <arg name="name" value="uav4"/>
      <arg name="tf_prefix" value="uav4"/>
      <arg name="model" value="$(arg model)"/>
      <arg name="x" value="2.000000"/>
      <arg name="y" value="0.000000"/>
    </include>
  </group>
</launch>
```

Código B.2 system_uavs_empty_world.xml.

```
<?xml version="1.0"?>
-<launch>
  <arg default="false" name="paused"/>
  <arg default="true" name="use_sim_time"/>
  <arg default="true" name="gui"/>
  <arg default="false" name="headless"/>
  <arg default="false" name="debug"/>
  -<include file="$(find gazebo_ros)/launch/empty_world.launch">
    <arg name="paused" value="$(arg paused)"/>
    <arg name="use_sim_time" value="$(arg use_sim_time)"/>
    <arg name="gui" value="$(arg gui)"/>
    <arg name="headless" value="$(arg headless)"/>
    <arg name="debug" value="$(arg debug)"/>
  </include>
  <include file="$(find wanderbot)/launch/spawn_quadrotors.xml"/>
</launch>
```

Código B.3 patrulla_circular.xml.

```
<?xml version="1.0"?>
-<launch>
  -<node type="simulacion_navegacion_sincronizada.py" pkg="wanderbot" name="
      simulacion_navegacion_sincronizada_uav1">
    <remap to="uav1/ground_truth/state" from="ground_truth/state"/>
    <remap to="uav1/cmd_vel" from="cmd_vel"/>
    <remap to="simulacion_navegacion_sincronizada_uav1/radio_deseado" from="
        radio_deseado"/>
    <param name="radio_deseado" value="3.000000"/>
    <remap to="simulacion_navegacion_sincronizada_uav1/omega_deseada" from="
        omega_deseada"/>
    <param name="omega_deseada" value="0.150000"/>
    <remap to="simulacion_navegacion_sincronizada_uav1/vel_subida_deseada" from
        ="vel_subida_deseada"/>
    <param name="vel_subida_deseada" value="0.150000"/>
    <remap to="simulacion_navegacion_sincronizada_uav1/altura_deseada" from="
        altura_deseada"/>
    <param name="altura_deseada" value="1.000000"/>
    <remap to="simulacion_navegacion_sincronizada_uav1/sentido_giro" from="
        sentido_giro"/>
    <param name="sentido_giro" value="1"/>
    <remap to="simulacion_navegacion_sincronizada_uav1/coordenada_x_centro"
        from="coordenada_x_centro"/>
```

```
    <param name="coordenada_x_centro" value="-5.000000"/>
    <remap to="simulacion_navegacion_sincronizada_uav1/coordenada_y_centro"
27      from="coordenada_y_centro"/>
    <param name="coordenada_y_centro" value="0.000000"/>
    <remap to="simulacion_navegacion_sincronizada_uav1/localizador_uav" from="
30      localizador_uav"/>
    <param name="localizador_uav" value="1"/>
    <remap to="simulacion_navegacion_sincronizada_uav1/numero_total_ausentes"
33      from="numero_total_ausentes"/>
    <param name="numero_total_ausentes" value="1"/>
  </node>
36  -<node type="simulacion_navegacion_sincronizada.py" pkg="wanderbot" name="
      simulacion_navegacion_sincronizada_uav2">
    <remap to="uav2/ground_truth/state" from="ground_truth/state"/>
39  <remap to="uav2/cmd_vel" from="cmd_vel"/>
    <remap to="simulacion_navegacion_sincronizada_uav2/radio_deseado" from="
      radio_deseado"/>
42  <param name="radio_deseado" value="3.000000"/>
    <remap to="simulacion_navegacion_sincronizada_uav2/omega_deseada" from="
      omega_deseada"/>
45  <param name="omega_deseada" value="0.150000"/>
    <remap to="simulacion_navegacion_sincronizada_uav2/vel_subida_deseada" from
      ="vel_subida_deseada"/>
48  <param name="vel_subida_deseada" value="0.150000"/>
    <remap to="simulacion_navegacion_sincronizada_uav2/altura_deseada" from="
      altura_deseada"/>
51  <param name="altura_deseada" value="1.000000"/>
    <remap to="simulacion_navegacion_sincronizada_uav2/sentido_giro" from="
      sentido_giro"/>
54  <param name="sentido_giro" value="-1"/>
    <remap to="simulacion_navegacion_sincronizada_uav2/coordenada_x_centro"
      from="coordenada_x_centro"/>
57  <param name="coordenada_x_centro" value="0.000000"/>
    <remap to="simulacion_navegacion_sincronizada_uav2/coordenada_y_centro"
      from="coordenada_y_centro"/>
60  <param name="coordenada_y_centro" value="5.000000"/>
    <remap to="simulacion_navegacion_sincronizada_uav2/localizador_uav" from="
      localizador_uav"/>
63  <param name="localizador_uav" value="2"/>
    <remap to="simulacion_navegacion_sincronizada_uav2/numero_total_ausentes"
      from="numero_total_ausentes"/>
66  <param name="numero_total_ausentes" value="1"/>
  </node>
  -<node type="simulacion_navegacion_sincronizada.py" pkg="wanderbot" name="
69    simulacion_navegacion_sincronizada_uav3">
    <remap to="uav3/ground_truth/state" from="ground_truth/state"/>
    <remap to="uav3/cmd_vel" from="cmd_vel"/>
72  <remap to="simulacion_navegacion_sincronizada_uav3/radio_deseado" from="
      radio_deseado"/>
    <param name="radio_deseado" value="3.000000"/>
75  <remap to="simulacion_navegacion_sincronizada_uav3/omega_deseada" from="
      omega_deseada"/>
    <param name="omega_deseada" value="0.150000"/>
78  <remap to="simulacion_navegacion_sincronizada_uav3/vel_subida_deseada" from
      ="vel_subida_deseada"/>
    <param name="vel_subida_deseada" value="0.150000"/>
```

```
81      <remap to="simulacion_navegacion_sincronizada_uav3/altura_deseada" from="
            altura_deseada"/>
        <param name="altura_deseada" value="1.000000"/>
84      <remap to="simulacion_navegacion_sincronizada_uav3/sentido_giro" from="
            sentido_giro"/>
        <param name="sentido_giro" value="-1"/>
87      <remap to="simulacion_navegacion_sincronizada_uav3/coordenada_x_centro"
            from="coordenada_x_centro"/>
        <param name="coordenada_x_centro" value="0.000000"/>
90      <remap to="simulacion_navegacion_sincronizada_uav3/coordenada_y_centro"
            from="coordenada_y_centro"/>
        <param name="coordenada_y_centro" value="-5.000000"/>
93      <remap to="simulacion_navegacion_sincronizada_uav3/localizador_uav" from="
            localizador_uav"/>
        <param name="localizador_uav" value="3"/>
96      <remap to="simulacion_navegacion_sincronizada_uav3/numero_total_ausentes"
            from="numero_total_ausentes"/>
        <param name="numero_total_ausentes" value="1"/>
99    </node>
      -<node type="simulacion_navegacion_sincronizada.py" pkg="wanderbot" name="
          simulacion_navegacion_sincronizada_uav4">
102     <remap to="uav4/ground_truth/state" from="ground_truth/state"/>
        <remap to="uav4/cmd_vel" from="cmd_vel"/>
        <remap to="simulacion_navegacion_sincronizada_uav4/radio_deseado" from="
105         radio_deseado"/>
        <param name="radio_deseado" value="3.000000"/>
        <remap to="simulacion_navegacion_sincronizada_uav4/omega_deseada" from="
108         omega_deseada"/>
        <param name="omega_deseada" value="0.150000"/>
        <remap to="simulacion_navegacion_sincronizada_uav4/vel_subida_deseada" from
111         ="vel_subida_deseada"/>
        <param name="vel_subida_deseada" value="0.150000"/>
        <remap to="simulacion_navegacion_sincronizada_uav4/altura_deseada" from="
114         altura_deseada"/>
        <param name="altura_deseada" value="1.000000"/>
        <remap to="simulacion_navegacion_sincronizada_uav4/sentido_giro" from="
117         sentido_giro"/>
        <param name="sentido_giro" value="1"/>
        <remap to="simulacion_navegacion_sincronizada_uav4/coordenada_x_centro"
120         from="coordenada_x_centro"/>
        <param name="coordenada_x_centro" value="5.000000"/>
        <remap to="simulacion_navegacion_sincronizada_uav4/coordenada_y_centro"
123         from="coordenada_y_centro"/>
        <param name="coordenada_y_centro" value="0.000000"/>
        <remap to="simulacion_navegacion_sincronizada_uav4/localizador_uav" from="
126         localizador_uav"/>
        <param name="localizador_uav" value="4"/>
        <remap to="simulacion_navegacion_sincronizada_uav4/numero_total_ausentes"
129         from="numero_total_ausentes"/>
        <param name="numero_total_ausentes" value="1"/>
      </node>
132   <node type="ausentes_publisher.py" pkg="wanderbot" name="ausentes_publisher
          "/>
    </launch>
135
```

Código B.4 orientacion_prevuelo.xml.

```
<?xml version="1.0"?>
-<launch>
  -<node type="simulacion_subida_y_orientacion.py" pkg="wanderbot" name="
      simulacion_subida_y_orientacion_uav1">
    <remap to="uav1/ground_truth/state" from="ground_truth/state"/>
    <remap to="uav1/cmd_vel" from="cmd_vel"/>
    <remap to="simulacion_subida_y_orientacion_uav1/coordenada_x_centro" from="
        coordenada_x_centro"/>
    <param name="coordenada_x_centro" value="-5.000000"/>
    <remap to="simulacion_subida_y_orientacion_uav1/coordenada_y_centro" from="
        coordenada_y_centro"/>
    <param name="coordenada_y_centro" value="0.000000"/>
    <remap to="simulacion_subida_y_orientacion_uav1/sentido_giro" from="
        sentido_giro"/>
    <param name="sentido_giro" value="1"/>
  </node>
  -<node type="simulacion_subida_y_orientacion.py" pkg="wanderbot" name="
      simulacion_subida_y_orientacion_uav2">
    <remap to="uav2/ground_truth/state" from="ground_truth/state"/>
    <remap to="uav2/cmd_vel" from="cmd_vel"/>
    <remap to="simulacion_subida_y_orientacion_uav2/coordenada_x_centro" from="
        coordenada_x_centro"/>
    <param name="coordenada_x_centro" value="0.000000"/>
    <remap to="simulacion_subida_y_orientacion_uav2/coordenada_y_centro" from="
        coordenada_y_centro"/>
    <param name="coordenada_y_centro" value="5.000000"/>
    <remap to="simulacion_subida_y_orientacion_uav2/sentido_giro" from="
        sentido_giro"/>
    <param name="sentido_giro" value="-1"/>
  </node>
  -<node type="simulacion_subida_y_orientacion.py" pkg="wanderbot" name="
      simulacion_subida_y_orientacion_uav3">
    <remap to="uav3/ground_truth/state" from="ground_truth/state"/>
    <remap to="uav3/cmd_vel" from="cmd_vel"/>
    <remap to="simulacion_subida_y_orientacion_uav3/coordenada_x_centro" from="
        coordenada_x_centro"/>
    <param name="coordenada_x_centro" value="0.000000"/>
    <remap to="simulacion_subida_y_orientacion_uav3/coordenada_y_centro" from="
        coordenada_y_centro"/>
    <param name="coordenada_y_centro" value="-5.000000"/>
    <remap to="simulacion_subida_y_orientacion_uav3/sentido_giro" from="
        sentido_giro"/>
    <param name="sentido_giro" value="-1"/>
  </node>
  -<node type="simulacion_subida_y_orientacion.py" pkg="wanderbot" name="
      simulacion_subida_y_orientacion_uav4">
    <remap to="uav4/ground_truth/state" from="ground_truth/state"/>
    <remap to="uav4/cmd_vel" from="cmd_vel"/>
    <remap to="simulacion_subida_y_orientacion_uav4/coordenada_x_centro" from="
        coordenada_x_centro"/>
    <param name="coordenada_x_centro" value="5.000000"/>
    <remap to="simulacion_subida_y_orientacion_uav4/coordenada_y_centro" from="
        coordenada_y_centro"/>
    <param name="coordenada_y_centro" value="0.000000"/>
```

```
    <remap to="simulacion_subida_y_orientacion_uav4/sentido_giro" from="
57      sentido_giro"/>
    <param name="sentido_giro" value="1"/>
  </node>
60 </launch>
```

Índice de Figuras

Índice de Códigos

Bibliografía

[1] D. Glade, "Unmanned aerial vehicles: Implications for military operations," tech. rep., DTIC Document, 2000.

[2] S. M. Adams and C. J. Friedland, *A survey of unmanned aerial vehicle (UAV) usage for imagery collection in disaster research and management*. publisher not identified, 2011.

[3] Y. U. Cao, A. S. Fukunaga, and A. Kahng, "Cooperative mobile robotics: Antecedents and directions," *Autonomous robots*, vol. 4, no. 1, pp. 7–27, 1997.

[4] A. Bicchi, A. Danesi, G. Dini, S. La Porta, L. Pallottino, I. M. Savino, and R. Schiavi, "Heterogeneous wireless multirobot system.," *IEEE Robot. Automat. Mag.*, vol. 15, no. 1, pp. 62–70, 2008.

[5] J. M. Díaz-Bánez, E. Caraballo, M. Lopez, S. Bereg, I. Maza, and A. Ollero, "The synchronization problem for information exchange between aerial robots under communication constraints," in *Robotics and Automation (ICRA), 2015 IEEE International Conference on*, pp. 4650–4655, IEEE, 2015.

[6] W. Burgard, M. Moors, C. Stachniss, and F. E. Schneider, "Coordinated multi-robot exploration," *Robotics, IEEE Transactions on*, vol. 21, no. 3, pp. 376–386, 2005.

[7] M. Ferrati and L. Pallottino, "A time expanded network based algorithm for safe and efficient distributed multi-agent coordination," in *Decision and Control (CDC), 2013 IEEE 52nd Annual Conference on*, pp. 2805–2810, IEEE, 2013.

[8] R. Zlot and A. Stentz, "Market-based multirobot coordination for complex tasks," *The International Journal of Robotics Research*, vol. 25, no. 1, pp. 73–101, 2006.

[9] W. Sheng, Q. Yang, J. Tan, and N. Xi, "Distributed multi-robot coordination in area exploration," *Robotics and Autonomous Systems*, vol. 54, no. 12, pp. 945–955, 2006.

[10] S. Bereg, L. Caraballo, J. M. Díaz-Báñez, and M. Lopez, "Resilience of a synchronized multi-agent system," *arXiv preprint arXiv:1604.08804*, 2016.

[11] M. Quigley, B. Gerkey, and W. D. Smart, "Programming robots with ros," 2015.

[12] M. Quigley, K. Conley, B. Gerkey, J. Faust, T. Foote, J. Leibs, R. Wheeler, and A. Y. Ng, "Ros: an open-source robot operating system," in *ICRA workshop on open source software*, vol. 3, p. 5, 2009.

[13] R. B. Rusu, "ROS Overview." *http://wiki.ros.org/Events/CoTeSys-ROS-School?action=AttachFile&do=view&target=ros_overview.pdf*, 2010. [Online; accessed 27-December-2015].

[14] V. autores, "wiki ros." *http://wiki.ros.org/es*.

[15] V. autores, "Foro oficial de preguntas ros." *http://answers.ros.org/questions/*.