
Proyecto de Sistemas Informáticos

Facultad de Informática

Universidad Complutense de Madrid

Curso 2008-2009

# Dolphin

## Cálculo de trayectorias óptimas y simulación bajo Google Earth de agentes marinos no tripulados

Creado por

Miguel Ángel Casado Hernández

Elena Montojo Vanrell

Francisco Javier Samper Sánchez

Dirigido por

Matilde Santos Peñas

Departamento de Arquitectura de Computadores y Automática

## RESUMEN

Desarrollo de una plataforma de simulación para la dinámica de agentes (vehículos marinos) con la integración de imágenes reales de satélite.

Implementación de un sistema de tratamiento de imágenes RGB así como de un sistema de cálculo y optimización de trayectorias para recorridos marítimos sobre las diversas zonas de la Tierra.



## ABSTRACT

Development of a simulation platform showing the movement of intelligent agents (marine vehicles) through real satellite images.

Implementation of a system for the treatment of RGB images as well as a system for calculus and optimization of worldwide covered marine trajectories.



## PALABRAS CLAVE

•Agente inteligente

•JADE

•KML – Google Earth

•Optimización de trayectorias

•LVQ

•Plataforma de simulación

•Vehículos marinos

•Dinámica del comportamiento de un barco

# Índice de contenido

**Capítulo 1 - Motivación y objetivos.....1**

1.1 - Motivación.....1

1.2 - Objetivos.....3

**Capítulo 2 - Estado del arte.....4**

2.1 - Catástrofes naturales:.....4

2.2 - Agentes y vehículos no tripulados.....5

2.2.1 - Concepto de agente.....5

2.2.2 - Modelos y arquitecturas de agente.....9

2.2.3 - ¿Cómo se comporta un agente?.....11

2.2.4 - Ejemplos de sistemas que integran agentes.....12

2.2.5 - Vehículos no tripulados.....14

2.3 - Trayectorias.....16

2.4 - Herramientas: Jade.....18

**Capítulo 3 - Descripción del sistema.....24**

3.1 - Modelo del barco.....28

3.1.1 - Ecuaciones del movimiento .....29

3.1.2 - Modelo sobre Simulink.....32

3.2 - Migración a Java.....36

3.2.1 - Funciones de transferencia .....37

3.2.2 - Módulo integrador .....38

3.3 - Validación del modelo .....38

3.4 - Implementación del barco como agente.....43

**Capítulo 4 - Google Earth y tratamiento de imágenes.....47**

4.1 - Google Earth.....48

4.1.1 - Marcas de señalización....48
4.1.2 - Formato de archivos....49
4.2 - Tratamiento de imágenes....51
4.2.1 - Procesamiento y binarización de imágenes....51
**Capítulo 5 - Planificación de trayectorias....58**
5.1 - Primera aproximación....58
5.2 - Segunda aproximación: Grafos de visibilidad....59
5.3 - Tercera aproximación: Pathfinding....61
5.3.1 - Pseudo-código del algoritmo....62
5.3.2 - Mejora del algoritmo....66
5.4 - Relación con el modelo del barco....67
5.5 - Sistemas de representación de un punto....70
5.5.1 - Posición en una matriz....71
5.5.2 - Metros....72
5.5.3 - Grados....72
**Capítulo 6 - Plataforma de simulación....74**
6.1 - Pantalla principal....75
6.2 - Generación de escenarios....75
6.3 - Diseño de Misiones actuales....79
6.4 - Información de los barcos actuales (agentes)....82
6.5 - Resultado de la simulación....83
**Capítulo 7 - Conclusiones y ampliaciones....85**
7.1 - Conclusiones....85
7.2 - Ampliaciones....86
**Capítulo 8 - Bibliografía y referencias....88**

# Capítulo 1 -

# Motivación y objetivos

## 1.1 - Motivación

Este proyecto se inició hace 4 años con la intención de profundizar el estudio y las posibilidades de trabajo de vehículos no tripulados, comportándose como agentes inteligentes.

En estos cuatro años se han generado, con esta, tres versiones (quizá mejor llamadas ampliaciones) de lo que empezó como un modelo en Java del comportamiento de un avión no tripulado (curso 2005-2006).

Al año siguiente se realizó una gran ampliación. En esta fijaron como objetivo de las búsquedas la detección de barcos u objetivos en el mar, estando centrados en la colaboración en naufragios y localización de pateras. Ampliaron la flota de uno a tres agentes que se comunicaban mediante radio y volaban en formación. Estos aviones, cada uno con una función definida, conseguían sacar fotos de la superficie marítima, detectando, gracias a una completa red neuronal, la localización de los objetivos. Con esto hemos llegado a nuestro

curso 2008-2009.

Intentando ampliar en otra dirección, este proyecto continúa el estudio sobre los agentes, ahora buscando el correcto movimiento y control de barcos no tripulados. Pareciéndonos interesante la temática de los naufragios y detección de objetivos, empezamos a interesarnos y a buscar información real sobre el tema.

En poco tiempo se puede ver que las principales catástrofes que se producen en el mar, habiendo también muchas otras, son humanas. La mayoría de ellas son muertes por naufragios (principalmente en pateras) o catástrofes marinas como fugas de petróleo. De éstas y de los medios actuales de colaboración y prevención hablaremos más adelante.

No perdiendo el interés en este tema, es importante mencionar que el trabajo en naufragios es sólo una aplicación del problema que tratamos: el trabajo con los agentes y algoritmos óptimos de recorrido y búsqueda, que aunque en este caso están centrados en superficies marítimas, pueden ser fácilmente migrados a otras superficies.

También hacer hincapié en que se busca una simulación real. Dejando al margen las condiciones de cada simulación, se ha trabajado buscando la mayor semejanza con la realidad.

Se puede ver con una simple ejecución, y también, llegado el momento hablaremos de ello, cómo se ha avanzado en el realismo de las superficies a recorrer. También cabe mencionar que todos los agentes están afectados por condiciones ambientales que buscan también la realidad (en nuestro caso, corrientes marítimas), característica también detallada en el debido punto de este documento.

## 1.2 - Objetivos

Las metas que se propusieron fueron:

- Crear un modelo en tiempo real del movimiento de un barco afectado por distintas corrientes. Este modelo se podía obtener a partir de una migración a Java de un proyecto ya creado en Simulink. Este modelo aportaba un desplazamiento en función de una trayectoria aplicada en forma de ángulo.

- Tras esto, se busca conseguir una simulación controlada de un punto origen a uno objetivo de manera continuada. Lo que queremos es conseguir un total control de la posición y velocidad en las curvas.

- Desarrollar una interfaz, buscando el mayor realismo del terreno, que pudiera ser el entorno del desplazamiento anteriormente definido. En este sentido se ha pensado en usar como soporte Google Earth para la visualización de los resultados y como generador de escenarios.

- Tratamiento del barco como un agente inteligente, que espere y entienda órdenes y comunique su estado. Se busca crear un entorno que permita una fácil incorporación de las funcionalidades objetivo y que facilite en la medida de lo posible cualquier linea de ampliación, desde modificaciones de propiedades hasta la creación de un sistema multi-agente pasando por la adaptación al trabajo en colaboración con otros medios de transporte.

- Crear algoritmos de recorrido óptimo que para cualquier posición inicial y objetivo, permitan conocer la trayectoria más corta posible entre ambos. Uno de los objetivos principales es que al tratarse de vehículos marinos se requiere evitar islas y zonas no navegables.

# Capítulo 2 -

# Estado del arte

Para empezar a familiarizarnos con la aplicación, vamos a empezar dando de nuevo importancia a la actualidad de cada una de las áreas de trabajo mencionadas anteriormente, así como el estado de estudio actual que se ha obtenido.

## 2.1 - Catástrofes naturales:

No nos hace falta buscar mucho para encontrar un ejemplo de estas catástrofes. Como titular de hoy en los periódicos encontramos [público.es]:

> *"Al menos 15 de los 21 fallecidos en Lanzarote eran menores: Hoy se han recuperado 16 cuerpos en las labores de búsqueda de las víctimas del naufragio de un patera y se han avistado otros tres. El de ayer es uno de los naufragios más graves ocurridos en la última década en aguas canarias."*

O algo que todos recordamos [ambientum.com]:

> *"El pasado 19 de noviembre el Prestige, un petrolero procedente de San Petesburgo que transportaba una carga de 77.000 toneladas de fuel ruso, se partió en dos en las costas gallegas."*

Las cifras son escalofriantes y los medios para prevenirlos, aunque no son pocos, necesitan ser ampliados.

Salvamento marítimo español trabaja con aproximadamente 1500 profesionales, teniendo casi 8.000 km de costa, con partes especialmente complicadas como puedan ser los archipiélagos, pequeñas islas cercanas a la costa, deltas y zonas de difíciles corrientes marítimas (principal ejemplo el estrecho de Gibraltar).

Así mismo, la flota española está siendo mejorada y ampliada durante los últimos años, con una destinación de fondos de más de 1.000 millones de euros. Sin embargo, a pesar de estos medios, la flota española no llega a 100 operativos y 12 bases estratégicas.

¿ Cómo puede este proyecto ayudar?

Trabajamos en la pronta y eficaz actuación tras haberse producido la catástrofe. Principalmente: no necesitando personal encargándose del direccionamiento y control del barco se suman esfuerzos para otras necesidades y los algoritmos de búsqueda realizan de manera rápida y automática el cálculo optimo de localización.

## 2.2 - Agentes y vehículos no tripulados

### 2.2.1 - Concepto de agente

El término de agente software es, como bien se señala en casi toda la biografía consultada, un concepto de difícil definición. El término agente, según la R.A.E, se explica como "persona o cosa que produce un efecto" o "persona que obra en poder de otra". Básicamente todas las definiciones presentadas por expertos en el tema, coinciden en que un agente es una entidad software que presenta un comportamiento autónomo; las discrepancias

surgen en el ámbito de la comunicación de éstos entre sí y con el entorno así como de las capacidades que obligatoriamente deben tener para poder ser considerados agentes.

Los agentes nacieron de la investigación en Inteligencia Artificial, y en concreto en el ámbito de la Inteligencia Artificial Distribuida. En un principio la IA hablaba de los agentes como programas especiales cuya naturaleza y construcción no se llegaba a detallar, pero desde hace ya años, se ha escrito mucho acerca de este tema explicando qué se puede entender por un agente y qué no [Wooldridge 02] [Gómez-Sanz, J.Pavon y F.Garijo 02].

Hay que entender un agente software como una entidad "independiente" que desarrolla su actividad en un entorno más o menos concreto pero no por ello estático; como un ente con capacidad de decisión y de adaptación al entorno y que, como será en el caso concreto de una implementación de este proyecto , puede ser la interfaz entre un sistema físico y el mundo exterior. En nuestro caso entenderíamos el agente como el software que controla la actuación del barco no tripulado. Podemos ver en la siguiente imagen [Figura 2.1]la arquitectura típica de una plataforma de agentes y de su relación con otra. Como queda patente, esta es exclusivamente mediante mensajes.

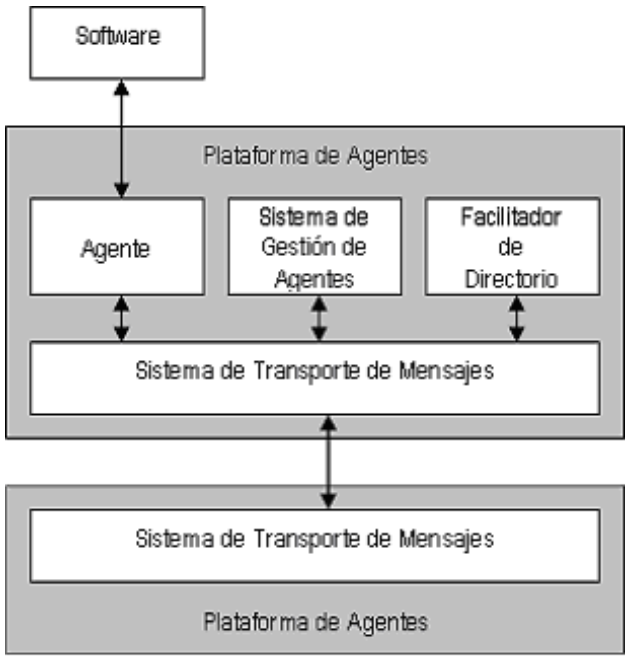



Figura 2.1: Arquitectura de Agentes

La diferencia general con un objeto típico de la POO radica en que la actuación de un

agente no es invocada explícitamente; es decir, así como invocamos un método concreto de un objeto con los parámetros necesarios, en el caso de los agentes el comportamiento no es igual. La manera de comunicarse con un agente y de solicitarle acciones es mediante mensajes o "peticiones"; en base a ellas y exhibiendo más o menos inteligencia, actuará según haya sido programado. Por tanto es poco útil desarrollar un sistema de agentes si mediante los mensajes se mandan órdenes exactas a los agentes sin que estos puedan "decidir" nada.

Un sistema multi-agente (SMA) es un conjunto de agentes que desarrollan su actuación en un entorno común, y según a la definición a la que nos atengamos veremos en un agente la necesidad de comunicación y negociación con otros agentes o la de coordinación distribuida, aprendizaje, movilidad o adaptación al entorno.

Una de las definiciones más citadas y aceptadas es la de Wooldridge y Jennings:

> *"Un agente es un sistema informático situado en un entorno y que es capaz de realizar acciones de forma autónoma para conseguir sus objetivos de diseño".*

Hay que prestar atención a que si bien se les ve siempre caracterizados por la autonomía, en estas aproximaciones todavía no podemos ver la diferencia con un sistema distribuido convencional. Sobre esto, Francisco Garijo, reconocido investigador en la materia, dijo las siguientes palabras:

> *"La incorporación de tecnología de agentes es compatible con sistemas actuales, los complementa y los enriquece. No es una tecnología milagro, se nutre de las mejores prácticas de las distintas corrientes de la ingeniería de sistemas distribuidos incluyendo ingeniería del software, ingeniería del conocimiento, inteligencia artificial y telecomunicaciones."*

Casi todo investigador involucrado en está tecnología coincide con el resto en una serie de elementos comunes que especifican los requisitos de comportamiento y arquitectura de un agente; veamos algunas de las características más atribuidas a los agentes:

•Disposición de realizar su tarea manera autónoma

•Comunicación e intervención asíncrona con otros agentes y en su entorno.

•Capacidad para modificar su conducta o comportamiento en base a lo aprendido

•Aprendizaje adaptativo y capacidad de inferencia [Figura 2.2]

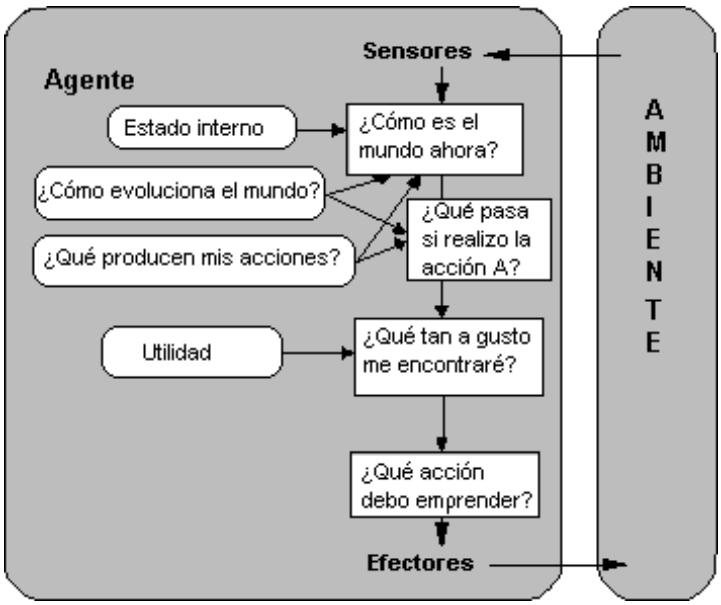



Figura 2.2: Relación de un Agente con su entorno

Estas son otras características muy utilizadas en las explicaciones de teoría de agentes. Aunque no todos coinciden en la necesidad de inteligencia, sí están de acuerdo en que si existe, lo deseable es que sea adaptativa; es decir, que el agente sea capaz de mejorar su funcionamiento según vaya avanzando su proceso de relación con el entorno. En el sector de la Inteligencia Artificial el concepto de agente se toma como propio para el desarrollo de sistemas inteligentes, por ejemplo en la aplicación de bases de conocimiento utilizadas para la interacción con humanos en herramientas basadas en perfiles adaptativos. Una aplicación de esto la podemos ver en el crawler que monitoriza las búsquedas realizadas en la mayoría de los portales de búsqueda.

•Lenguaje natural

Con la revolución de la web surgida en los últimos años y gracias a la evolución en la implementación de complejos sistemas multiagente que se relacionan directamente con usuarios humanos, muchos investigadores asumen una interfaz de lenguaje natural, siguiendo una corriente iniciada años atrás por Shoham presentada en su libro Agent-oriented programming. *Artificial Intelligence en* 1993.

### 2.2.2 - Modelos y arquitecturas de agente

La arquitectura determina los mecanismos que utiliza un agente para reaccionar a los estímulos, comunicarse, etc. Un hecho evidente, en este sentido, es que existe hoy en día una amplia diversidad de propuestas, casi tantas como equipos de investigación centrados en este tema [Wooldridge & Jennings 1994].

Uno de los aspectos básicos que diferencia una arquitectura de otra es el método de descomposición del trabajo en tareas particulares. En este sentido hay que decir que la planificación es un área muy frecuentemente ligada al mundo de la agencia. Está área se centra en el estudio de mecanismos que permitan organizar la ejecución de acciones, y un agente no es más que un sistema que ejecuta acciones en un entorno determinado [Ana Mas 2005]

Una clasificación de tipos de agentes según sus características individuales puede ser la que los clasifica en agentes reactivos y agentes cognitivos.

•Un agente reactivo es un agente de bajo nivel, que no dispone de un protocolo ni de un lenguaje de comunicación y cuya única capacidad es responder a estímulos. Su modelo computacional está basado en la recepción de eventos externos y son en base a lo que actúan. Realiza acciones según el estado en que se encuentre y el evento que reciba, pero no realiza procesos de razonamiento. No son individualmente inteligentes, sino globalmente inteligentes. Bajo este marco se ha desarrollado numerosas teorías de

cooperación y comunicación que permiten el desempeño de estas acciones.

- Un agente cognitivo es capaz de realizar acciones complejas y es individualmente inteligente. Puede considersarse en sí mismo como un sistema experto con capacidad de razonamiento sobre una base de conocimiento inferida adaptativamente por él mismo. Puede comunicarse directamente en un SMA con los demás agentes y llegar a un acuerdo con ellos para tomar alguna decisión.

Naturalmente los SMA reales no se ciñen de manera habitual a estas definiciones, surgiendo por ello los agentes híbridos, que siendo agentes cognitivos, se les suele dotar en ocasiones de capacidades de reacción.

Se pueden clasificar atendiendo a otras perspectivas de observación, como puede ser el tipo de entorno en que funcionan o en el modo de interacción con éste; pero con la clasificación anterior conocemos ya lo suficiente para entender la aplicación en este proyecto.

### 2.2.3 - ¿Cómo se comporta un agente?

En el siguiente gráfico [Figura 2.3], extraido de un tutorial de una universidad de Méjico [Herrera. M], podemos ver el ciclo de vida de un agente:

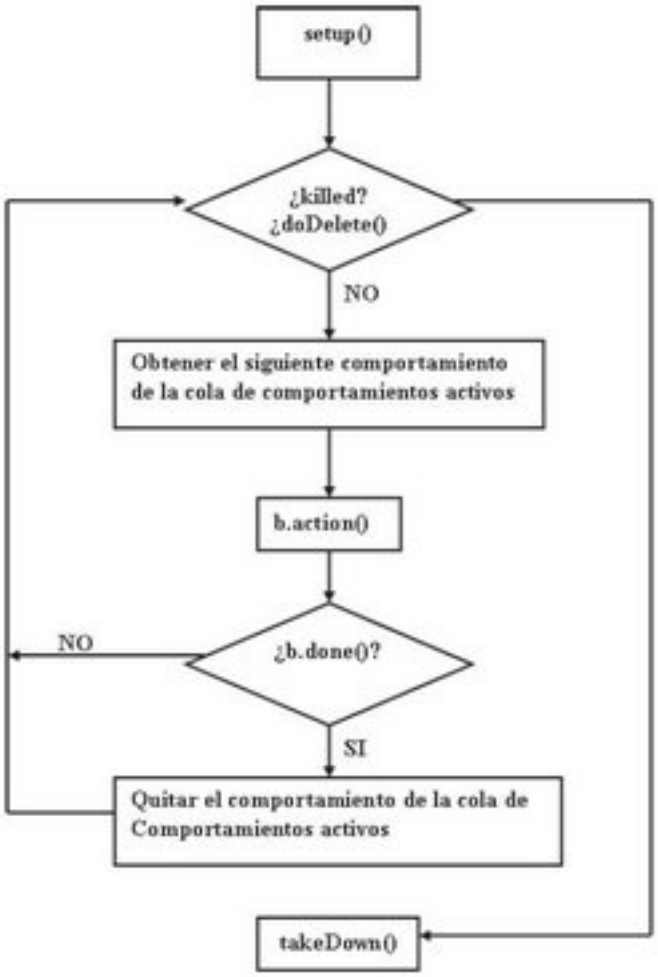



Figura 2.3: Ciclo de vida de un Agente

Como podemos ver, el primer método en ejecutarse tras iniciar y activar el agente es el arriba comentado setup(). Mientras el agente no muera por sí mismo y nadie decida eliminarlo, éste mirará en su cola de comportamientos el siguiente comportamiento "b" a realizar. Ejecutará con él una acción atómica, preguntará si ha terminado, y volverá a mirar su cola (que puede haber cambiado desde la vez anterior). Cuando el comportamiento mencionado acabe, se eliminará de la cola.

A menudo la cola de comportamientos contiene un sólo comportamiento que se dedica a escuchar por si llega algún mensaje, y en función de esto, ejecutará una serie de acciones u otras. Se quedará siempre escuchando por si llega un mensaje con un contenido al que tenga que dar prioridad.

### 2.2.4 - Ejemplos de sistemas que integran agentes

Veamos unos cuantos ejemplos de proyectos que utilizan tecnología de agentes [Computación ubicua y agentes]:

•Proyecto ACES: Agentes para la gestión distribuida de tráfico aéreo. Es un proyecto desarrollado por la NASA para la simulación y modelado del espacio aéreo con el objetivo de validar conceptos del vuelo libre. ACES utiliza un arquitectura basada en el modelo de agentes y un entorno de simulación distribuida. Los agentes representan a los participantes en el tráfico aéreo: pilotos, gestión del tráfico aéreo y control de operaciones. Estos agentes realizan las operaciones típicas en el espacio aéreo. Podemos ver su arquitectura en la siguiente imagen [Figura 2.4].

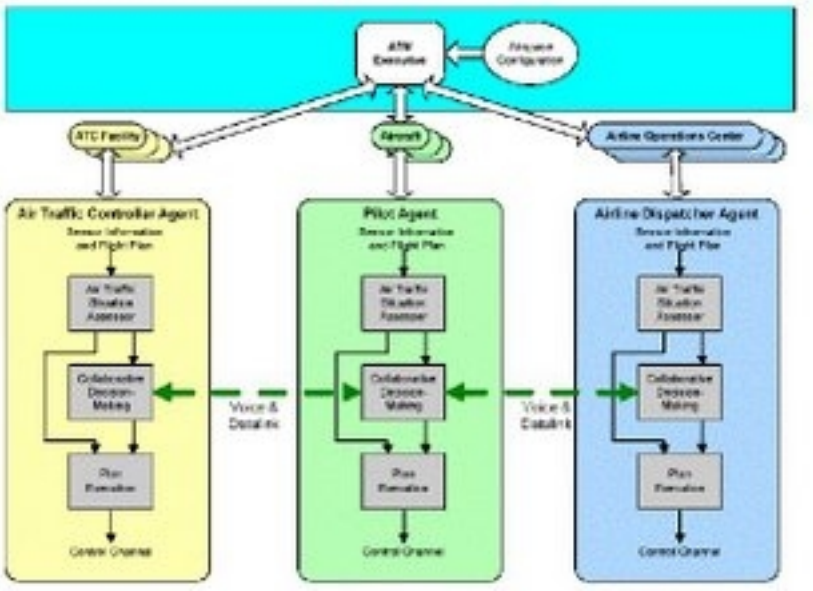



Figura 2.4: Arquitectura del proyecto ACES

•RETSINA: Sistema de detección de minas. Es un sistema desarrollado por el grupo de agentes inteligentes de la Universidad de Carnegie Mellon. Consiste en un sistema de eliminación de minas robótico basado en una aplicación multiagente. Cada robot planifica su estrategia en base a suposiciones sobre lo que están realizando sus compañeros y sobre datos sensoriales. La arquitectura está compuesta por 4 tipos de agentes básicos: Agentes Interfaz, Agentes Tarea, Agentes Información y Agentes Intermediarios como se puede observar en la siguiente imagen [Figura 2.5]


Figura 2.5: Arquitectura del proyecto RETSINA

Control de vehiculos no tripulados: La Universidad de Michigan ha desarrollado el proyecto de vehículos de tierra sin tripulación (UGV, Unmanned Ground Vehicles). Los UGV's se tratan como agentes semiautónomos que son capaces de sentir el entorno y adoptar planes del repertorio que más se ajuste a alcanzar sus objetivos en unas circunstancias dadas. La solución propuesta es un sistema de ejecución y planificación de misiones integradas que está distribuido entre diversos agentes que expresan los objetivos, planes y creencias.

Aquí podemos ver su arquitectura [Figura 2.6].

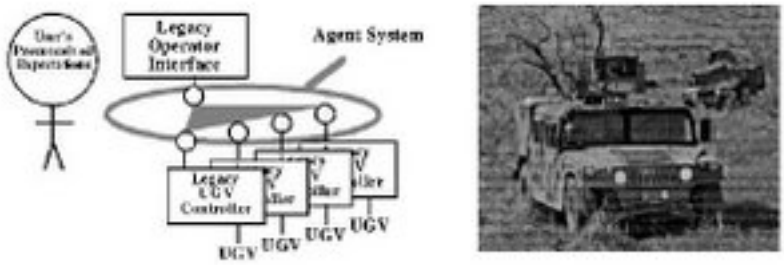



Figura 2.6: Arquitectura del proyecto UGV de U.Michigan

## CONCLUSIONES

Siguiendo la tendencia en el mercado sobre el uso de API´s y heramientas basadas en las especificaciones de la FIPA, concluimos que el marco general de nuestro proyecto en el ámbito de agentes debería ser creado mediante JADE. Se han barajado otras plataformas, como Cougaar, también implementada en Java pero que a priori parece apropiada para agentes más complicados y al parecer su uso está menos extendido. Otra plataforma gratuita para uso no comercial que hemos estudiado ha sido Grasshoper, que soporta el estándar MASIF además del FIPA. Está plataforma cuenta con numerosas GUI´s de desarrollo y su implantación en el mercado real es bastante extensa, sin embargo y dado que el alcance de nuestro proyecto no requiere una compleja trama de agentes en diversos hosts, hemos optado por JADE para facilitar una futura unión con el proyecto Seagull (tambien en JADE), del que hemos hablado anteriormente.

### 2.2.5 - Vehículos no tripulados

La Automática es una disciplina cada vez más solicitada en todos los campos de la Industria moderna.

Teniendo en cuenta que la Automática es una disciplina horizontal (con aplicaciones en diversos campos) se hace necesario crear mecanismos que permitan la divulgación de las investigaciones y las técnicas propias que son aplicables en cada uno de estos campos. En concreto esta necesidad se da en la industria naval. Muchas veces ingenieros navales, en

empresas de desarrollo tecnológico, se están enfrentando a problemas (que en alguna manera les puede desbordar) y que son propios de las disciplinas de la automática (estos problemas van desde el control y modelado a la integración multisensorial, etc.).

La automática se encuentra presente en multitud de sistemas en un buque: guiado y control de rumbo, control de diferentes servos (para poleas de una grúa, hélices, etc), sistemas de estabilización (compensación de movimientos no deseados del buque, estabilización de plataformas, etc), control de grúas de precisión que permiten el traspaso de mercancías de un barco a otro con presencia de oleaje, etc. También en el proceso de diseño y fabricación se encuentra presente la automática: programas de optimización, simulación, etc.

Pero además de armadores y astilleros, los suministradores de equipos tecnológicos también necesitan de la automática, tanto en la fase de producción de los sistemas que fabrican como para incorporarlo en esos mismos sistemas y dotarlos de ese valor de alta tecnología que necesitan para hacerlos competitivos.

En la actualidad empresas como Siemens, Asea Beown Boveri, Norcontrol entre otras, ocupan en Europa puestos muy altos en el ranking de fabricantes, integradores de sistemas y exportadores de material de automatización naval.

Los países fabricantes de instrumentación de captación de información y actuadores son candidatos muy competitivos en la industria naval sin excluir cualquier otra industria, porque lo que vale para los barcos vale para cualquier proceso industrial. En España existe aún una gran necesidad de entrar de lleno en el campo del desarrollo de sensores, detectores y actuadores.

Lo que a nosotros nos ocupa son los vehículos no tripulados. Esta rama de la Automática está más desarrollada en aviones. Denominados UAV (*Unmanned Aerial Vehicle)*, estos vehículos se posicionan por medio de señales GPS, navegan a través de sistemas GIS (Geographic Information System o SIG en Castellano)… La CPU que lleva a bordo se encarga de *pilotar* sin que sea necesario disponer de un humano a bordo.

El UAV es simplemente un avión sin piloto que puede ser controlado remotamente o volar autónomamente con planes de vuelo preprogramados u otros sistemas de automatización

más complejos. Se puede emplear para diferentes misiones militares, incluidos el ataque y la exploración. También hay una creciente tendencia en su empleo para necesidades civiles, como la lucha contra el fuego, inundaciones u otros desastres naturales y operaciones policiales como la vigilancia y observación de escenarios de delitos, alteraciones del orden público y control de rutas.

En nuestro proyecto desarrollaremos una jerarquía de agentes para el control de los vehículos no tripulados, en nuestro caso marinos.

Para consultar un estudio sobre UAVs y trayectorias óptimas visitar: http://www.mssanz.org.au/modsim05/papers/marsh.pdf

## 2.3 - Trayectorias

Un Sistema de Información Geográfica (SIG o GIS, en su acrónimo inglés) es una integración organizada de *hardware*, *software* y datos geográficos diseñada para capturar, almacenar, manipular, analizar y desplegar en todas sus formas la información geográficamente referenciada con el fin de resolver problemas complejos de planificación y gestión.

Un SIG destinado al cálculo de rutas óptimas para servicios de emergencias es capaz de determinar el camino más corto entre dos puntos teniendo en cuenta impedancias como direcciones de circulación, giros prohibidos o evitando determinadas áreas impracticables. Un SIG para la gerencia de una red de abastecimiento de aguas sería capaz de determinar, por ejemplo, a cuantos abonados afectaría el corte del servicio en un determinado punto de la red.

La información geográfica puede ser consultada, transferida, transformada, superpuesta, procesada y mostrada utilizando numerosas aplicaciones de software. Dentro de la industria empresas comerciales como ESRI, Intergraph, Mapinfo, Autodesk o SmallWorld ofrecen un completo conjunto de aplicaciones. Los gobiernos suelen optar por modificaciones *ad-hoc* de programas SIG, productos de código abierto o software especializado que responda a una necesidad bien definida.

El manejo de este tipo de sistemas es llevado a cabo generalmente por profesionales de diversos campos del conocimiento con experiencia en Sistemas de Información Geográfica (cartografía, geografía, topografía, etc.), ya que el uso de estas herramientas requiere un aprendizaje previo que necesita conocer las bases metodológicas sobre las que se fundamentan. Aunque existen herramientas gratuitas para ver información geográfica, el acceso del público en general a los geodatos está dominado por los recursos en línea, como Google Earth y otros basados en tecnología web mapping.

Originalmente hasta finales de los 90, cuando los datos del SIG se localizaban principalmente en grandes ordenadores y se utilizan para mantener registros internos, el software era un producto independiente. Sin embargo con el cada vez mayor acceso a Internet y a la demanda de datos geográficos distribuidos, el software SIG ha cambiado gradualmente su perspectiva hacia la distribución de datos a través de redes. Los SIG que en la actualidad se comercializan son combinaciones de varias aplicaciones interoperables y APIs.

La razón fundamental para utilizar un SIG es la gestión de información espacial. El sistema permite separar la información en diferentes capas temáticas y las almacena independientemente, permitiendo trabajar con ellas de manera rápida y sencilla, y facilitando al profesional la posibilidad de relacionar la información existente a través de la topología de los objetos, con el fin de generar otra nueva que no podríamos obtener de otra forma.

Las principales cuestiones que puede resolver un Sistema de Información Geográfica, ordenadas de menor a mayor complejidad, son:

1. **Localización**: preguntar por las características de un lugar concreto.
2. **Condición**: el cumplimiento o no de unas condiciones impuestas al sistema.
3. **Tendencia**: comparación entre situaciones temporales o espaciales distintas de alguna característica.
4. **Rutas**: cálculo de rutas óptimas entre dos o más puntos.
5. **Pautas**: detección de pautas espaciales.
6. **Modelos**: generación de modelos a partir de fenómenos o actuaciones simuladas.

Por ser tan versátiles, los Sistemas de Información Geográfica, su campo de aplicación es muy amplio, pudiendo utilizarse en la mayoría de las actividades con un componente

espacial. La profunda revolución que han provocado las nuevas tecnologías ha incidido de manera decisiva en su evolución.

Para más información: http://es.wikipedia.org/wiki/GIS

En este proyecto se trabaja con Google Earth 5 que es un programa informático similar a un Sistema de Información Geográfica, creado por la empresa Keyhole Inc., que permite visualizar imágenes en 3D del planeta, combinando imágenes de satélite, mapas y el motor de búsqueda de Google que permite ver imágenes a escala de un lugar específico del planeta.

## 2.4 - Herramientas: Jade

En este proyecto, tras evaluar distintas herramientas para el desarrollo de arquitecturas de agentes, hemos optado por el uso de JADE por los motivos explicados anteriormente.

JADE (Java Agent DEvelopment Framework) es una plataforma desarrollada íntegramente en Java para la creación de sistemas multi-agente. Además de proporcionar un API para la creación de agentes y elementos relacionados, pone a nuestra disposición una interfaz gráfica y una serie de herramientas para el control y la depuración de nuestro sistema durante el desarrollo del mismo.

Una de las principales características de la plataforma es que los desarrollos se encontrarán dentro del estándar FIPA, especificado por FIPA; una organizacción formada por empresas, entidades y particulares cuyo objetivo es crear estándares para sistemas multi-agente y que en 2005 entró a formar parte de la IEEE.

Entre otras cosas ofrece la creación de diversas plataformas continentes de agentes, y las herramientas necesarias para migraciones entre las mismas. El intercambio de mensajes entre agentes, así como las performativas empleadas se corresponderán con lo especificado con este estándar.

La documentación de JADE es numerosa y está bien estructurada, sin embargo la información para la creación de un SMA utilizando las herramientas

proporcionadas está demasiados dispersa entre la gran cantidad de documentos ofrecidos. Por ello se ve la necesidad de recurrir a la consulta de los diversos tutoriales que se pueden encontrar en Internet para poder iniciarse en el desarrollo con JADE.

Cabe destacar que JADE es Software Libre y que se distribuye bajo los términos de la licencia LGPL versión 2, lo que lo ha convertido en el estándar de facto para el desarrollo de sistemas con arquitectura de agentes.

Se puede descargar el paquete del proyecto en diferentes combinaciones de fuentes, binarios, ejemplos y documentación en la web oficial: http://sharon.cselt.it/projects/jade/.

La plataforma incluye en sus librerías:

•Todo lo necesario para la creación básica de agentes

•Programación de comportamientos para asociar a los agentes (behaviours)

•Paquetes de utilidades para el desarrollo de mensajes entre-agentes según el estándar FIPA ACL.

•Numerosas clases auxiliares para facilitar tanto el desarrollo y la depuración de un SMA como para su integracion directa. (Por ejemplo agentes de comunicaciones, gestores de plataformas...).

## INICIACIÓN A JADE

Será la clase Boot.java la que nos permita iniciar la interaz de JADE; para ello es preciso ejecutar desde el terminal el comando java jade.Boot -gui (añadiendo al classpath claro, la ruta con los binarios del proyecto JADE).

Esta es la pantalla que nos aparecerá [Figura 2.7]:

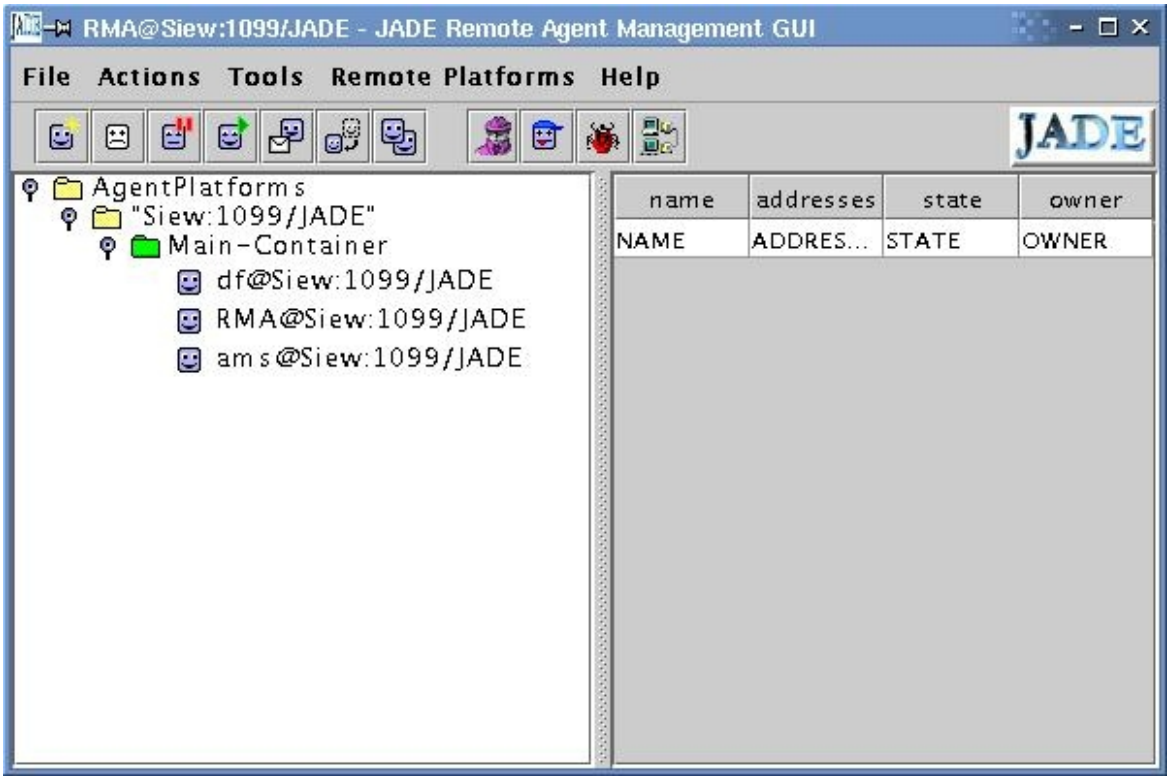



Figura 2.7: GUI del Agente RMA de JADE

Para poder entender rápidamente el funcionamiento de la herramienta, serán necesarios los siguientes conceptos:

- Las plataformas de agentes (AgentPlatforms) se corresponden con máquinas y constan de contenedores cuya función es almacenar agentes, no necesariamente en el mismo ordenador.. Los agentes utilizan el protocolo RMI para comunicarse entre varios contenedores.

- La imagen anterior muestra la interfaz del Remote Monitoring Agent (RMA) que aparece cuando se utiliza la opción -gui. Además de a sí mismo, el RMA muestra la presencia de otros dos agentes en el Main Container.

- Los agentes deben tener nombres únicos. Un nombre se compone de un alias o nickname y una dirección separados por el signo @. Por ejemplo RMA@Siew:1099/JADE es un agente con nickname RMA en la dirección Siew:1099/JADE (Siew es el nombre de la máquina en la LAN y 1099 el puerto

utilizado por defecto).

Como puede observarse, el agente AMS iniciado con el comando anterior, ha generado otros dos agentes que facilitarán la depuración y la observación del resto de agentes; el RMA, comentado arriba, y el agente df. Este es el Directory Facilitator, que proporciona un servicio de Páginas Amarillas para los agentes conocidos para la plataforma. Los agentes se registrarán en él y ofrecerán una serie de servicios que el resto de ellos podrán consultar y utilizar.

Desde la interfaz (y obviamente desde la API en código JAVA), podrán instanciarse (crearse) además de los mencionados anteriormente, agentes ya implementados en JADE; son los siguientes:

•Agente Dummy:

> Se trata de un agente enviador de mensajes que tiene asociada una interfaz gráfica que nos permite indicar los distintos campos del mensaje a enviar, así como visualizar los mensajes enviados y recibidos [Figura 2.8].

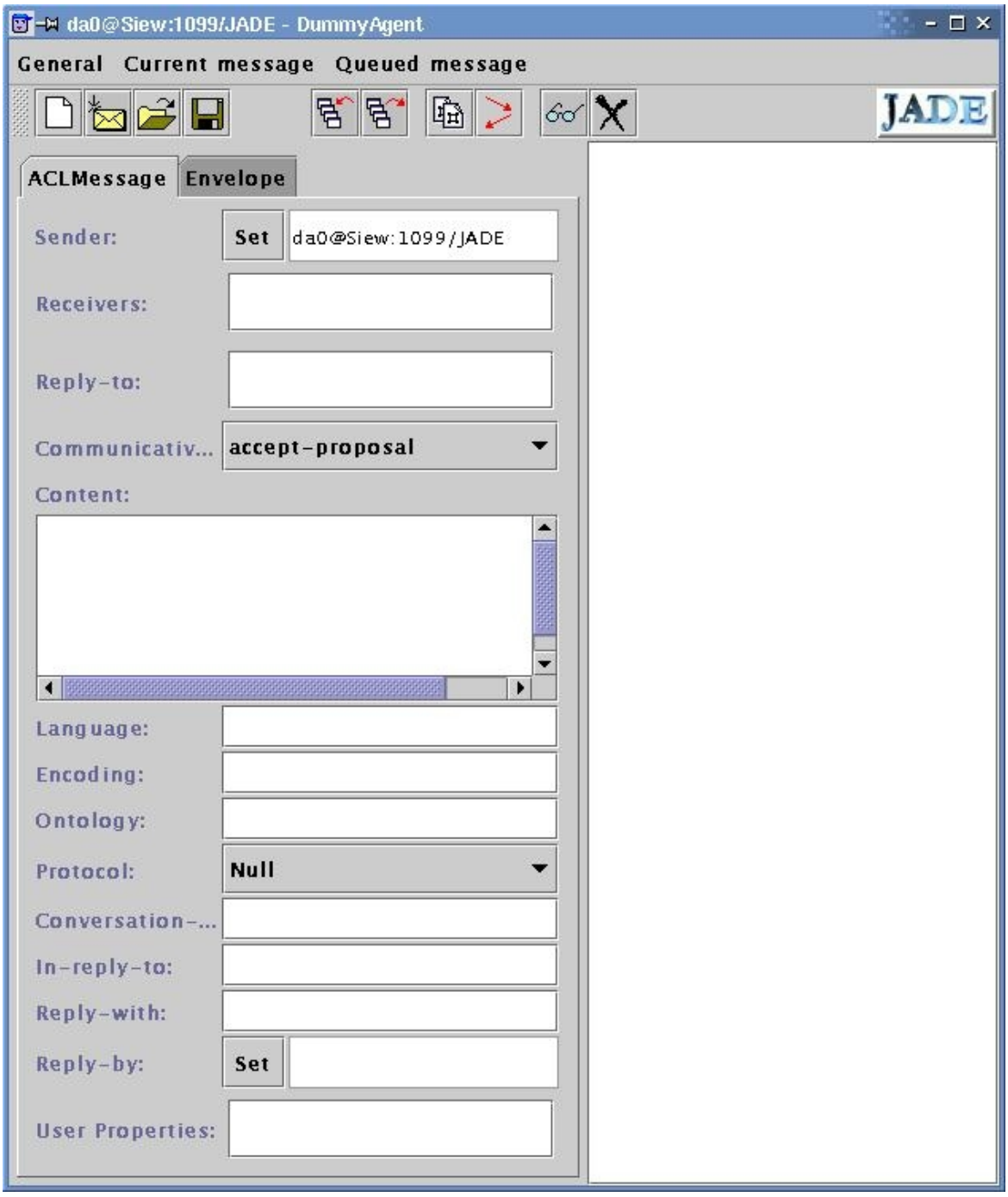



Figura 2.8: GUI del Agente Dummy de JADE

•Agente Sniffer:

Este agente nos permitirá observar la interacción y el cambio de mensajes entre los agentes que le indiquemos y que estén en el momento de escucha

en ejecución. En la siguiente imagen [Figura 2.9] podemos ver al agente ping cómo se comunica con el sniffer (agente da0).

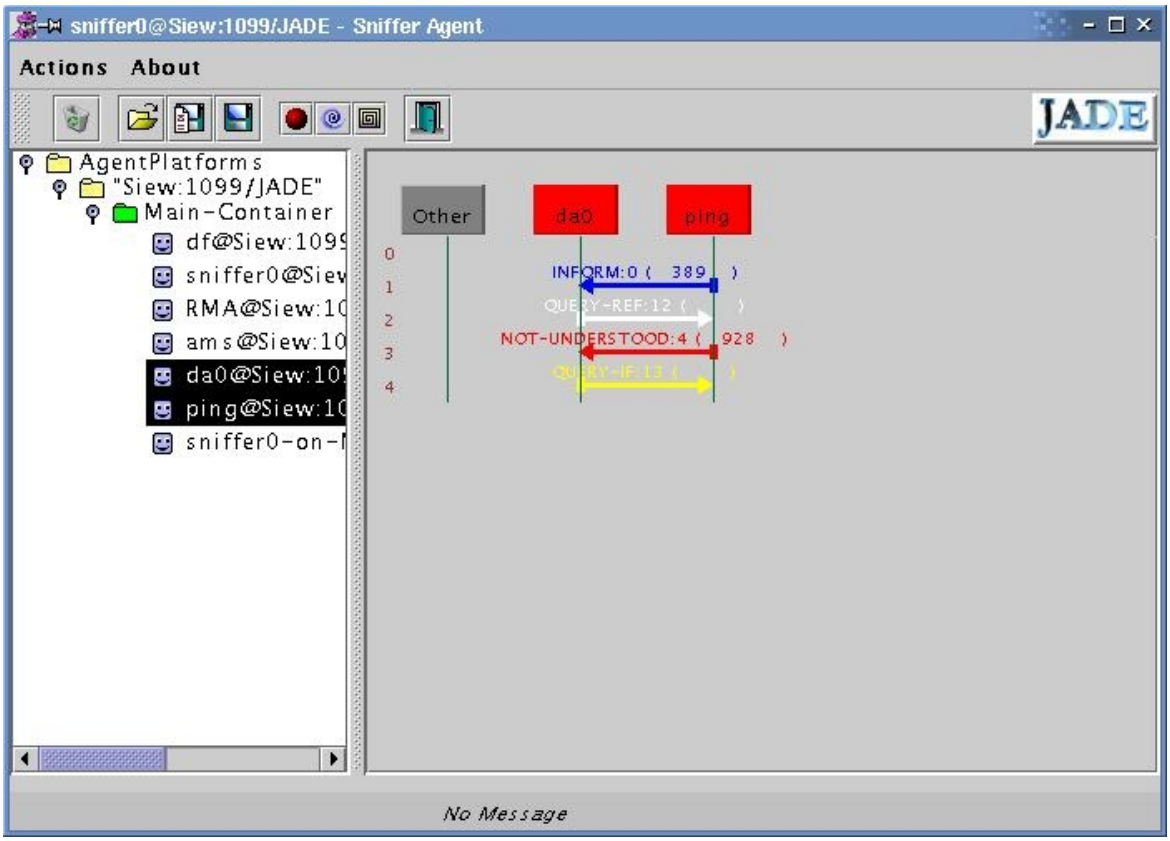



Figura 2.9: GUI del Agente Sniffer de JADE

Para invocar a un agente podremos hacerlo desde la interfaz del RMA, o desde la consola tecleando lo siguiente:

*java jade.Boot nombreAgenteEjemplo:ClaseAgenteEjemplo*

Cada agente de nuestro sistema será una instancia de una clase que sea subclase de jade.core.Agent. Esta nueva clase tendrá básicamente un método setup, que se ejecutará al iniciarse el agente y que contendrá código de inicialización (como el registro en el df y la asignación de nombre y oferta de servicios). Además de este método setup se dispondrá de una clase interna a la clase del agente por cada uno de los comportamientos asociados al agente (behaviours). Estos comportamientos implementarán tanto el comportamiento del agente como todo lo necesario para sus comunicaciones.

# Capítulo 3 -

# Descripción del sistema

A continuación describiremos de forma general una visión global del proyecto Dolphin enunciando características, funcionalidades, estructura y uso para que el lector se haga una idea general.

Los proyectos realizados en cursos anteriores trataron el tema de vehículos aéreos no tripulados (conocidos como UAVs) que se basaban en el recorrido óptimo de áreas donde a través de un sistema de reconocimiento de imágenes eran capaces de identificar puntos de catástrofe o naufragios. Una vez detectados estos puntos es el momento para los vehículos acuáticos de entrar en escena.

Desde el instante en que una situación como las anteriormente mencionadas es reconocida desde el aire hasta que los barcos de salvamento (en el caso de naufragios) llegan al punto de conflicto no es descabellado pensar en un posible radio de desplazamiento debido a posibles corrientes marítimas, tormentas o diversos fenómenos atmosféricos. Es por esto que un vehículo marino no tripulado debe tener también que reconocer imágenes de satélite

en tiempo real para seguir la búsqueda de su objetivo.

Desde el punto donde se encuentra el barco hasta el punto objetivo es necesario por cuestiones de tiempo la búsqueda de la ruta más corta. Por esto, tomamos la decisión de basar nuestro proyecto en imágenes "reales" obtenidas a través de Google Earth 5. En esta nueva versión, el programa de Google mejoró notablemente todo lo que se refiere a zonas de mares y océanos. Entre esto, la fidelidad a la realidad (uso de mapas reales) y la posibilidad de contar con una versión gratuita nos hizo decantarnos rápidamente por su posible uso.

El usuario que está dispuesto a usar Dolphin no precisa un control de todas las funcionalidades de Google Earth. Sólo necesita un par de conceptos sobre cómo guardar la imagen de la zona en la que quiere trabajar. Para guardar una zona de la Tierra en la que se quieren probar diversas rutas se pensó la idea de que el usuario seleccionara dos puntos en la interfaz de Google Earth y pasar esta información a Dolphin. Para aumentar la sencillez de este paso, decidimos trabajar con un único punto de referencia de la zona en la que navegar. Se guardará esta zona en un archivo en disco junto con su imagen JPEG y será el propio Dolphin quien le dé la opción de seleccionar los puntos entre los que navegar.

Los archivos que debe el usuario guardar previamente son la imagen en la que trabajaremos y un archivo *.kml* que servirá de referencia para trabajar con coordenadas reales.

Una vez ejecutándose Dolphin, lo primero que se debe hacer es seleccionar el mapa en el que se va a trabajar, así como el número de barcos que navegarán en esta zona. Disponemos de una *preview* para la mejor identificación de las imágenes.

Una vez seleccionado el mapa, la siguiente acción a realizar es la de señalar los puntos entre los que se quiere navegar. Por supuesto, Dolphin da la opción de elegir dichos puntos, pero a la hora de la integración con proyectos de reconocimiento aéreo, el punto de origen será la situación actual del barco, y el destino la zona dónde se ha detectado el posible punto de catástrofe.

Para la selección de dichos puntos se optó por la selección directa sobre el mapa, sobre la fotografía de Google Earth que previamente el usuario había guardado. A través de dos iconos se podrá indicar *origin* y *target* (origen y objetivo), así como cualquier cantidad de

puntos intermedios mediante el botón *goal*. Por supuesto se podrá rectificar cualquiera de estos puntos y también se podrán indicar distintas rutas para distintos agentes inteligentes con el fin de trabajar simultáneamente y ganar en tiempo, algo fundamental en este asunto.

El siguiente paso es el cálculo de la ruta a seguir para cada uno de los vehículos que están trabajando. Los archivos que el usuario debe obtener y transferir a Dolphin no dan información sobre las zonas transitables a través de barco. Nos hemos basado en software de reconocimiento de imágenes para diferenciar las zonas navegables de las zonas terrestres. Empleando el algoritmo LVQ (*Learning Vector Quantization*), nuestro proyecto es capaz de discernir tierra de mar y crear un mapa ajeno al usuario con el que trabajará. La ejecución del algoritmo LVQ no es precisamente ligera, por lo que optamos por el uso de archivos binarios para guardar los mapas que ya distinguen tierra de mar. Esta idea es completamente lógica, ya que se prevé que un mismo mapa se utilizará más de una vez y el tiempo ahorrado es realmente alto.

Para calcular la trayectoria óptima nos hemos basado en el algoritmo A* (A estrella). El algoritmo de búsqueda A* se clasifica dentro de los algoritmos de búsqueda en grafos. Presentado por primera vez en 1968 por Peter E. Hart, Nils J. Nilsson y Bertram Raphael, el algoritmo encuentra, siempre y cuando se cumplan unas determinadas condiciones, el camino de menor coste entre un nodo origen y uno objetivo. Las rutas encontradas por este algoritmo posteriormente son optimizadas para mayor realismo de lo que haría un barco, y ya de paso ahorrar algunos puntos innecesarios de la trayectoria.

Esta trayectoria sería la “ideal”. Por supuesto un barco no puede hacer giros de noventa grados en dos metros; aquí es donde entra la relación con el modelo del barco.

Dolphin trabaja con el modelo de un barco concreto. Se nos facilitó en MATLAB – Simulink el comportamiento de dicho vehículo y nosotros migramos el código a JAVA para su posterior simulación. El realismo en la ruta a recorrer se consigue dándole cada uno de los puntos de nuestra *trayectoria óptima* al modelo del barco y así obtener giros del mayor realismo posible. La nueva trayectoria contendrá muchos puntos intermedios, pues era necesario representar curvas.

Esta nueva ruta ya se encuentra en función de coordenadas reales, pues antes

estábamos trabajando con puntos de la imagen (archivo JPEG guardado por el usuario). Esta ruta se incluye a través de una plantilla que hemos creado en un archivo *.kml* con los que trabaja Google Earth. Desde el simulador se podrá observar en cada momento el estado de cada barco y su misión. Mediante un botón situado en esta pantalla de resumen de rutas, podrán cargarse en Google Earth los kml generados y archivados para cada barco; de esta forma podremos ver sobre el globo terráqueo y a una altura idónea, la trayectoria que describiría nuestro agente inteligente.

Como colofón a esta simulación el usuario podrá seguir la trayectoria de manera dinámica simulando que se encuentra sobre un barco. Únicamente deberá pulsar sobre el botón de PLAY y la cámara se colocará al nivel del mar y cubriremos la ruta que Dolphin calculó para este agente.

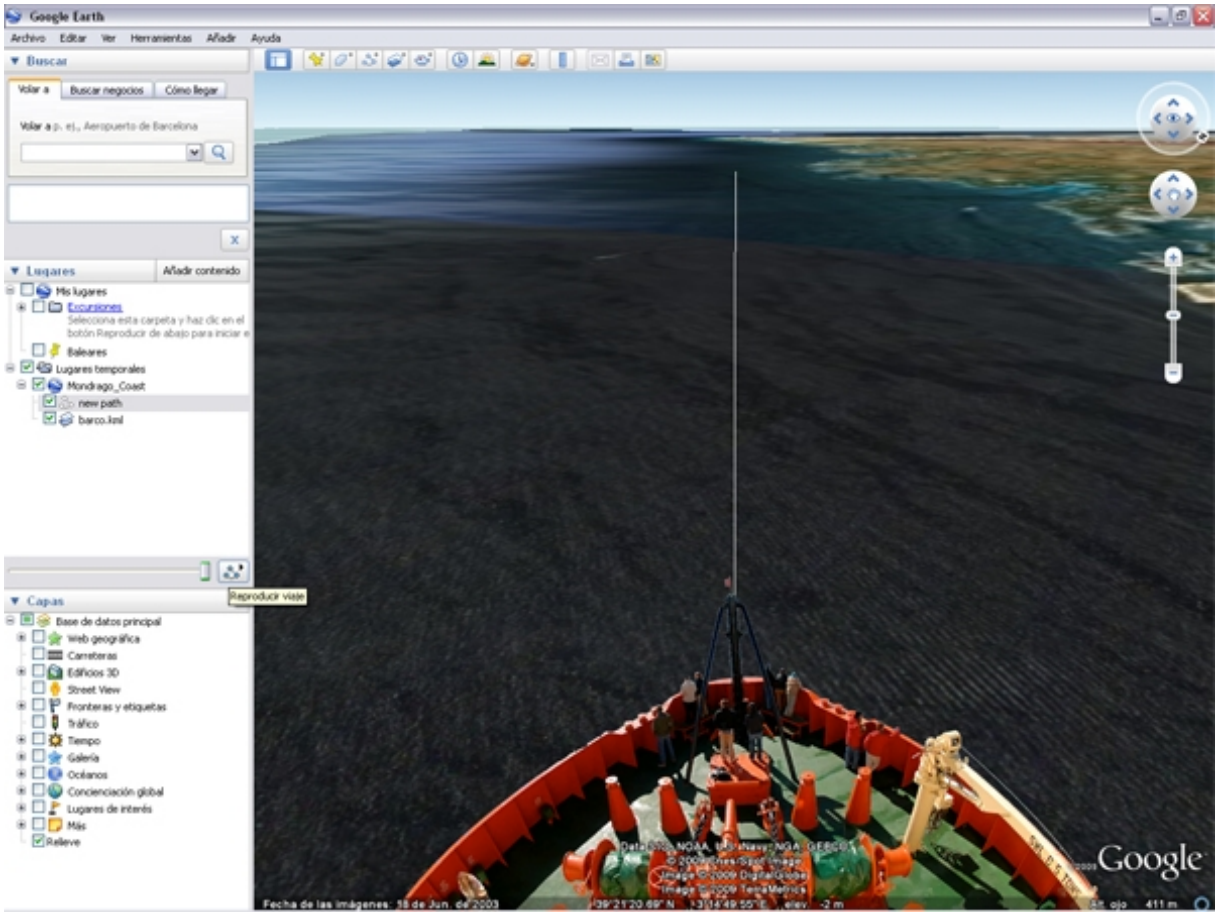



Figura 3.1: Simulación final sobre Google Earth

Todas estas capacidades están englobadas en un marco común que las trata, la arquitectura desarrollada para los agentes inteligentes. La herramienta de simulación implementa una base de comunicaciones de agentes que es capaz de instanciar a los mismos, comunicarse con ellos y en consecuencia darles órdenes. Toda simulación que se haga tendrá que ir asociada a un barco en concreto, el cual dispondrá de una localización, de una tarea y de una acción concreta en cada instante. Desde el momento en que se crea un Agente (un Barco), éste permanecerá en todo momento a la escucha de posibles mensajes de la base a la espera de órdenes pero siempre realizando su tarea actual.

# 3.1 - Modelo del barco

En este apartado se van a explicar las reglas y ecuaciones por las que se rige la simulación de la navegación del barco.

Como se ha mencionado anteriormente, la información de la que partíamos provenía de un proyecto (del profesor Jesús M de la Cruz) en Simulink (aplicación de Matlab) y la documentación adjunta a éste.

Para explicar su funcionamiento vamos a ver el desarrollo de las ecuaciones del movimiento, su implementación en Simulink (con algunas características básicas de este entorno de desarrollo), su migración a lenguaje Java y finalmente veremos un simple estudio de los resultados obtenidos.

## 3.1.1 - Ecuaciones del movimiento

Resumimos este movimiento con los siguientes valores a partir de la información que se nos aportó desde distinta bibliografía [Fossen 94]:

$U$: Vector velocidad del barco

$u$: velocidad de avance (eje longitudinal del barco: popa a proa) (surge)

$v$: velocidad de través (eje transversal)(sway)

$\psi$ : ángulo de guiñada (yaw)

r: velocidad de guiñada (yaw rate)

δ : deflexión del timón, positiva cuando da un giro positivo a r, como se muestra en la figura

Variables que regulan la dinámica del buque, según definen las siguientes ecuaciones:

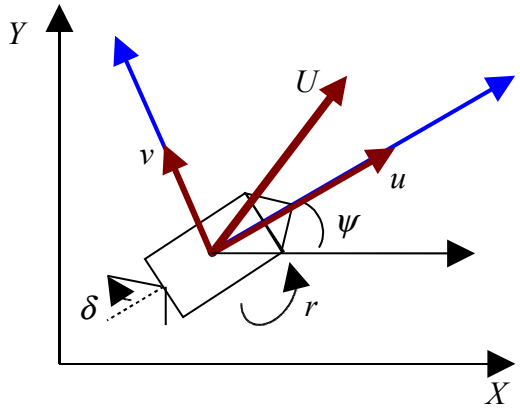
$$T_1 \cdot T_2 \cdot r''(t) + (T_1 + T_2) \cdot r'(t) + r(t) = K(T_3 \cdot \delta'(t) + \delta(t))$$

$$\begin{aligned} \psi'(t) &= r(t) \\ x'(t) &= u(t) \cdot \cos(\psi(t)) - v(t) \cdot sen(\psi(t)) \\ y'(t) &= u(t) \cdot sen(\psi(t)) + v(t) \cdot \cos(\psi(t)) \\ U(t) &= (u^2(t) + v^2(t))^{1/2} \end{aligned}$$

Estas ecuaciones pueden ser muy complicadas, por lo que a veces se utiliza un modelo simplificado, que se representa mediante las siguientes ecuaciones:

$$\begin{aligned} T \cdot r'(t) + r(t) &= K \cdot \delta(t) \\ T &= T_1 + T_2 - T_3 \end{aligned}$$

Para intentar explicarlo mejor, veamos la siguiente imagen, en la que podemos ver las distintas fuerzas que empujan al barco.




Hay que tener también en cuenta información sobre el barco concreto con el que trabajamos ya que sus características influirán en el recorrido. Conocemos dos posibilidades:

| | Mariner (Barco de carga) | Petrolero a plena carga |
|---|---|---|
| Longitud metros | 161 | 350 |
| $U_0$ m/s | 7.7 | 8.1 |
| K $seg^{-1}$ | 0.185 | -0.019 |
| $T_1$ seg | 118.0 | -124.1 |
| $T_2$ seg | 7.8 | 16.4 |
| $T_3$ seg | 18.5 | 46.0 |

## SIMULINK

Esta aplicación es muy utilizada para el modelado de sistemas. Se puede encontrar documentación sobre ella en diversos libros, pero nosotros centramos las consultas en la web oficial del sistema, que coincide con la web oficial de MATLAB, desde la que hemos seleccionado la siguiente información para resumir un poco que aporta esta aplicación:

> *"Simulink es un entorno para la simulación multidominio y el diseño basado en modelos para sistemas dinámicos y embebidos. Presenta un entorno gráfico interactivo y un conjunto personalizable de bibliotecas de bloques que permiten simular, implementar y probar una serie de sistemas variables con el tiempo, incluido comunicaciones, controles y procesamiento de señales, vídeo e imagen.*
>
> *Los productos complementarios extienden el software de Simulink a múltiples dominios de modelaje y ofrecen herramientas para tareas de diseño, implementación, verificación y validación.*
>
> *Simulink está integrado con MATLAB y ofrece acceso*

*inmediato a una amplia gama de herramientas que permiten desarrollar algoritmos, analizar y visualizar simulaciones, crear series de procesado de lotes, personalizar el entorno de modelaje y definir señales, parámetros y datos de pruebas."*

**FUNCIONES PRINCIPALES**

- *Bibliotecas extensas y ampliables de bloques predefinidos*
- *Editor gráfico interactivo para ensamblar y administrar diagramas de bloque intuitivos*
- *Capacidad de gestionar diseños completos segmentando los modelos en jerarquías de componentes de diseño*
- *Model Explorer para navegar, crear, configurar y buscar todas las señales, parámetros, propiedades y código generado asociados con el modelo*
- *Interfaces de programación de aplicaciones que permiten conectar con otros programas de simulación e incorporar código escrito manualmente*
- *Embedded MATLAB™ -Bloques de funciones para implementar los algoritmos de MATLAB en Simulink e implementaciones de sistemas empotrados*
- *Modos de simulación (normal, acelerador y acelerador rápido) para ejecutar simulaciones de forma interpretativa o a velocidades de código C compilado utilizando algoritmos de paso fijo o variable*
- *Depurador y perfilador gráfico para examinar los resultados de simulación y diagnosticar el rendimiento y el comportamiento inesperado del diseño*
- *Acceso completo a MATLAB para analizar y visualizar resultados, personalizar el entorno de modelaje y definir señales, parámetros y datos de prueba*
- *Herramientas de análisis de modelos y diagnosis para garantizar la coherencia de los modelos e identificar errores de modelaje"*

### 3.1.2 - Modelo sobre Simulink

En la siguiente imagen [Figura 3.2], podemos ver el esquema del funcionamiento del barco desarrollado en Simulink.

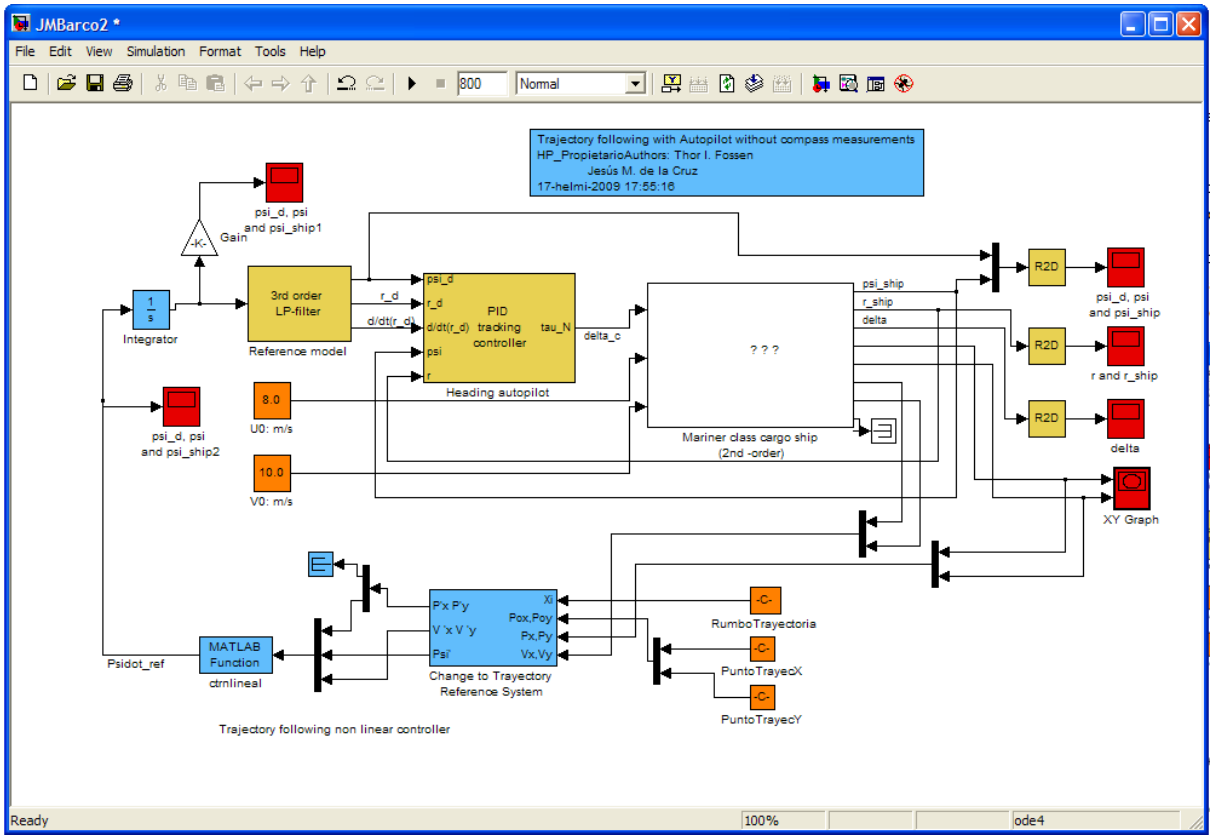



Figura 3.2: Pantalla principal del archivo origen .mdl

Los datos principales que introducirá el usuario (**entradas**) están representados por las cajas naranjas que aparecen en la imagen anterior [Figura 3.2].

Contamos con:

•U0: es la velocidad en m/s del buque en dirección longitudinal (de popa a proa).

•V0: velocidad lateral que puede llevar el buque debido a una corriente de velocidad constante, o a la unión de corriente, viento y oleaje.

Estas velocidades las estamos tomando por defecto a unos valores constantes, pero pueden ser modificados en futuras ampliaciones sin perjudicar en ningún momento al movimiento. Una velocidad lateral muy grande puede suponer que no se alcance el objetivo ya que puede conllevar mucho tiempo en obtener una dirección correcta para el barco.

•Rumbo Trayectoria: ángulo respecto al eje X objetivo ("rumbo de referencia que une el way point de coordenadas Po con una recta de rumbo Xi_r con respecto al eje X geográfico"). Es importante para cualquier modificación saber que este valor se introduce en grados, aunque se trabaja con el como radianes.

•Punto Trayectoria ( X, Y): El punto Po indica el origen del way point.

Las **salidas**: la única que nos interesa mantener siempre accesible es la posición del barco en cada instante. Dato que en este entorno podemos ver representado en la última gráfica a través de los parámetros Px, Py que podemos ver en la parte inferior derecha de la Figura 3.2. con el nombre "*XY Graph*", aunque no nos interesará en la aplicación en Java.

Independiente a ello vamos a basarnos, para comprobar el funcionamiento del sistema en las distintas gráficas generadas. Éstas corresponden a los valores obtenidos en las zonas señaladas por recuadros rojos. Pero eso será tras comentar el funcionamiento de los principales módulos.

**MÓDULO CHANGE_TO_TRAYECTORY**

Cambia el sistema de referencia de la información. Está representado en la imagen inferior [Figura 3.3], que resume su estructura.

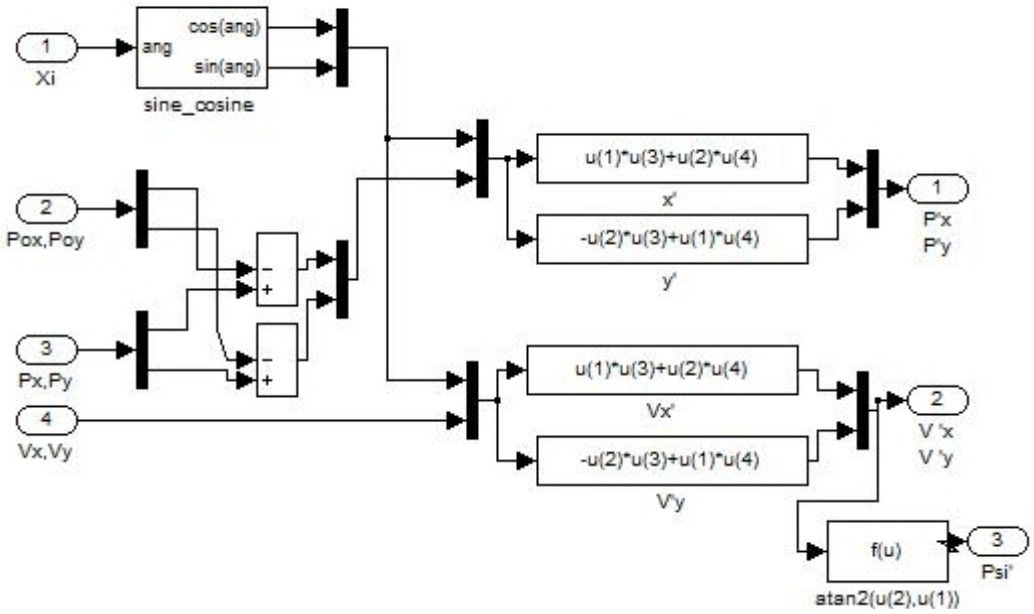



Figura 3.3: Módulo de cambio del sistema de referencia

**MÓDULO CTRNLINEAL**

El algoritmo calcula la velocidad angular del ángulo de rumbo que debe seguir el barco para acercarse a la recta que une el *way point* actual con el siguiente. Tras esto, la velocidad angular del rumbo de referencia se integra para generar el ángulo de rumbo que es la entrada al auto-piloto.

**MÓDULO REFERENCE MODEL**

El rumbo deseado llega a un módulo [Figura 3.4] que genera valores de referencia para el rumbo, su velocidad angular y aceleración angular.

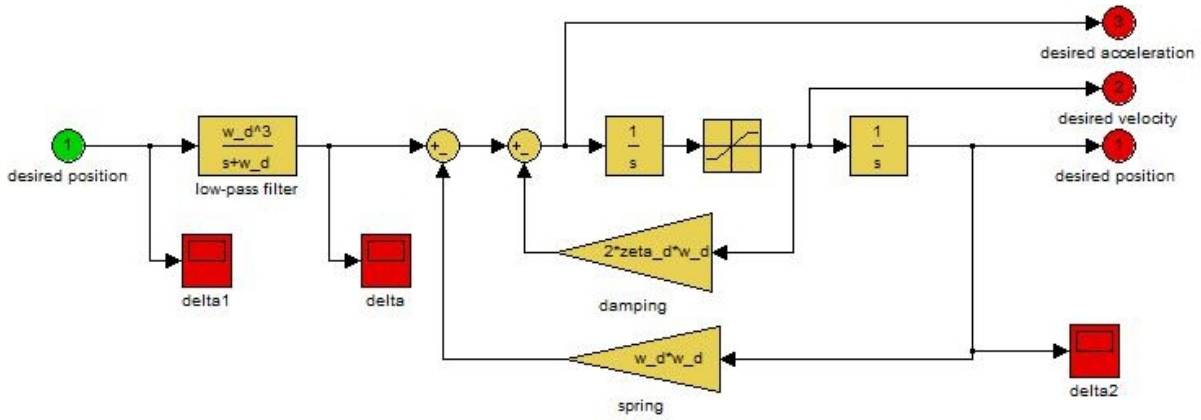



Figura 3.4: Cálculo de referencia

**HEADING AUTOPILOT**

El auto-piloto recibe el rumbo deseado y el rumbo y velocidad angular del barco y genera la señal de control para el timón [Figura 3.5].

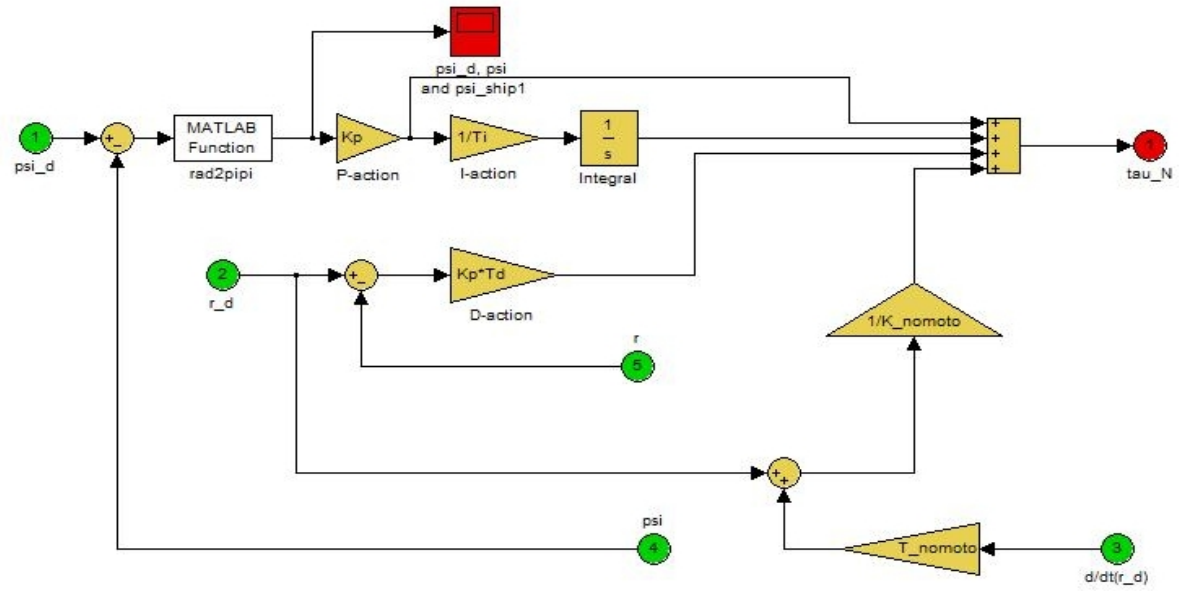



Figura 3.5: Autopiloto

**MARINER CLASS CARGO SHIP**

Afecta al desplazamiento del barco según las corrientes y calcula todos sus valores en función del nuevo movimiento [Figura 3.6].

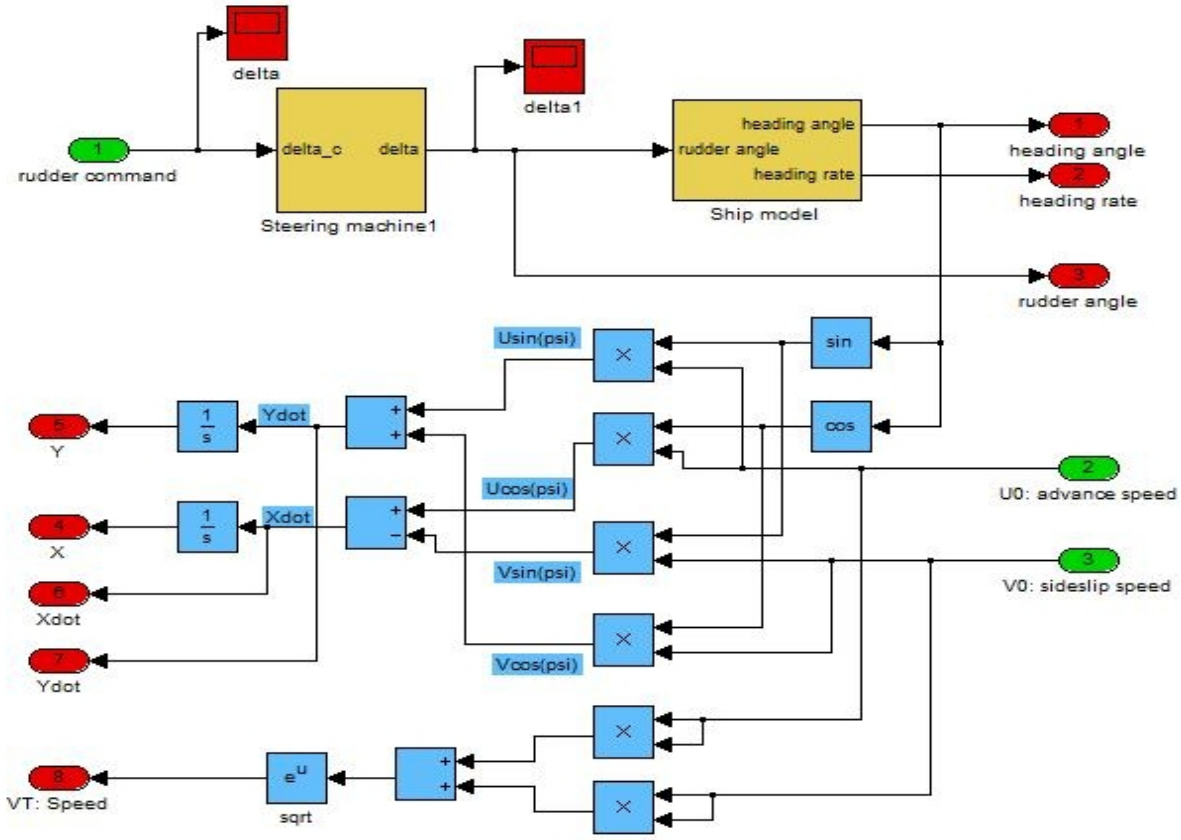



Figura 3.6: Cálculo del nuevo movimiento

## 3.2 - Migración a Java

En un primer momento se intentó la migración de este modelo a Java de manera automática. Por tiempo y eficacia, al tratarse de módulos matemáticos y físicos, buscamos herramientas que cambiaran de lenguaje el código de este modelo.

Las conclusiones después de un tiempo de estudio sobre el tema fueron que Matlab y Simulink permiten una migración directa a C y C++, pero este código no es nada obvio, por lo que en el caso de C los cambios necesarios para conseguir un proyecto orientado a objetos o en C++ pretender cualquier tipo de modificación del código para adecuarlo a nuestro proyecto no resultaban fáciles.

El tiempo dedicado a buscar esta conversión se alargó tanto que decidimos migrar manualmente cada uno de los módulos.

Por lo general ningún módulo supuso un especial problema más allá de comprender Simulink y consultar su documentación. Sólo dos módulos precisan de más atención debido a la escasa información que ofrece la documentación de este programa y a su dificultad:

### 3.2.1 - Funciones de transferencia

Este módulo ha supuesto mucho trabajo. Conseguir un buen funcionamiento de las numerosas funciones de transferencia necesarias no fue fácil.

No estaban disponibles las frecuencias, por lo que se intentó conseguir las transformadas inversas de esas funciones y así dejarlas en función del tiempo. Intentarlo en función de la teoría no funcionó.

El último intento, ya que alcanzó el objetivo fue tras leer el archivo de la ayuda de Simulink: "*Modeling a Continuous System*". Este archivo contiene un ejemplo de cómo pasar a un diseño en Simulink la siguiente ecuación:

$$x'(t) = -2x(t) + u(t)$$

El resultado de este ejemplo es el que podemos ver en la Figura 3.7.

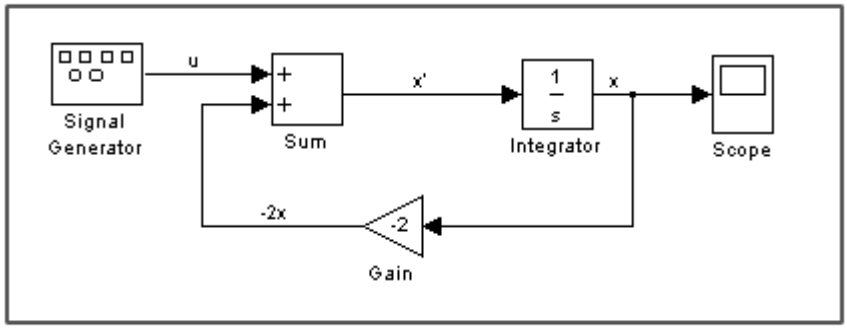



Figura 3.7: Función de transferencia

Este esquema [Figura 3.7] es muy semejante a los diseños con los que trabajamos nosotros, así que basados en él encontramos una solución a todas las funciones de transferencia de primer orden. Para la función de transferencia de segundo orden (sólo teníamos una) se solucionó buscando poner la ecuación como producto de dos de primer

orden.

Convertíamos con esto los dos problemas en uno, los integradores.

### 3.2.2 - Módulo integrador

Consideramos una integral como la suma de las pequeñas áreas que genera la función. Esto, como se recordará de años anteriores supone añadir un pequeño error a estos cálculos, en nuestro caso este error se ha tomado por defecto.

El problema que supone esta implementación es el intervalo que toma la función en cada una de esas áreas, ya que depende del módulo en el que se encuentre el integrador. Para tomar estos valores, se ha trabajado de manera individual con cada uno de ellos, calculando en función de los resultados esperados y obtenidos, el intervalo más conveniente.

## 3.3 - Validación del modelo

Informe sobre la migración del proyecto en Simulink llamado "*Modelo de barco para control, planificación y seguimiento de trayectorias"* creado por el profesor Jesús M. de la Cruz en el año 2008 a Java realizado al acabar este proceso:

Esta simulación a generado sobre un barco de carga, tras darle una ruta objetivo y siendo influenciado por una corriente constante.

**RESULTADOS OBTENIDOS:**

Vamos a comparar alguna de las gráficas que obteníamos a través de la aplicación Simulink con los resultados en Java.

En todas ellas emplearemos los siguientes valores de entrada:

$$U_0 = 8$$
$$V_0 = 10$$
$$Rumbo\,Trayectoria = 95^o$$
$$Posición\,Trayectoria(X, Y) = (100{,}50)$$

Hemos tomado muestras en distintos puntos del modelo durante una ejecución; estos son los resultados obtenidos. En la primera gráfica podemos observar el comportamiento en Simulink y en la segunda los resultados de nuestra aplicación

Muestra 1: $\psi_d$, $\psi$, $\psi_{Ship2}$

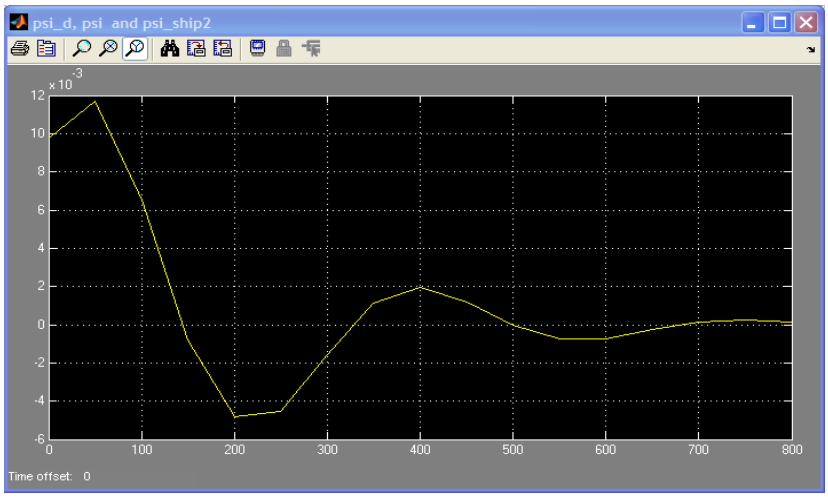



Figura 3.8: Muestra 1 - Simulink

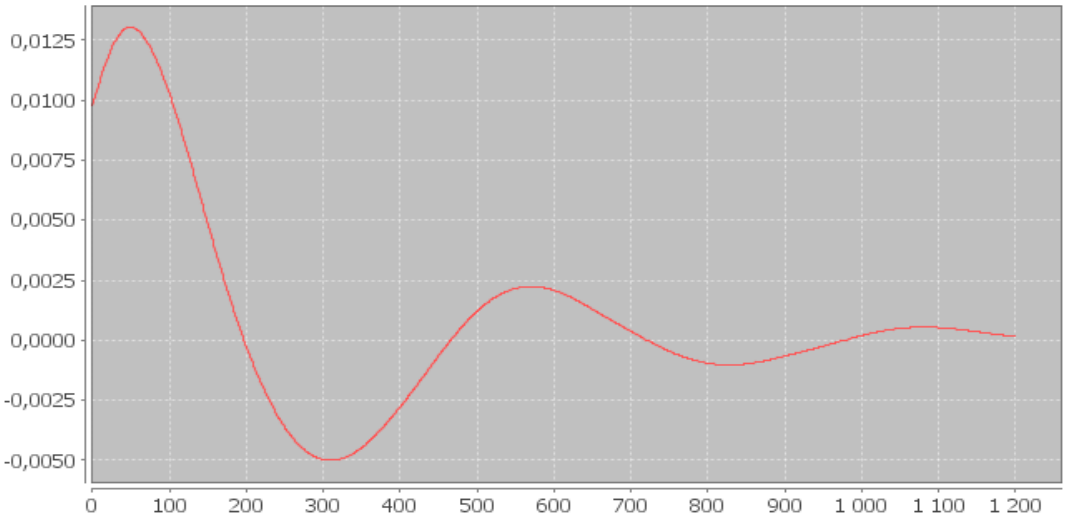



Figura 3.9: Figura 1 - Pruebas sobre Dolphin

Muestra 2: *Ángulo del timón*: $r_{Ship}$

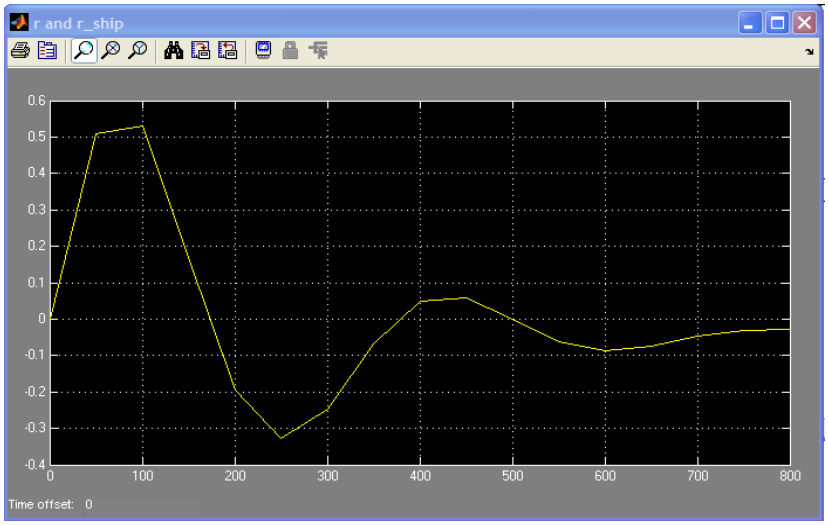



Figura 3.10: Muestra 2 - Simulink

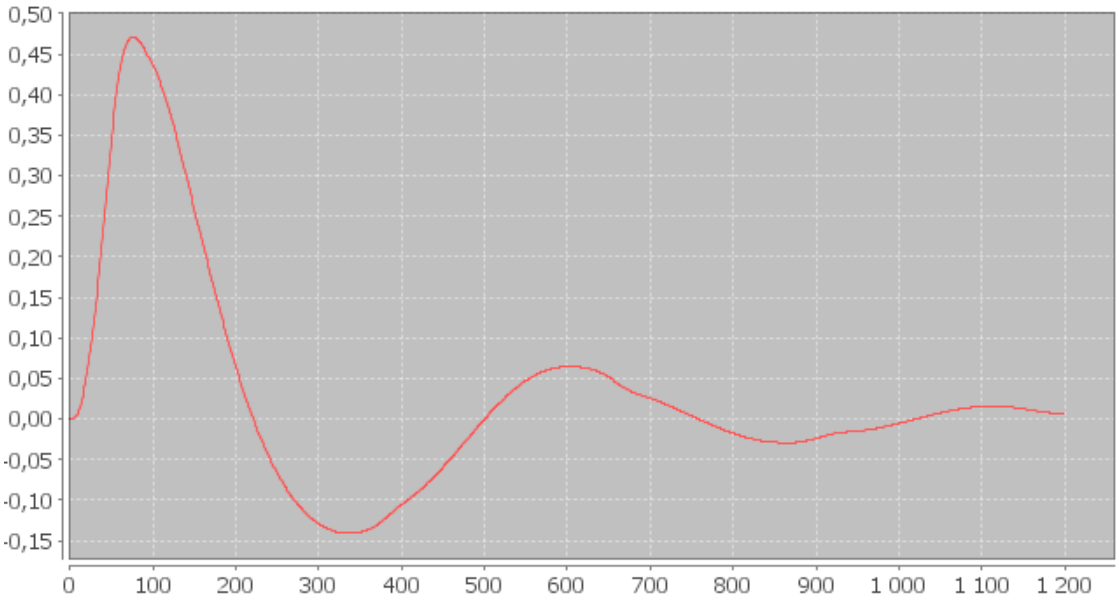



Figura 3.11: Muestra 2 - Prueba sobre Dolphin

Muestra 3: *Ángulo de trayectoria* $:\Delta$

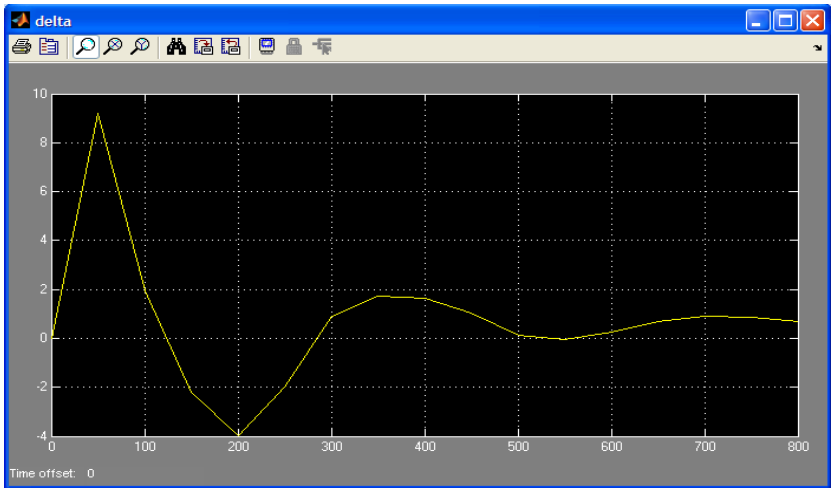



Figura 3.12: Muestra 3 - Simulink

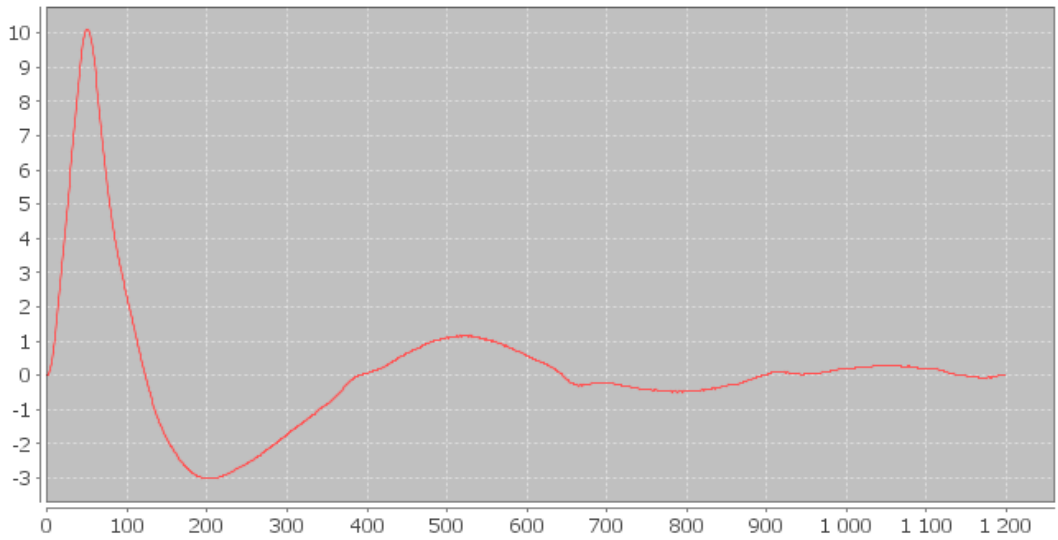



Figura 3.13: Muestra 3 - Pruebas sobre Dolphin

Muestra 4: *Resultado de la ejecución*

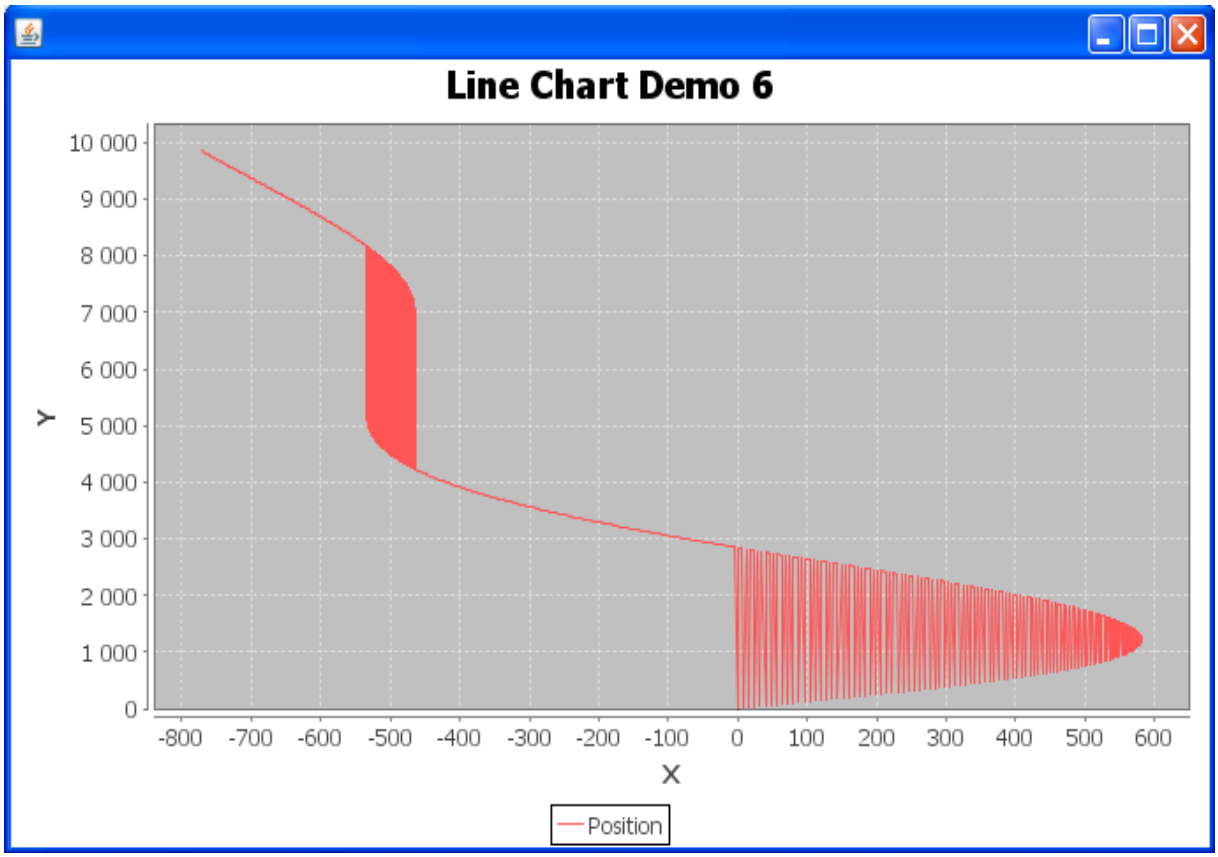



Figura 3.14: Resultado sobre Dolphin

Conclusiones:

Lo primero decir que aunque en esta gráfica final parezca haber pulsos incorrectos, no son producto de un mal funcionamiento del modelo sino de las librerías de representación de gráficas utilizadas. Esto se comprobó estudiando valor a valor los resultados, pero no se ha incluido en la memoria por simplificar.

Podemos ver que en todos los casos las gráficas obtenidas mantienen los rangos y la evolución de la gráfica original, como se puede apreciar tienen un pequeño desfase en iteraciones.

El producto de este desfase, aunque no está comprobado, creemos que es la aproximación experimental del paso temporal de los integradores. Independiente del motivo,

este pequeño error no modifica ni perjudica al movimiento del barco, que mantiene una trayectoria y posición correcta.

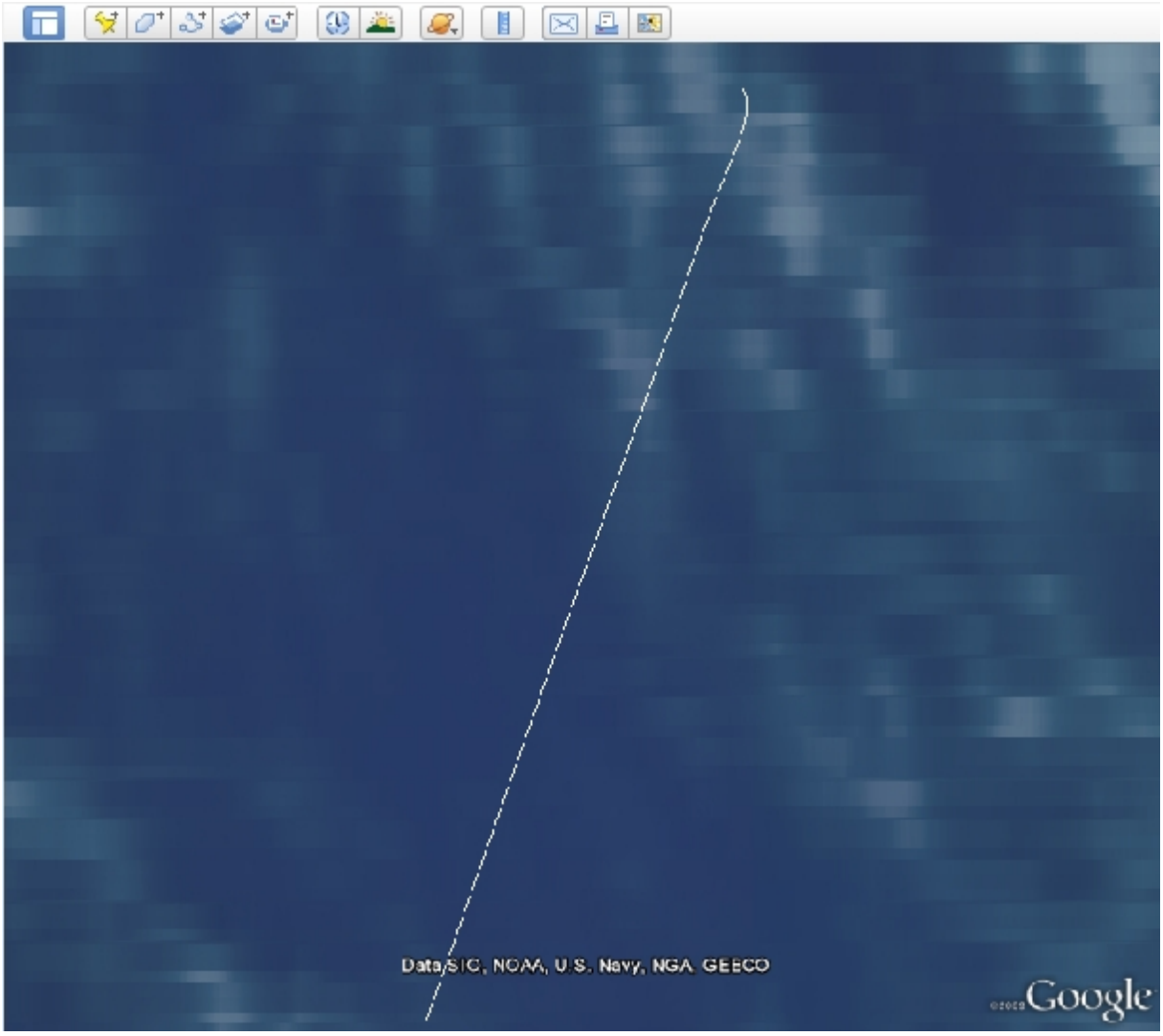



Figura 3.15: Modelo sobre Google

Para hacer funcionar este trabajo, tenemos que tener un vehículo que se desplace. Este vehículo tendrá el comportamiento de un agente inteligente. Este concepto ya se ha explicado en el apartado anterior, pero ahora vamos a ver la aplicación concreta sobre este proyecto:

## 3.4 - Implementación del barco como agente

Desde un principio, Dolphin se planteó como una posible continuación del proyecto

Seagull: “Coordinación de sistemas Multiagente para Tareas de Localización de Objetos”. Resumidamente constaba de la creación de una arquitectura de agentes y de un sistema de comunicaciones aplicado en un principio para agentes que representan aviones no tripulados.

En esta primera versión del denominado proyecto Dolphin, hemos realizado un estudio sobre la viabilidad del uso del framework JADE para la implementación de la plataforma de agentes. Como hemos dicho antes, se han descartado opciones muy interesantes como Grasshoper para evitar dificultades con la integración con el proyecto hermano que trabajaba con aviones.

En Seagull, se diseñó una arquitectura de agente nueva procedente del diseño TouringMachine, clasificada como híbrida, a la que se le asignó el término de Paralell Machine. Básicamente combina capacidades reactivas de los agentes con elementos propios de BDI: *Creencias (Beliefs), Deseos (Desires) e Intenciones(Intentions)*, a los que incorpora paralelismo de tareas.

En el planteamiento principal de Dolphin, se barajaron muchas posibilidades con las que trabajar, una de ellas era la ampliación de Seagull a agentes marítimos. Dada la complejidad que aportó la adaptación del modelo del barco, se optó por iniciar un nuevo proyecto que estuviera preparado para unirlo en un futuro con Seagull.

Por tanto, en Dolphin las comunicaciones entre agentes han sido implementadas al uso, es decir, no se ha creado una ontología específica para el paso de mensajes entre agentes pero si se ha creado una sintáxis para el reconocimiento de órdenes provinientes de la base . El desarrollo con JADE ha consistido en la creación del agente que determina el comportamiento del Barco; se ha implementado por tanto la recepción de mensajes y la actuación frente a estos, la unión con el modelo y el comportamiento de trayectoria de un punto a otro. En la sección de posibles ampliaciones hablaremos más de esta unión.

De cara a dar facilidades a los posibles continuadores de estos dos proyectos, especificaremos las clases que contiene el paquete *agents* creado en Dolphin:

*#AgenteBarco.java*

Esta clase es la clase que se instancia cuando se crea un agente de tipo barco. Por tanto extiende a la clase jade.core.Agent e implementa los métodos necesarios para su creación, registro en el df y especificación de comportamientos. El comportamiento inicial del agente es escuchar a la base en espera de instrucciones.

# ComportamientoEscucha.java

Esta clase, que extiende a la clase jade.Corebehaviours. SimpleBehaviour, define el comportamiento por defecto del agente del barco. Es la capa que determina cuál de los otros comportamientos deben adoptarse. Actualmente está preparado para aceptar mensajes para el comportamiento de realizar el trayecto óptimo de un lugar a otro.

# RecorrerRuta.java

Aquí se realiza la unión con el módelo para la determinación del comportamiento de trayecto de un punto a otro. Procesa y valida los datos y genera las respuestas oportunas.

# BarrerEspacio.java

Esta clase define el comportamiento de barrer un área de mar. No se ha podido terminar de implementar pero el esqueleto está planteado y no sería difícil terminarlo en una futura ampliación. Se ha dejado parado para no repetir los métodos algorítmicos de planificación de recorridos de espacio que ya han sido implementados en la herramienta Seagull.

# BoatAgent.java

Como reseña histórica se ha dejado esta clase para la verificación de la aplicación con la exclusión de la plataforma de

agentes. En un principio fue la clase que se utilizó para unir el comportamiento de ruta con el modelo del barco. Si en un futuro se deseara ampliar alguna parte y las pruebas no quisieran hacerse con agentes por las limitaciones en tiempo que ello conlleva (debido al paralelismo de procesamiento entre los agentes), esta clase sería la que haría falta utilizar.

Con el fin de poder presentar buena parte del trabajo, se ha creado una base de comunicaciones capaz de comunicarse con los agentes barco. También han sido creados y utilizados los métodos necesarios para poder instanciar los agentes; si bien estos se crean al simular la aplicación y conviven en una base común propia para cada misión, estos no interactúan todavía entre sí; esta será la ampliación de cooperacion en misiones.

# Capítulo 4 -

# Google Earth y tratamiento de imágenes

Dolphin está programado para leer imágenes de mapas de satélite y procesarlas para poder luego ejecutar el modelo del barco sobre dicho mapa. En nuestro caso hemos usado imágenes de Google Earth 5. En principio Dolphin debería poder reconocer imágenes de mapas de otras fuentes, pero los resultados no se mostrarán entonces sobre esta plataforma.

La incorporación de este programa suponía uno de los grandes retos del proyecto, ya que aunque nos evitaba el tiempo que se necesita en desarrollar una interfaz en Java 3D, u opciones similares, teníamos en contra una herramienta de tamaño, opciones y desarrollo ingente completamente desconocida en los aspectos que íbamos a necesitar.

# 4.1 - Google Earth

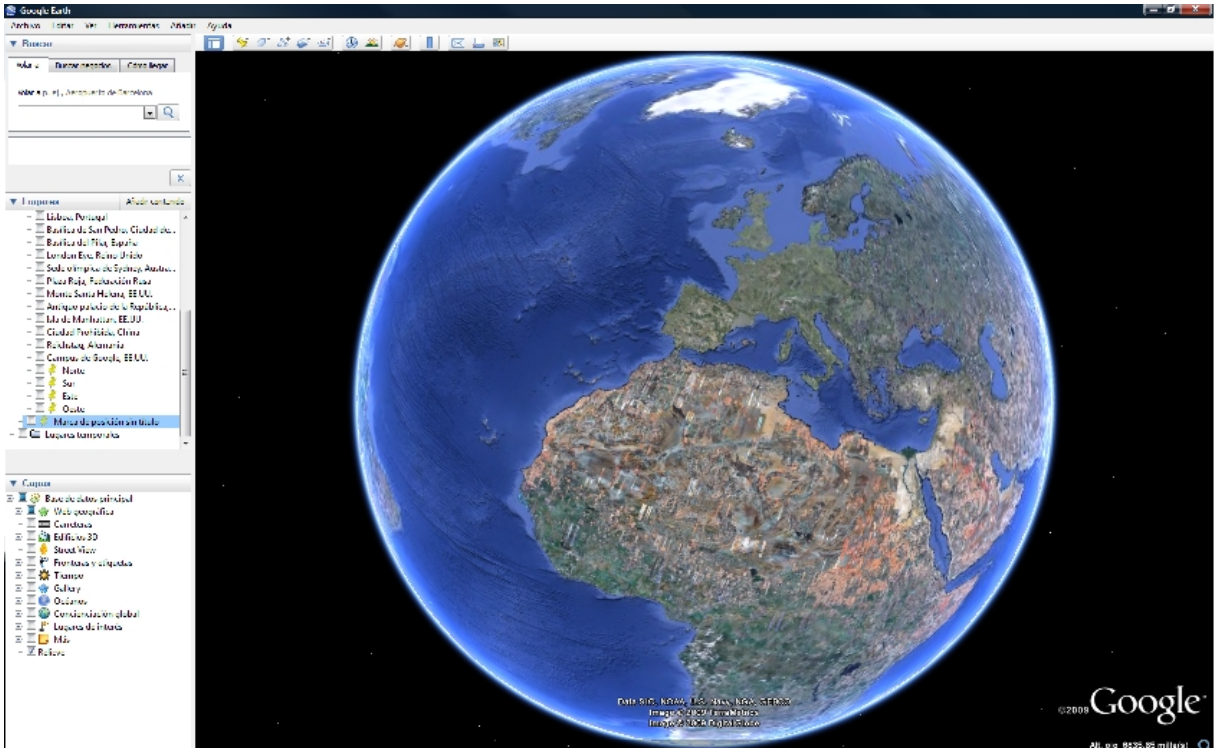



Figura 4.1: Vista inicial de Google Earth

El embrión de Google Earth lo realizó una empresa llamada Keyhole, Google compró a Keyhole y lanzó el programa, con muchas nuevas facilidades bajo la denominación Google Earth en 1995.

Pero tras esta curiosidad no merece la pena aburrir con historia o una descripción exhaustiva del programa cuando toda la información referente a esta herramienta la podemos encontrar en: http://earth.google.es/intl/es_es/.

Vamos a resumir las características con las que más hemos tenido que tratar y los problemas que nos hemos ido encontrando:

## 4.1.1 - Marcas de señalización

Seleccionar un punto y hacer que Google lo recuerde es muy sencillo. Es posible

marcar cualquier punto en el mapa, asociarle una información y conseguir que cualquiera, desde cualquier ordenador tenga acceso a ello, esta era la idea de Google. Pero obviamente este no es nuestro interés, sin embargo podemos usar parte de esta funcionalidad para lo que poder identificar distintos puntos en el mar.

Por otro lado, Google ofrece la posibilidad de guardar la información en distintos formatos seleccionables desde la opción del menú "Añadir" o más sencillo desde los siguientes iconos:

De ellos, nosotros usamos la chincheta, para señalar puntos únicos que nos son útiles por ejemplo para señalizar los mapas con los que trabajamos y las líneas, para dibujar el resultado obtenido en forma de recorrido, que puede ser reproducido en "modo viaje": seleccionando el icono que sale en la mitad izquierda de la pantalla, después de haber seleccionado un archivo de tipo ruta:

Con esto, aparecerá un controlador del video y se irá reproduciendo el recorrido desde el origen hasta el fin, dejando la linea generada en la zona central de la pantalla.

### 4.1.2 - Formato de archivos

Cada una de las imágenes visibles desde el programa puede ser guardada en 2 formatos reconocibles por Google Earth (.kml y .kmz, siendo el segundo simplemente un archivo .kml comprimido) así como en formato imagen con extensión jpg.

Aunque en un principio pensábamos que con tratar sólo el primero mencionado nos bastaría, tuvimos que utilizar también el formato imagen por razones que se comentarán más adelante. Primero creemos necesario definir un poco las características básicas del lenguaje kml, por ser éste menos conocido y muy usado en nuestro proyecto.

#### KML

**¿Qué es?** El KML es un lenguaje que se basa en los estándares del XML y que puede ser interpretado por Google Earth. Del mismo modo que un navegador lee un archivo HTML y nos muestra su contenido, Google Earth lee un archivo KML y muestra en las imágenes de satélite los contenidos que contenga. Éstos pueden ser marcas de posición, descripciones, líneas, polígonos…

**¿En qué consiste?** El lenguaje KML tiene una estructura de tags con elementos anidados y atributos. Un archivo .kml puede contener desde un objeto con sus atributos (por ejemplo una chincheta marcando la situación de la Torre inclinada de Pisa) hasta tantos objetos como queramos (por ejemplo la situación de todos los estadios de los equipos de la 1ª División española).

Por supuesto en ningún caso se ha tenido que realizar un archivo de este estilo desde el principio, Google los genera desde la opción "Archivo -->Guardar" (opción sólo permitida si se ha generado anteriormente alguna marca de señalización).

Pero aquí nos encontramos una pequeña sorpresa. Las plantillas del archivo son ligera, pero significativamente diferentes si lo que estamos guardando es un punto único (con la chincheta) o una línea, pero en ambos casos sólo permite en un mismo archivo guardar una única marca.

La primera característica mencionada este último párrafo (plantillas distintas), no supuso un problema mayor en el momento que se identificó, mientras que la segunda (no poder guardar más de una marca) supuso un replanteamiento de la secuencia inicial del programa.

En un primer momento, la selección de los puntos origen y destino se iban a realizar desde el programa Google Earth en lugar de hacerlo desde nuestra aplicación, pero ahora esto suponía perder el tiempo de apertura del programa, más la generación de los dos puntos y guardarlos por separado, siguiendo unos protocolos para la selección de nombres y directorio en los que guardarlos. Además de que en una versión tan prematura del proyecto, era complicada la posibilidad de permitir al usuario modificar la amplitud, aumentándola (disminuirla no genera en ningún caso problemas).

Por otro lado, también es importante comentar que este formato no nos permitía de una manera sencilla diferenciar si un punto determinado del mapa es tierra o mar.

Estos fueron los motivos que nos llevaron a llevar una doble representación: cada mapa tiene que tener almacenada su información en formatos kml y jpg.

## 4.2 - Tratamiento de imágenes

El objetivo de esta sección es reconocer y distinguir las zonas navegables de las que no lo son. Basándonos en imágenes de Google Earth 5 debemos diferenciar mares y océanos de islas y demás superficies terrestres. El reconocimiento del terreno se fundamenta en la distinción de colores. Con unas imágenes del mismo tipo y unos colores de tonos semejantes (como la mayoría de mapas actuales), esta parte del proyecto seguiría funcionando con normalidad.

### 4.2.1 - Procesamiento y binarización de imágenes

La diferenciación de colores a simple vista parece complicada. Estamos trabajando sobre imágenes en RGB. Cada píxel está formado por 3 componentes de color: rojo(R), verde(G) y azul(B). Y cada una de estas componentes puede tomar valores de 0 a 255, es decir, que podemos obtener $256^3 = 16.777.216$ colores distintos. La clasificación de tantos colores puede llegar a ser imposible. Necesitamos reducir el abanico de colores.

La manera de actuar es leer uno a uno todos los píxeles de la imagen e intentar irlos clasificando. La distinción se hará en base a los colores, por tanto hemos de mirar para cada píxel sus componentes R, G y B. Pasamos cada píxel por un filtro que determinará si pertenece o no a una zona navegable.

La construcción de este filtro fue difícil, pues aunque no lo parezca el rango de colores al que puede pertenecer un píxel de zona marítima es bastante amplio. No vale sólo con fijarse en la componente azul de cada punto. Necesitábamos de alguna manera reducir este rango.

Decidimos aplicar el algoritmo LVQ a nuestro problema. A continuación presentamos

una introducción teórica sobre este método para luego explicar los resultados que obtuvimos para nuestro problema:

## **ALGORTIMO LVQ** (LEARNING VECTOR QUANTIZATION):

Introducción teórica:

Esta red es un híbrido que emplea tanto aprendizaje no supervisado, como aprendizaje supervisado para clasificación de patrones. Cada neurona de la primera capa es asignada a una clase. Después, cada clase es asignada a una neurona en la segunda capa. El número de neuronas en la primera capa, debe ser mayor o al menos igual que el número de neuronas en la segunda capa.

Cada neurona en la primera capa de la red LVQ aprende un vector prototipo, el cual permite a la neurona clasificar una región del espacio de entrada.

Así, la neurona cuyo vector de pesos esté cercano al vector de entrada tendrá salida 1 y las otras neuronas, tendrán salida 0. En este aspecto, la red LVQ se comporta igual a las redes competitivas, la única diferencia consiste en la interpretación. Mientras que en las redes competitivas la salida no cero representa una *clase* del vector de entrada, para el algoritmo LVQ, índica más bien una *subclase*, y de esta forma muchas neuronas (subclases), conforman una clase.

La segunda capa de la red LVQ es usada para combinar subclases dentro de una sola clase. La red LVQ combina aprendizaje competitivo con aprendizaje supervisado, razón por lo cual necesita un set de entrenamiento que describa el comportamiento propio de la red.

Para nuestro caso, lo que conseguimos es que un conjunto de tonos semejantes pasen a formar parte de la misma clase. Con esto logramos reducir considerablemente el abanico de colores que forman parte de las zonas marinas. Notar que no nos interesa mucho trabajar con zonas terrestres, pues su rango de posibles colores es mucho mayor: muchos tonos de verde, blanco de los polos, desiertos…

Para lograr reducir el número de colores de nuestro mapa debemos ajustar dos

parámetros importantes sobre la ejecución de nuestro algoritmo: el umbral y el número de muestras. Un mal ajuste podría llevarnos a una ejecución demasiado larga o incluso a una mala diferenciación de las regiones de agua como demostraremos después.

Tanto el número de muestras como el umbral determinarán el número de clases distintas de colores a las que reduciremos nuestra imagen. El número de muestras es el número de píxeles elegidos al azar de todo el mapa para trabajar. Para cada uno de estos píxeles buscamos su clase más cercana, es decir, miramos la distancia euclidea de las componentes R, G y B para cada una de las clases ya clasificadas. Si la distancia es menor que el umbral seleccionado, asignamos el píxel en cuestión a la clase de menor distancia encontrada. Si por el contrario, la distancia es mayor que el umbral, creamos una nueva clase para el punto con el que estamos trabajando. Por supuesto, al comenzar la ejecución del algoritmo sólo hay una clase ya clasificada, la del píxel elegido en la primera muestra.

Una vez que ya hemos tratado todas las muestras, simplemente tenemos que asignar al resto de píxeles de la imagen a la clase a la que están más cerca (también según la distancia euclidea de sus componentes R,G,B).

A continuación, presentamos una serie de ejecuciones del algoritmo LVQ para ver los posibles errores de una mala elección de los parámetros requeridos.

En la siguiente imagen [Figura 4.2], podemos ver una captura de Google Earth 5 con la que se podría trabajar perfectamente en el proyecto Dolphin. Es un mapa del archipiélago Balear. A partir de esta imagen ejecutaremos nuestro algoritmo.

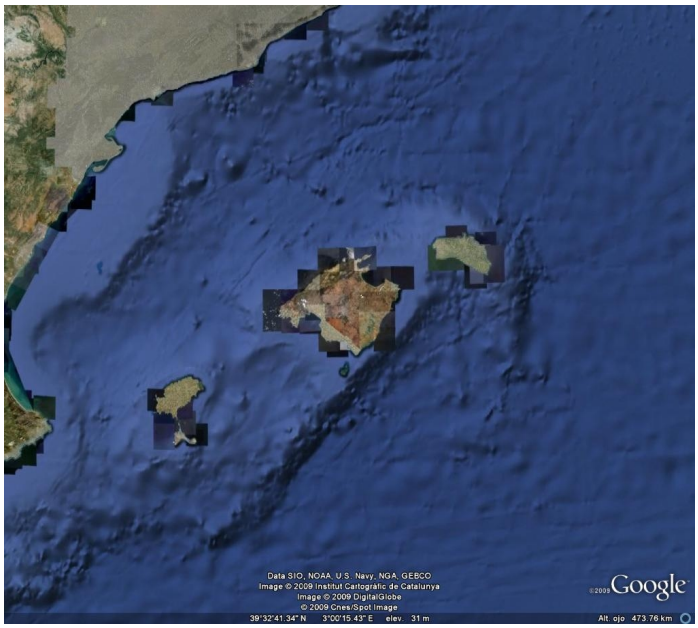



Figura 4.2: Imagen original tomada de Google Earth

Un buen ajuste da como resultado la imagen mostrada en la siguiente ilustración [Figura 4.3]. La mayor parte del mar queda en función de dos tonos de azul. Es cierto que luego hay zonas más oscuras, pero la reducción de colores de la mayor parte es considerable.

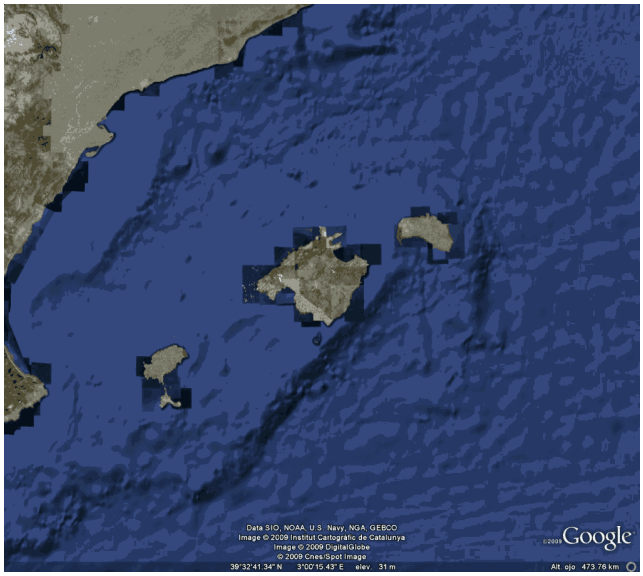



Figura 4.3: Resultado de la primera ejecución de LVQ

El tratamiento que el algoritmo haga a los colores referentes a zonas de tierra nos es irrelevante, siempre y cuando ninguna zona de mar pertenezca a la clase de una zona de tierra

. Se podría ajustar aún más para intentar reducir el número de clases de colores. El resultado se puede ver en la captura que sigue [Figura 4.4]. Observamos cómo el mar Mediterráneo es cada vez más uniforme, hay un color predominante y ciertas zonas más oscuras como en el ejemplo anterior.

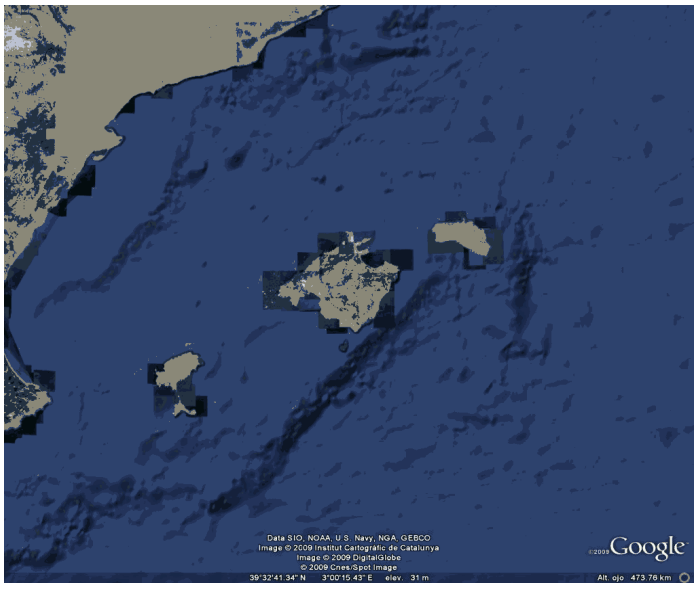



Figura 4.4: Resultado de la segunda ejecución de LVQ

El problema viene cuando nos fijamos en la parte de Mallorca. Si nos fijamos detenidamente, podemos ver unas zonas de la isla que no deberían tener ese tono de azul. Vemos cómo este ajuste de umbral y número de muestras comienza a dar lugar a zonas de error.

Para demostrar de manera más clara este asunto, nos podemos fijar en la siguiente ilustración [Figura 4.5]. Encontramos zonas muy extensas de mar que se pueden confundir con zonas terrestres. Hemos intentado reducir en exceso el número de colores y hemos llegado a un error.

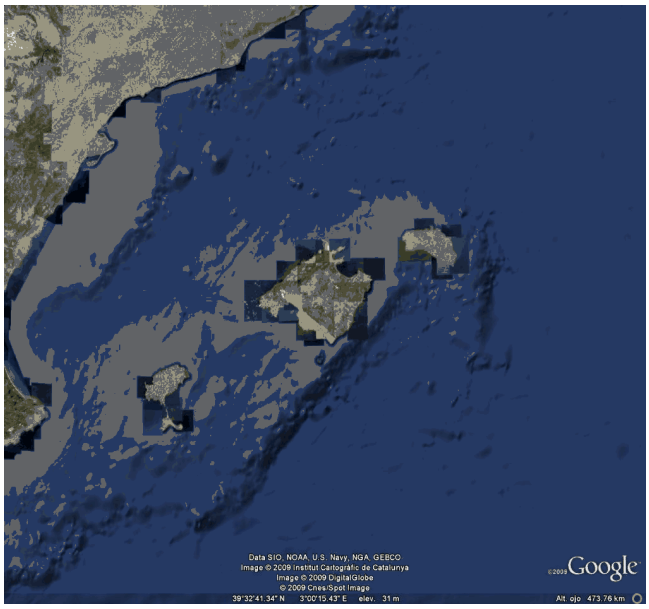



Figura 4.5: Resultado de la tercera ejecución de LVQ

Por tanto, concluimos con que es necesario un buen ajuste para reducir considerablemente los distintos tonos de mar/océano pero sin dar lugar a posibles confusiones.

Una vez tenemos la imagen después de aplicar nuestro algoritmo LVQ, el siguiente paso es la *binarización de la imagen*. Para esto, hemos construido un filtro a través del cual pasamos todos los píxeles de nuestro *mapa modificado*. Dicho filtro nos dirá si el pixel en cuestión pertenece o no a una zona navegable para nuestro barco.

La representación siguiente [Figura 4.6] muestra el resultado de todo este capítulo. Observamos el mapa de Baleares en blanco y negro. Las regiones negras son las zonas de tierra (0's en nuestro mapa binario) y el mar Mediterráneo lo vemos en blanco (1's según nuestra representación).

Figura 4.6: Mapa de bits resultante

A partir de este mapa se calcularán las trayectorias óptimas para recorrer trayectos; todo esto se explicará en el capítulo 5.

Para reducir tiempos de ejecución en lo que respecta a este algoritmo, hemos decidido usar archivos binarios para copias de seguridad en disco de las matrices de bits. Al pensar que para un mismo mapa es bastante probable que el usuario decida hacer varias rutas, el LVQ sólo ejecutará la primera vez. El resto leerá un archivo *.dat* desde el disco, agilizando la ejecución en un tiempo considerable.

# Capítulo 5 -

# Planificación de trayectorias

En este capítulo explicaremos cómo nuestro barco es capaz de moverse entre dos puntos evitando obstáculos (islas). Por supuesto, cuando hablamos de la trayectoria entre estos dos puntos, estamos refiriéndonos a la trayectoria más corta o una aproximación a ésta. Aquí hablamos de obstáculos en general como islas, pero claro está que, con imágenes más detalladas, se podrán considerar más tipos de obstáculos e introducir zonas de radar a evitar.

Detallaremos los pasos dados hasta alcanzar el algoritmo usado y la transformación entre los diversos sistemas de representación necesarios para trasladar los resultados a Google Earth.

## 5.1 - Primera aproximación

El trayecto más corto entre dos puntos es la línea recta. Cuando se está trabajando sobre una matriz de bits, muchas veces es complicado ir de un punto a otro en cuando estos dos puntos no se encuentran en la misma horizontal, vertical o diagonal. Es sencillo llevar los

índices de los bucles en la misma horizontal o vertical y también recorrer matrices en diagonal.

Aún así, intentamos desarrollar un algoritmo que para ir de un punto a otro cogiera siempre la mejor opción. Llevábamos bucles teniendo en cuenta las pendientes de las rectas para recorrer la matriz de bits, considerando la parte entera y decimal de la pendiente para no tener un "desfase" a la hora de incrementar los índices. Al final lo conseguimos, unos bucles bastante engorrosos pero que funcionaban bien. Teníamos el trayecto más corto entre dos puntos, pero ¿y si nos encontrábamos un obstáculo?

En la primera aproximación intentamos "jugar" con la Geometría. Nos hemos desplazado en línea recta hacia nuestro objetivo hasta dar con "un píxel negro". No podemos avanzar y buscamos alejarnos de ese obstáculo. Para ello cogemos la recta perpendicular a la recta que unía los puntos inicial y final. Nos desplazamos por esa recta hasta encontrar una nueva zona transitable e intentamos continuar nuestro viaje hasta el punto destino.

Esta idea funcionó al principio sobre simples bocetos con obstáculos de formas básicas como cuadrados, círculos o triángulos. Pero a la hora de llevarlo sobre un mapa de verdad, la variedad de formas de las islas, islotes o cabos a rodear con nuestro barco, dejó nuestro algoritmo inservible.

## 5.2 - Segunda aproximación: Grafos de visibilidad

Nuestra siguiente opción fue el uso de *grafos de visibilidad*. Estos algoritimos son muy usados en el ámbito de la Robótica. Los grafos de visibilidad (Nilsson, 1.969) proporcionan un enfoque geométrico para solventar el problema de la planificación. Este método se encuentra muy extendido debido a que opera con modelos poligonales de entorno, con lo que existen algoritmos que construyen esta clase de grafos con un coste computacional relativamente bajo: $O(n^3)$ (Lozano-Pérez y Wesley;1.979). Este método necesita modelos de entornos definidos con polígonos.

Consideramos los nodos del grafo como los vértices de cada polígono u obstáculo. Nos queda un grafo no dirigido con unos nodos conectados siempre que sean "visibles", es

decir, siempre que se pueda alcanzar el segundo nodo desde el primero y viceversa; no hay ningún obstáculo en la línea recta que los une.

Una vez construido el grafo, el siguiente paso era la elección de un algoritmo de búsqueda. Nunca alcanzamos este punto. La construcción de los grafos de visibilidad a partir de los mapas de Google Earth nos daba unos grafos excesivamente complejos. Aprendiendo ya de los errores de la primera aproximación (punto 8.1), tuvimos en cuenta desde el principio la clase de mapas que íbamos a tratar, y no los ejemplos clásicos de rutas con dos obstáculos de formas cuadradas [Figura 5.6]. La diversidad de formas de los obstáculos de nuestro mapa de bits nos dio como resultado unos grafos enormes. Llevar búsquedas a través de semejantes grafos retrasaría muchísimo la ejecución.

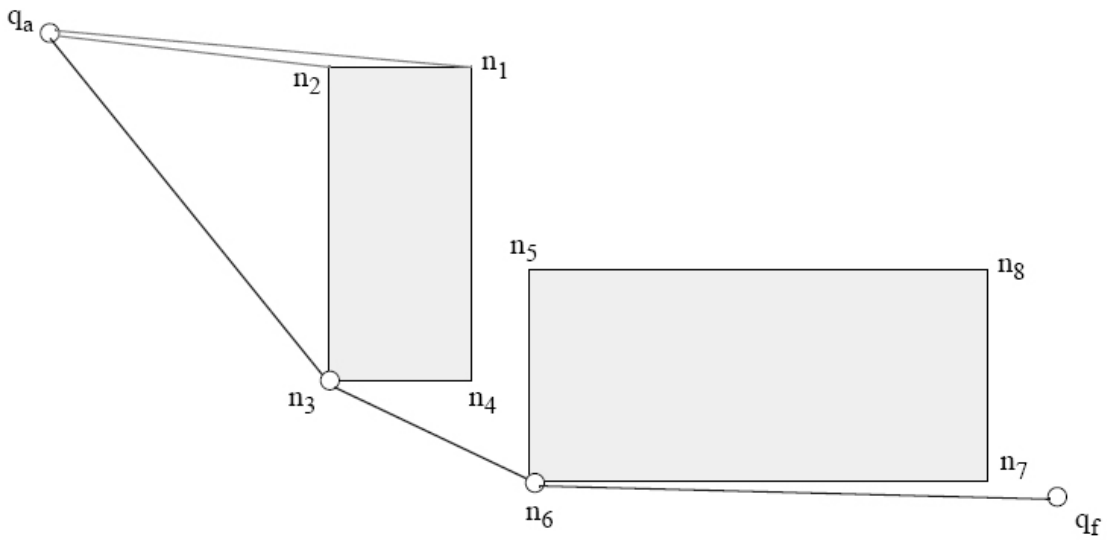



Figura 5.1: Ejemplo de grafo de visibilidad

# 5.3 - Tercera aproximación: Pathfinding

Tras una mejor documentación sobre el tema, decidimos llevar a cabo la implementación del algoritmo *Pathfinding* o A* (A estrella) en nuestro proyecto.

El *Pathfinding* es uno de los algoritmos más populares de búsqueda de trayectos óptimos entre dos puntos. Se ayuda de una función heurística que estima el coste que conllevaría ir desde cada nodo al objetivo. Gracias a esa función, iremos eligiendo los nodos

de acuerdo con la probabilidad de que pasando por ellos se alcance el camino más corto.

Nuestro algoritmo trabaja sobre imágenes, es decir, sobre una matriz de píxeles. El resultado obtenido, la ruta óptima, estará representada sobre esta matriz. Por tanto, al finalizar la ejecución del A* deberemos hacer las conversiones pertinentes a los sistemas de representación necesarios en la simulación y en la salida a través de Google Earth.

Las imágenes que reconocemos, son capturas de Google Earth 5, y por lo tanto depende del ordenador donde se hayan tomado la resolución de éstas. En vistas de que los ordenadores cada vez disponen de monitores más grandes y con mayor resolución, dejamos preparado el proyecto para que ejecute de manera *rápida* en estas nuevas pantallas. Si tomamos una fotografía de Google Earth con una pantalla grande a una gran resolución, nuestra imagen tendrá también una gran resolución. Si tratáramos de ejecutar el *Pathfinding* sobre semejante cantidad de píxeles, la ejecución se demoraría en exceso para las rutas complejas.

Como solución al problema, hemos planteado el enceldado de la imagen. Tomando divisores de las resoluciones de las imágenes, dividimos en varias celdas de mayor tamaño el mapa. Disminuyendo el número de nodos, el *Pathfinding* ejecutará a una mayor velocidad, y la aproximación es bastante buena.

Podemos movernos en 8 direcciones: por los ejes horizontal y vertical, y por las diagonales que forman ángulos de 45º [Figura 5.2]. Esto posteriormente fue cambiado y mejorado para ajustarse más a la realidad y para mayor sencillez a la hora de la simulación.

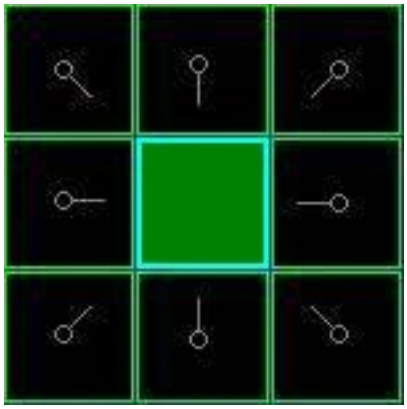


Figura 5.2: Posibles direcciones

A continuación, pasamos a explicar a un nivel más detallado la estructura del *pathfinding*, pseudo-código, heurística... Todo el código referente al cálculo de la trayectoria óptima se encuentra en el paquete NavigationArea.

Internamente, nos basamos en dos listas de nodos: una llamada lista abierta, que contiene los nodos pendientes de comprobar y que pueden formar parte de nuestro camino, y otra llamada lista cerrada, que contiene los nodos ya visitados o expandidos y no hay que volver a tratar.

**Función heurística:** El algoritmo se basa en cada paso en elegir el nodo más prometedor e intentar ir desde ahí al objetivo. La decisión sobre qué nodo debemos expandir en cada momento de la lista abierta depende de la función heurística elegida. Cada celda contiene unos valores F, G y H, donde $F = G + H$ .

•G es el coste actual acumulado desde la celda inicial hasta la actual

•H es el coste de movimiento estimado para ir desde la rejilla actual hasta el destino final. La estimación de este valor para cada celda se calcula antes de comenzar la búsqueda. Nosotros hemos empleado la "distancia de Manhattan": el número total de celdas recorridas horizontal y verticalmente para llegar al nodo objetivo.

### 5.3.1 - Pseudo-código del algoritmo

Presentamos aquí un esquema del algoritmo implementado en el proyecto para realizar

el cálculo de las trayectorias:

*1) Añadir la celda inicial a la lista abierta.*

*2) Repetir:*

*a) Buscar la celda con menor coste de F en la lista abierta. Ésta será la* ***celda actual****.*

*b) Cambiar la celda actual a la lista cerrada.*

*c) Para cada una de las ocho direcciones posibles desde la celda actual, mirar las celdas adyacentes:*

- *Si no es transitable o si está en la lista cerrada, ignorar. En cualquier otro caso hacer lo siguiente:*
- *Si no está en la lista abierta, añadirla a la lista abierta. Hacer que la celda actual sea el padre de esta celda. Almacenar los costes F, G y H de la celda.*
- *Si ya está en la lista abierta, comprobar si el camino para ésta a través de la celda actual es mejor usando el coste G como baremo. Un coste G menor significa que éste es un mejor camino. Si es así, cambiar el padre de la celda a la celda actual y recalcular G y F. Si estás manteniendo la lista abierta por orden de puntuación F, podrías necesitar reordenar la lista para llevar cuenta del cambio.*

*d) Terminar si:*

- *añades el cuadro objetivo a la lista abierta en cuyo caso el camino ha sido encontrado.*
- *fallas en encontrar el cuadro objetivo y la lista abierta esté vacía. En este caso no hay camino.*

*3) Guardar el camino. Moverse hacia atrás desde la celda objetivo. Ir a través de la celda padre de cada nodo hasta llegar a la celda inicial. El camino seguido es el que buscas.*

Al buscar siempre en la lista abierta el nodo con menor F, optamos por implementar esta lista como una cola de prioridad para acelerar las ejecuciones.

El coste de cada movimiento (horizontal, vertical o diagonal) dependerá del tamaño de cada celda. Así por ejemplo en celdas de tamaño 1x1 los costes de movimientos horizontales y verticales podrían ser de 10, mientras que el coste diagonal sería de 14: $\sqrt{1^2+1^2}=1{,}4142$ , aproximando, 14.

## EJEMPLO

Vamos a mostrar gráficamente un ejemplo sencillo del resultado obtenido por el algoritmo.

Ejecutaremos el algoritmo A* sobre este mapa [Figura 5.3], para buscar el camino más corto entre las dos celdas. En este ejemplo tomaremos la precaución de que para ir en diagonal de una celda a otra, la celda lateral debe estar también vacía, para no *chocar* con la esquina del obstáculo.

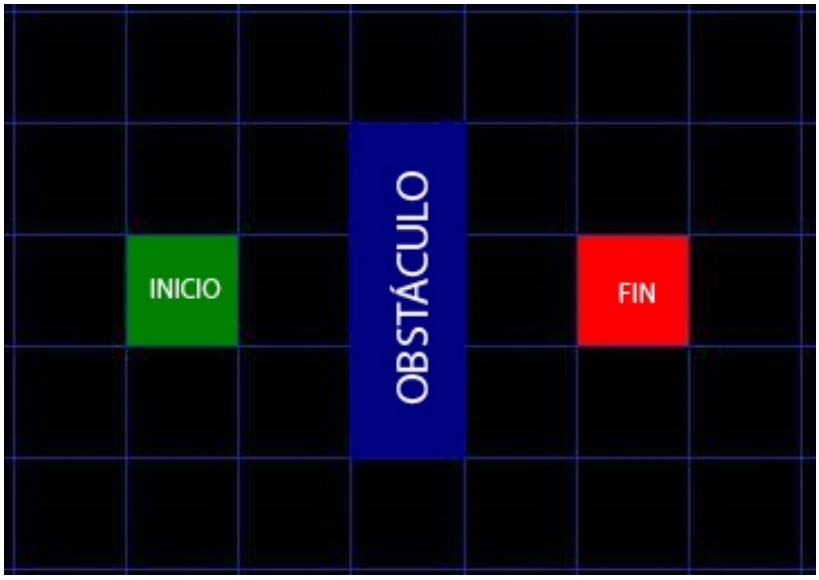



Figura 5.3: Mapa inicial para ejecutar el pathfinding

En la siguiente ilustración [Figura 5.4] se ven todos los nodos que han sido desarrollados. Para cada celda tenemos abajo a la izquierda el coste de G, abajo a la derecha el coste de H y arriba a la izquierda el coste de F, así como también un puntero que indica el nodo padre. Queda reflejado el camino óptimo devuelto por el *pathfinding*, que obtendremos desplazándonos desde el objetivo hasta el punto inicial a través de los punteros *padre* de cada celda.

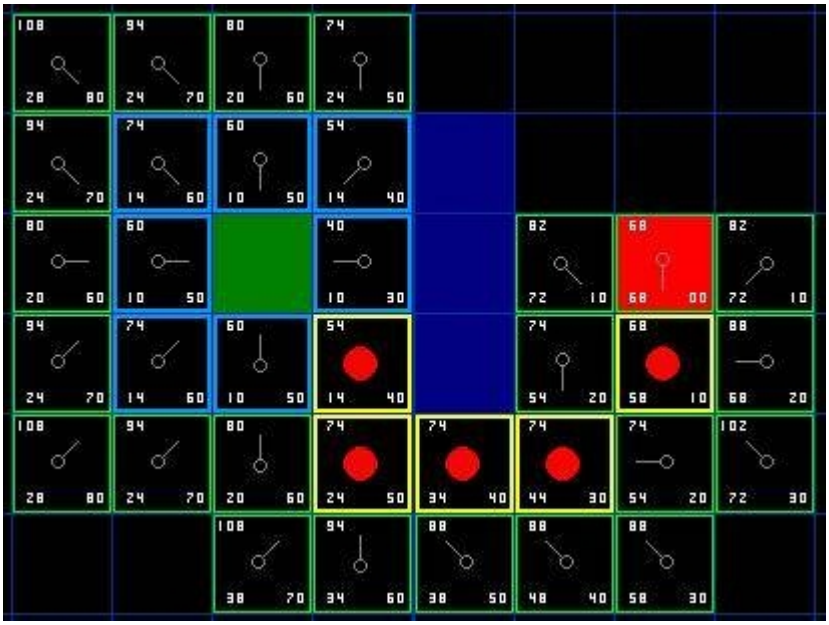


Figura 5.4: Resultado final del ejemplo

### 5.3.2 - Mejora del algoritmo

Al dividir nuestro mapa en celdas rectangulares el trayecto resultante tiende a tener continuas formas en zig-zag no deseables y que no se aproximan a movimiento real de un barco. Intentar ejecutar nuestro modelo sobre estas rutas tan complejas resulta imposible. Hay demasiados cambios de dirección en poco espacio, y hay que tener en cuenta el tiempo que tarda un barco en girar. Es posible que para cuando se alcanzara el ángulo de giro deseado, ya se estuviera haciendo la siguiente maniobra.

Para solucionar esto hemos quitado puntos intermedios *inservibles* de nuestra trayectoria. Si para ir del punto1 al punto3 no hace falta pasar por el punto intermedio2 porque no hay obstáculos en la línea recta que los une, pues eliminamos este punto que lo único que hace es alargar la ruta. La siguiente imagen [Figura 5.5] ejemplifica nuestro problema y solución:

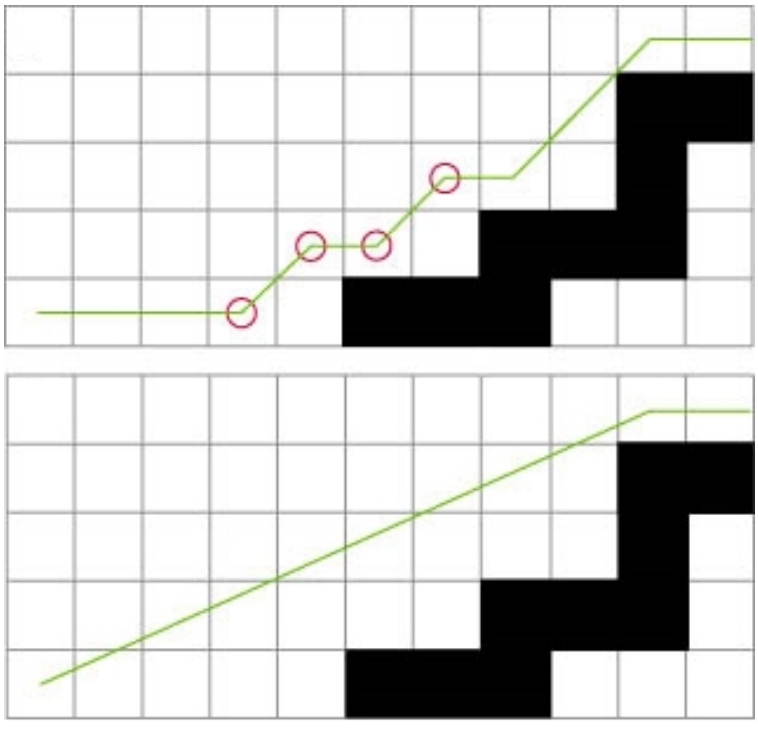


Figura 5.5: Mejora de trayectos

Como se puede observar en la figura de abajo el recorrido entre los dos primeros puntos de la trayectoria no se puede hacer con los habituales incrementos usados en el recorrido de matrices. Hemos tenido que tomar un número de muestras a lo largo de la ruta comprobando que en ninguna de las muestras había ningún obstáculo.

A continuación incluimos dos ilustraciones [Figura 5.6 y Figura 5.7] donde se aprecia significativamente la mejora llevada a cabo en nuestro algoritmo sobre un mapa real.

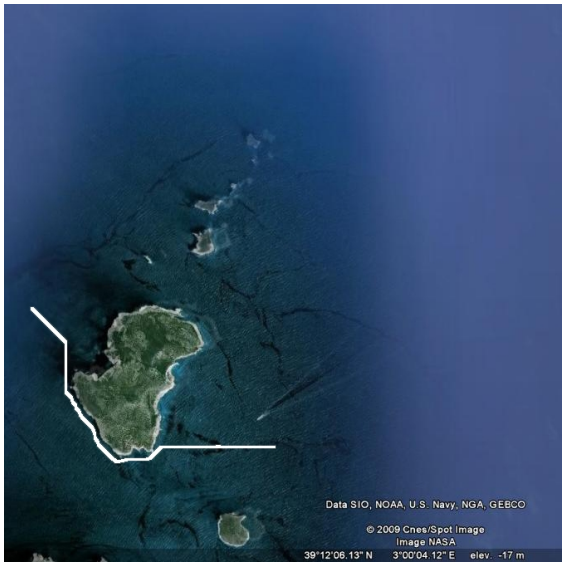



Figura 5.7: Ruta sin optimizar

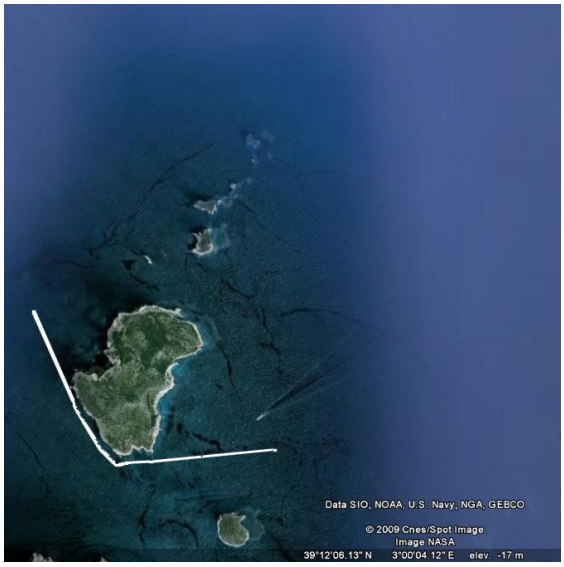



Figura 5.6: Ruta optimizada

Podemos ver que ambas trayectorias parten y acaban en el mismo punto. La Ilustración 11 se basa en giros de 45 ó 90 grados únicamente. En la Ilustración 12 apreciamos cómo hemos reducido el número de puntos intermedios de nuestra ruta, dando lugar a una trayectoria más corta y más realista.

## 5.4 - Relación con el modelo del barco

Con lo explicado hasta ahora, hemos obtenido la trayectoria óptima entre dos puntos. Ya tenemos el recorrido que debería hacer nuestro barco. Sin embargo, esta no es la trayectoria exacta que hacemos.

Ahora mismo, tenemos dividida nuestra ruta en varios puntos. Lo que hacemos es desplazar nuestro barco entre los dos primeros puntos. Para ello, nos ayudaremos de la clase

BoatAgent.java. Aquí, dando un punto de inicio y punto de llegada, nuestro modelo del barco intentará alcanzar el objetivo por medio de una línea recta.

Hemos de destacar que nuestro barco se maneja por medio de ángulos. Es decir, cuando intentamos ir de un punto inicio a punto fin, lo primero que realizamos es el cálculo del ángulo que forma la trayectoria con la horizontal. Una vez obtenido ese ángulo, nuestro barco comienza a girar desde la orientación en la que estaba hasta alcanzar el nuevo ángulo. Como ocurre en la realidad, este giro termina con una pequeña corrección de la trayectoria.

Esta orientación que precisa todo barco para girar, aparte de la simulación que hacemos de corrientes laterales, hace que nos salgamos ligeramente de la trayectoria idónea que le hemos indicado que realice. Lo que queremos indicar con esto es que nuestro barco a la hora de hacer los virajes, una vez ha girado mantendrá el rumbo en una dirección, y esta dirección estará ligeramente desplazada de la óptima. Este desplazamiento es prácticamente despreciable visto sobre un mapa o sobre Google Earth, pero a la hora de trabajar sobre coordenadas, tuvimos que aceptar que no alcanzaríamos exactamente cualquier punto intermedio o el punto objetivo.

Para solucionar este problema actuamos de la siguiente manera. Creamos un radio de visión para nuestro barco. Una vez que el punto destino se encuentra dentro de nuestro radio de visión damos por alcanzado el siguiente objetivo. Este radio de visión es pequeño, pero es necesario pues siempre tendremos un ligero desplazamiento en la trayectoria.

Es lógico pensar que la ruta óptima entre dos puntos se aproximará lo máximo posible a los obstáculos con el fin de bordearlos excediendo lo menos posible la trayectoria (Véase Ilustración 12). Y por otro lado, hemos explicado antes que a la hora de realizar ciertos giros, hay un pequeño desplazamiento en la trayectoria. Este desfase es pequeño pero existe.

Al pasar tan cerca de los obstáculos y tener este desplazamiento, es posible que nuestro barco *choque* sin querer contra una región de tierra. Para evitar esto, hemos añadido la posibilidad de crear un radio de seguridad para nuestro barco, con lo que nos aseguramos que posibles giros y cambios en la trayectoria no den lugar a colisiones con obstáculos. Para aligerar la computación de este asunto, en vez de comprobar en cada punto si guardábamos la distancia permitida, optamos por recubrir todos los obstáculos de una "corteza" de grosor igual al mencionado radio [Figura 5.8]. Posteriormente, ejecutamos nuestro algoritmo de la misma manera que antes pero sobre el nuevo mapa.

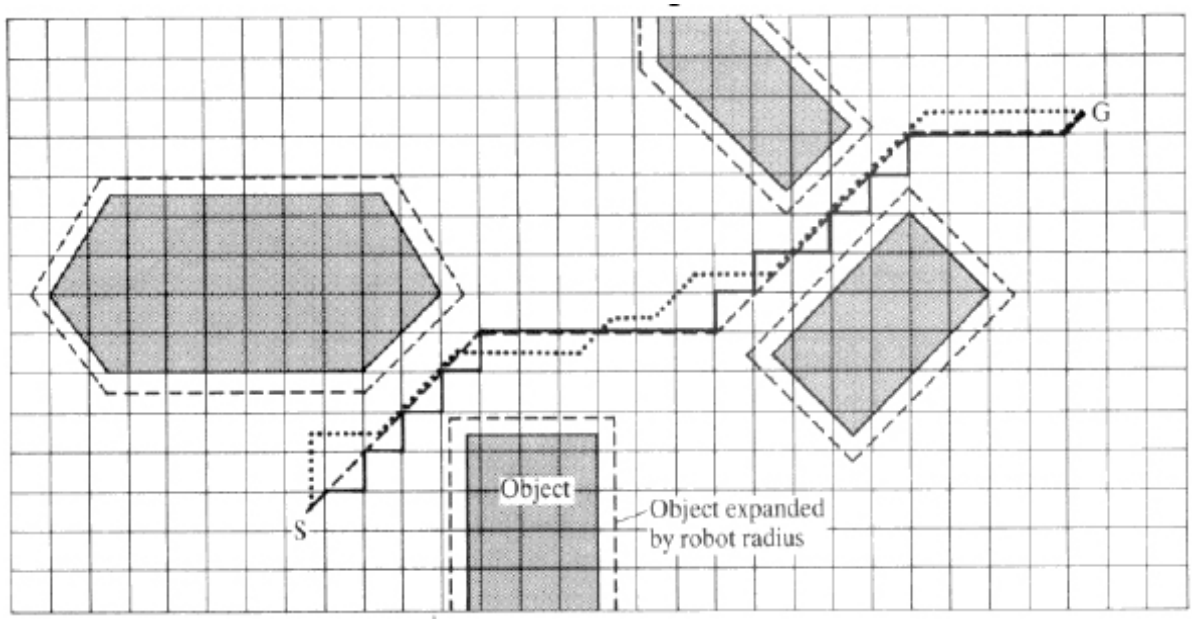



Figura 5.8: Ejemplo de recubrimiento de obstáculos con radio de seguridad

Queremos indicar que gracias a este método de simulación de un radio de seguridad o *corteza* que recubre todas las islas de nuestro mapa, se considera automáticamente dos islotes muy próximos entre sí como una única porción de tierra, lo que nos ayuda a evitar estas zonas de difícil acceso para barcos.

Vamos a resumir brevemente en qué sistemas hemos trabajado hasta llegar a este punto, el momento en que ya tenemos calculada la trayectoria que debe recorrer nuestro barco.

Hemos partido de una imagen JPEG, un mapa obtenido de Google Earth sobre el cuál ejecutaremos nuestra búsqueda *pathfinding*. Procesamos la imagen para obtener una

binarización de la misma y poder trabajar (Consultar capítulo 4).

Con este mapa de bits, aplicando nuestra versión del algoritmo A* logramos la ruta óptima hasta nuestro objetivo. Esta trayectoria se obtiene trabajando sobre una imagen, luego el resultado estará en función de **píxeles**. Vulgarmente podríamos decir que tenemos que "*movernos de este píxel a este otro"*.

Por supuesto, el modelo que migramos a JAVA de nuestro barco no trabajaba en píxeles, trabajaba en **metros**. He aquí la primera conversión que necesitamos: debemos poder convertir nuestra trayectoria de píxeles a metros.

Una vez obtenida nuestra trayectoria en metros, debemos simularla sobre Google Earth. El sistema utilizado por esta plataforma para plasmar coordenadas no es metros, sino **grados**. Encontramos aquí una segunda conversión entre dos sistemas de medida que debemos solventar.

Presentamos a continuación un detallado informe sobre los tres sistemas de representación usados en nuestro proyecto y cómo hemos trabajado sobre ellos.

## 5.5 - Sistemas de representación de un punto

Tenemos tres sistemas de representación distintos para cada punto. Vamos a pasar a estudiar cada uno de ellos con un poco más de detenimiento.

Para Google el planeta queda imaginariamente dividido en 4 partes, que, por comodidad vamos a llamar noreste, noroeste, sureste y suroeste. En cada una de esas divisiones, un punto, al que llamamos Coordenada en nuestro proyecto, está representado por 3 números o componentes:

•X : distancia a su origen este u oeste.

•Y : distancia a su origen norte o sur.

•Z : representa la altitud del punto respecto al nivel del mar, obviamente en nuestro proyecto

esta coordenada es nula en todo punto, por lo que nunca la tenemos en cuenta.

Esta división en 4 partes no importaría, centrándonos en territorio español, si no diera la casualidad de que España queda dividida. Aunque Google reconoce el paso de un lado al otro sin problemas, los algoritmos no están preparados a este cambio, por considerarse no necesario para los objetivos principales del proyecto.

### 5.5.1 - Posición en una matriz

Añadir el formato imagen que mencionamos anteriormente suponía tener que buscar una manera de relacionar cualquier punto de esta imagen con su correspondiente punto sobre el planeta en Google y esto no resultó trivial:

Inicialmente sabemos que cada uno de los archivos KML que tenemos guardan información sobre la coordenada central del mapa (al añadir una chincheta se coloca por defecto en este punto), así como de la altitud desde la que se creó.

Cogimos un mapa ejemplo en Google de la que calculamos el espacio en grados que se podía observar tanto a lo alto como a lo ancho, estos datos, así como la amplitud desde la que fue medido están guardados en el proyecto como constantes.

Con esta información asumimos que, si desde esa altitud (a la que vamos a llamar rango), se podían ver X grados a lo largo e Y a lo alto, con cualquier otra alcanzaríamos a ver "((X * altitud)/rango)" grados a la ancho y análogo para el espacio representado de norte a sur.

Por otro lado la imagen en formato jpg, puede ser convertida, como se explicará en el apartado de tratamiento de la imagen, a una matriz de pixeles. Cada pixel de la imagen corresponde a una fila y una columna de la matriz.

Obviamente cada posición de la matriz está modificando la posición en el mapa en un valor obtenible dividiendo el espacio en grados entre el número de filas/columnas de la matriz. La esquina superior izquierda tendrá las coordenadas obtenidas tras restar o sumar (según la parte del planeta en la que estemos) al valor del punto central, la mitad de las filas y columnas de la matriz por lo que supone en coordenadas la "ascensión" por cada uno de esas filas/columnas.

Con todos estos datos, cualquier punto de la imagen debería poder ser identificable con un punto en Google y viceversa.

### 5.5.2 - Metros

Otro sistema de representación a tener en cuenta es el del modelo del movimiento del barco, en todas las ecuaciones, todo movimiento está en metros. Para relacionar el modelo con el resto del proyecto hay que convertir esos metros a grados. La idea con la que realizamos el cálculo fue la siguiente:

Una "circunferencia" de la tierra mide 46190 Km, es decir 46190000m, lo que en grados, más intuitivamente, son 360. Por tanto cada grado son 46190000 / 360 metros.

Hay generados métodos tanto para una conversión como para la otra.

### 5.5.3 - Grados

En cuanto al sistema métrico utilizado, Google permite distintas opciones, como puedan ser kilómetros, millas, pulgadas o grados, pero el sistema al que otorga mayor importancia es a la representación mediante grados, ya que independientemente de la elección que tomemos en este sentido, el número siempre queda guardado como la conversión a decimal de los grados minutos y segundos que definan al punto. Es este el valor que también tomaremos nosotros como principal.

Esta conversión a decimal es muy sencilla: un grado tiene 60 minutos y un minuto tiene 60 segundos, por tanto, un grado tiene 3600 segundos. Si un punto está definido como xº, y',z" su número decimal asociado será:

$$d = x + \frac{y}{60} + \frac{z}{3600}$$

Pero para poder llevar a cabo este proyecto, no se puede trabajar sólo con este sistema de medida; esto es sólo una simple representación.

# Capítulo 6 -

# Plataforma de simulación

La plataforma de simulación necesaria para la ejecución de este proyecto así como la interfaz gráfica de usuario (GUI) ha sido creada mediante la herramienta de desarrollo NetBeans. Al ser esta la primera versión de Dolphin, las pantallas se han creado pensando en facilitar ampliaciones y modificaciones de las que hablaremos más adelante.

Para ver un poco qué interfaces se han creado y cómo deben utilizarse, vamos a ir una a una hablando de sus posibilidades. Esto será útil también para un usuario no habituado al uso del sistema ya que se podría considerar este capítulo como un manual de uso.

# 6.1 - Pantalla principal

La primera pantalla que aparece al ejecutar la aplicación será esta [Figura 6.1]. Es una simple presentación en la que podremos decidir salir o ejectuar el simulador.


Figura 6.1: Pantalla inicial de Dolphin

Seleccionando la opción de ejecutar, accederemos a la siguiente pantalla:

# 6.2 - Generación de escenarios

Esta será la pantalla [Figura 6.2] desde la que se crearán las diferentes misiones sobre las que simularemos el comportamiento de los barcos. Como hemos dicho, ésta se abre trar decidir ejectuar el proyecto, pero se puede volver a acceder a ella desde la siguiente pantalla para crear en cualquier mometo cuantas misiones se deseen.

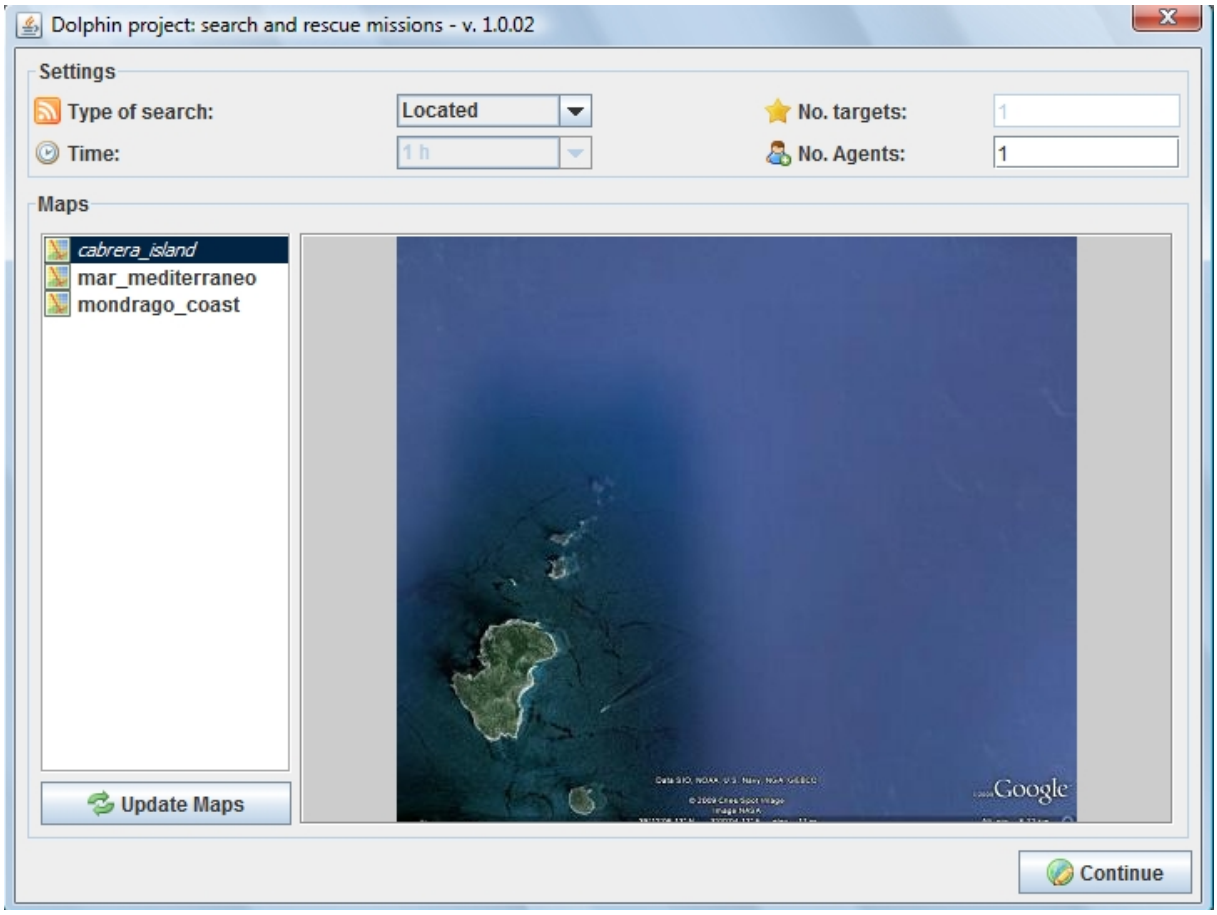



Figura 6.2: Pantalla de creación de escenarios

Como vemos en la imagen, podremos aquí elegir qué tipo de misión vamos a realizar. En esta versión de Dolphin sólo está implementado "Located"; búsqueda del trayecto óptimo de un punto a otro pasando por posibles metas intermedias. La herramienta queda preparada para otros tipos de búsqueda como por ejemplo el barrido de un área. Esta funcionalidad implementada es la que hemos considerado básica para facilitar la ampliación con cualquier otro tipo de búsqueda, ya que en cualquier caso sólo habría que combinar de manera adecuada los distintos módulos que hemos generado y añadir las funciones específicas.

Las opciones que podemos ver en la interfaz no están accesibles para esta opción, pero se activan cuando seleccionamos una búsqueda de tipo "área". Vamos a hablar un poco de esta opción:

En el proyecto anterior a este se trabajó se plateó que los aviones recorrieran de manera óptima distintas áreas que el usuario introducía manualmente. Este objetivo se

solucionó, aportando un algoritmo de recorrido óptimo. Para ampliar este trabajo, pensamos que una posible mejora sería que en lugar de introducir manualmente el área de probabilidad de localización de los objetivos, nuestros vehículos lo calculasen en función de diversos datos.

Para ello estudiamos las variables que podían afectar a la posición de un cuerpo inanimado al caer en un punto cualquiera de la superficie marítima. Simplificando mucho resumimos que necesitábamos conocer, además de un punto aproximado de origen, las fuerzas de las corrientes, el tiempo transcurrido desde la "caída", y la superficie geográfica que envuelve el suceso.

Además se incorpora el número de objetivos a localizar, permitiendo con esto que el barco "sepa" que puede finalizar la búsqueda aunque no haya terminado de recorrer el área. Esto supone entrar en un problema muy interesante que planteaba ya en sus documentos el profesor Jesús M de la Cruz, vamos a trasladarla ahora a esta documentación:

> *"Suponemos un conjunto de objetos distribuidos aleatoriamente en una superficie perfectamente delimitada del océano, por ejemplo, un hectómetro cuadrado.*
>
> *Allí cada elemento se ve sometido a las fuerzas del viento (oleaje) y de las corrientes, lo que hace que se vayan dispersando los objetos. Transcurrido un cierto tiempo se inicia la búsqueda de los objetos por un UAV. Este posee un elemento detector que permite la localización de objetos que se encuentren dentro de una circunferencia centrada en el UAV de, por ejemplo, un kilómetro de radio. También permite determinar como se va desplazando el objeto.*
>
> *¿Qué trayectoria debe seguir el UAV para encontrar el máximo número de objetos en el menor tiempo posible? ¿Cómo debe modificar su trayectoria para mejorar la búsqueda si encuentra en su campo de visión un nuevo objeto y mide su posición y trayectoria?"*

Este es uno de los desarrollos que no hemos podido continuar por las lógicas limitaciones de tiempo, pero que creemos que merece la pena mencionar.

Estas opciones que hemos decidido dejar no funcionales en nuestra herramienta, buscamos que sean una idea o ayuda en caso de que se trabaje en una continuación a este proyecto. Creemos que sería muy interesante lograr una combinación del proyecto de agentes aéreos con este de agentes marítimos; y en este caso, estas opciones llegarían a ser funcionales a partir de una generalización de los algoritmos del otro proyecto, añadiendo a éste las mejoras de las que hemos hablado ahora.

Continuando con nuestro proyecto, y en concreto con esta pantalla, vemos que también existe la opción de introducir un número de agentes, el objetivo final de esta variable es que refleje el número de barcos disponibles para la búsqueda y conseguir que estos viajen en formación.

Actualmente se puede decidir para cada barco de cada cada misión una trayectoria independiente de las de los demás, pero la herramienta está preparada para la adaptación a la cooperación entre estos para llevar a cabo una tarea común de manera distribuida.

Otra campo que podremos modificar será el relativo al nombre de la misión, que deberá ser único en cada instancia del programa de simulación.

Tras estas opciones podemos ver la lista de mapas actualmente disponibles. Que un mapa sea disponible supone que en la carpeta correspondiente ( “resources/maps”) existan una imagen en extensión .jpg y su correspondiente archivo .kml. Esta lista de mapas se puede actualizar seleccionando la opción “Update Maps” que vemos en la parte inferior de la interfaz. Esta opción recorre la carpeta mencionada mostrándonos en la lista superior todos los mapas disponibles.

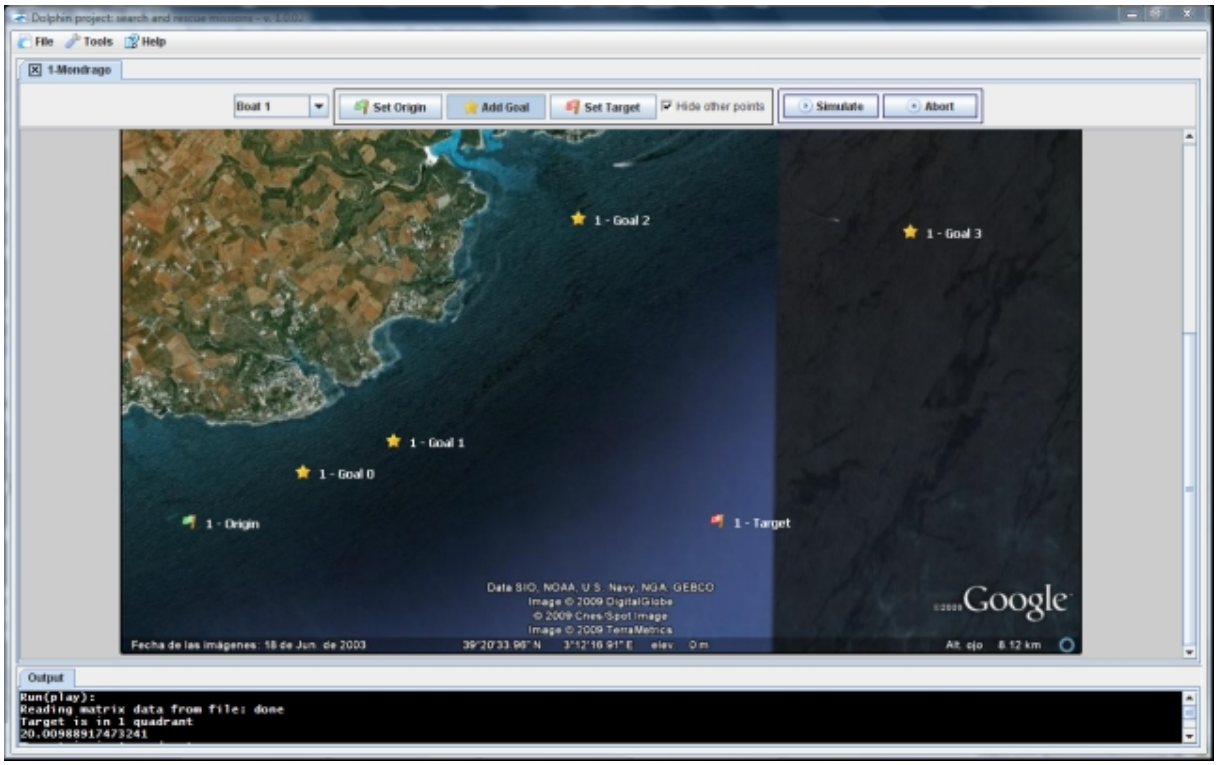
La opción “Continue” sólo se habilita una vez seleccionado algún mapa, al pulsarlo habremos creado la misión y pasaremos a la siguiente pantalla. En las primeras fases de desarrollo de este proyecto esta opción daba paso a la ejecución del programa Google Earth desde el que se seleccionaban los puntos origen y destino, pero tal y como se comentó cuando hablamos de esta herramienta de Google, esto planteó una serie de problemas que

desembocaron en un cambio en la manera de captación de dichos puntos.

El mapa que vemos en la imagen es la previsualización del mapa seleccionado, y su única finalidad es que la selección del mapa sea más fácil, siendo realmente útil cuando la lista de mapas disponibles sea un número grande.

## 6.3 - Diseño de Misiones actuales

Esta será la pantalla principal gracias a la cual podremos determinar para cada barco de cada misión cuál será su cometido en la simulación. Mediante los botones "Set Origin", "Add Goal" y "Set Target", podremos establecer un origen, una serie de metas intermedias, y un destino para cada Barco respectivamente. Si se marca la opción "Hide other points", los puntos marcados visibles sólo serán los relativos al barco actual seleccionado, facilitando así la creación de rutas que mantengan independecia respecto de otras.




Como hemos comentado antes, a esta pantalla se accede mediante la creación de una

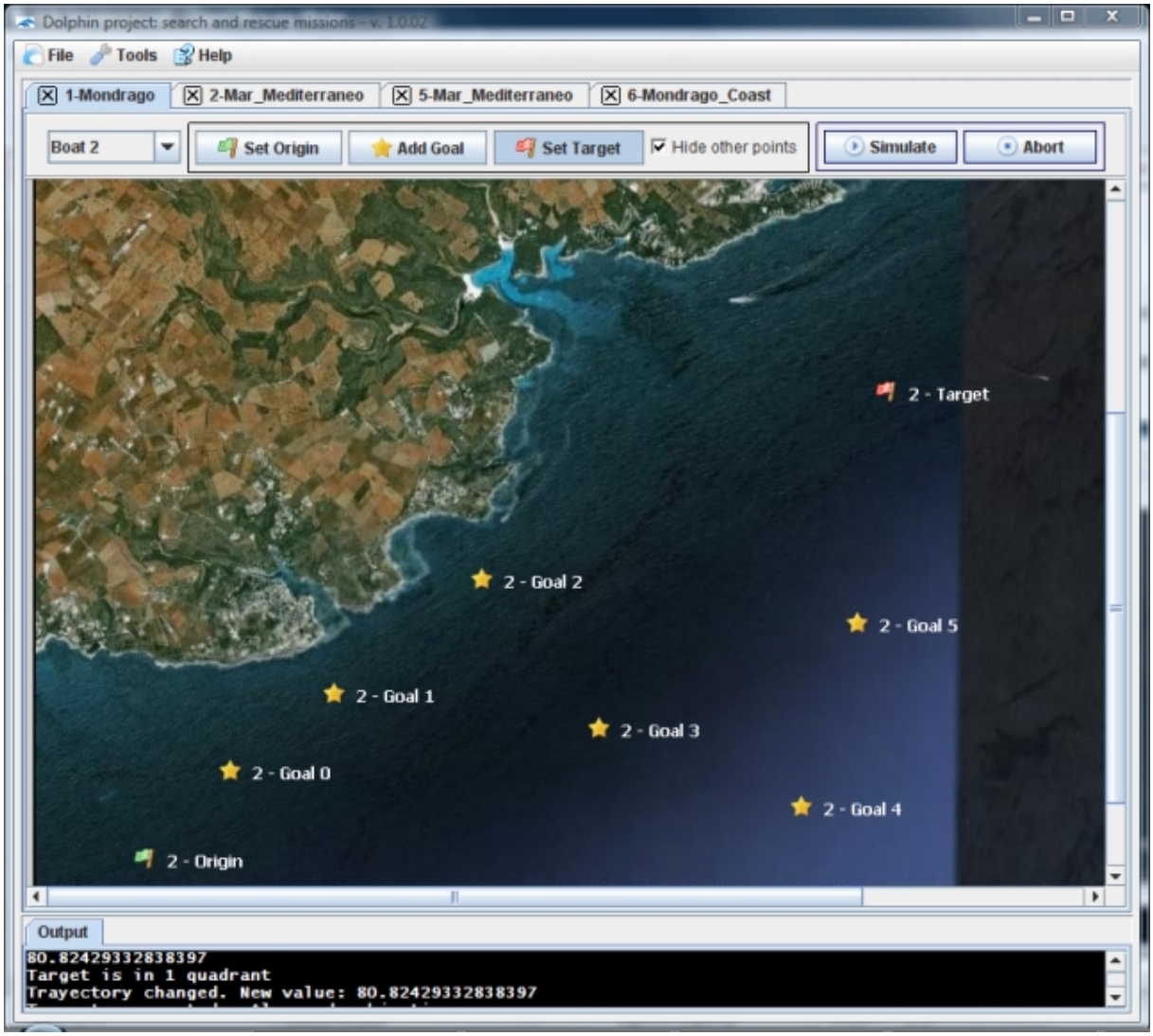



Figura 6.3: Pantalla principal del diseño de misiones

misión mediante la selección de un mapa y su configuración en la inerfaz anterior. Si se desea abrir otro mapa o se ha cometido un error, desde la opción: Tools → Create Mission, se accede a la pantalla de selección de mapas desde la que se podrá arreglar el error o modificar cualquier detalle de la búsqueda. En caso de tomar esta opción, al volver a esta pantalla mantendremos la búsqueda anterior y la nueva en distintas pestañas con las que podemos trabajar o eliminar de manera independiente, tal y como observar en la Figura 6.3.

Tras la pestaña con el nombre del mapa para nuestra búsqueda vemos una serie de botones. Ya hemos hablado de los que determinarán los puntos de trayectoria de cada barco pero vamos a  comentarlos más a fondo junto al resto de ellos.

A la izquierda del todo está el elemento visual que nos va a permitir seleccionar el agente (barco) con el que queremos trabajar [Figura 6.4].

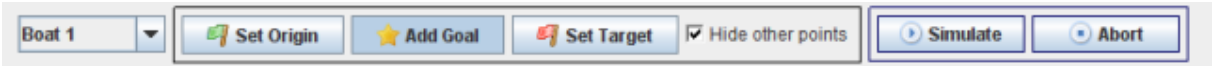



Figura 6.4: Botones para selección de barco, objetivos y simulación

Las opciones "Set Origin" y "Set Target" servirán para marcar en el mapa los puntos Origen y Destino. Una vez seleccionado cualquiera de ellos podremos cambiarlo simplemente volviendo a hacer un click en el nuevo punto deseado. El origen quedará marcado con una bandera verde y el destino con una roja.

Dado que puede ser interesante plantear para un barco un recorrido que pase por diferentes puntos, hemos añadido la opción de poder seleccionar varios hitos intermedios y anteriores al destino por los que deberá pasar el barco. Para marcarlos será necesario hacer click en el botón "Add Goal" y a continuación en el mapa tantas veces como puntos intermedios se desee insertar.

Se sobreentiende que todos estos puntos se situarán en el mar. Consideramos también que en una distancia muy cercana a la costa un barco del tamaño que estamos manejando no puede navegar; por lo que si alguno de ellos está mal colocado, tendremos que volver a situarlo en un lugar correcto.

No poder acercar mucho el barco a la costa tiene varios sentidos: si el objetivo esta cercano a la costa, entenderemos que son los equipos de tierra los que podrán localizarlo, con lo que no tendría sentido enviar un barco tan grande. Si es el barco el que se sitúa muy cercano, al no tener información sobre la profundidad no podemos garantizar que navegue con normalidad. Otra característica de esta decisión es que los pequeños archipiélagos son considerados como un bloque si están tan juntos que añadiéndoles este "margen de seguridad" se solapan. Si pudiera parecer que esto resta eficacia al sistema, hay que tener en cuenta que el radio de visión del barco haría que localizase el objetivo, aunque no llegar hasta él.

Hablemos ahora de los botones "Simulate" y "Abort":

Mediante el botón “Simulate”, el centro de control (nuestra máquina), le enviará un mensaje al agente barco correspondiente en el que le indicará cuál va a ser su misión. El barco calculará la ruta óptima y procederá en un segundo plano a la realización de la misma. Este es el grueso de la simulación de la trayectoria.

Con el botón “Abort” enviaremos un mensaje al Barco indicándole que aborte la misión.

Todos los mensajes que envíe el barco a la base informando de sus acciones y estado, podremos verlos en el recuadro negro situado en la parte inferior de la pantalla; tal y como podemos observar en la imagen que comentamos.

## 6.4 - Información de los barcos actuales (agentes)

Si desde la pantalla anterior procedemos r a pulsandr en Tools → Boat Info, accederemos a la siguiente interfaz [Figura 6.5]:

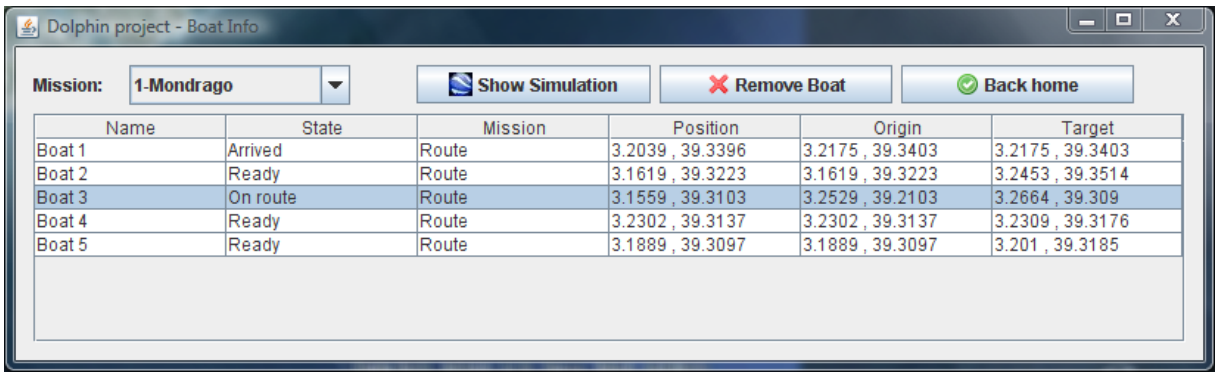



Figura 6.5: Pantalla de información de Agentes

Aquí podemos ver para cada barco de cada misión toda la información relevante. Los mensajes que envién los agentes informando de su posición, misión adoptada y estado, serán recibidos por la base y procesados para la actualización en todo momento de dicha tabla.

Desde aquí podremos además comunicarnos con los barcos.

Mediante la opción “Remove Boat” podremos eliminar el barco seleccionado de la

misión escogida, desvinculándolo así de la base de agentes. Desde ese momento no podremos comunicarnos con él ni asignarle nuevas misiones.

Disponemos también del botón "Back home"; esto simplemente enviará un mensaje al barco seleccionado indicándole que puede regresar a su base.

Pulsando el botón "Show Simulation", abriremos el Google Earth y cargaremos el kml que ha generado cada agente con la información de su ruta actual. Este será el momento en que podremos ver la simulación del trayecto planificado y realizado por el agente que controla en barco.

## 6.5 - Resultado de la simulación

Veamos ahora un ejemplo de trayectos calculados para uno de los mapas que actualmente está cargado en el simulador. Estos han sido calculados con los parámetros de entrada que se observan en la siguiente imagen [Figura 6.6].


Figura 6.6: Mapa con datos de entrada para simulación

Este es el resultado observable desde Google Earth [Figura 6.7].


Figura 6.7: Resultado para la simulación de los datos anteriores

# Capítulo 7 -

# Conclusiones y ampliaciones

Pasaremos ahora a comentar las conclusiones finales deducibles de la implementación de este sistema de simulación, y hablaremos de posibles futuras ampliaciones.

## 7.1 - Conclusiones

Se ha desarrollado una herramienta muy completa que integra:

- Modelo de la dinámica de un barco teniendo en cuenta corrientes marinas.
- Representación de los vehículos marítimos como agentes inteligentes, que les permite estar a la espera de ordenes, acatarlas y comunicarse con una estación base.
- Algoritmos de trayectoria muy sólidos que calculan recorridos óptimos para el desplazamiento del barco. Una aplicación puede ser que le permiten

esquivar obstáculos no sólo una vez encontrado sino al iniciar la trayectoria gracias a la información propia del agente.

•Un sistema de tratamiento de imágenes perfectamente capaz de diferenciar tierra de mar en cualquier imagen aportada.

•Ejecución inmediata sobre Google Earth para visualización de todos los recorridos.

•Interfaces muy simples pero muy completas que facilitan el correcto uso de la aplicación a usuarios no familiarizados con ella.

## 7.2 - Ampliaciones

•Dolphin comenzo como un proyecto asociado a Seagull, por tanto las ampliaciones que se nos plantean con este marco común son muchas, pasamos a comentar las que hemos barajado y las que han ido surgiendo como ideas a lo largo de toda la ejecución del proyecto.

•Unión con Seagull adaptando las comunicaciones a nuestros agentes

•Navegación en formación (ya implementada en Seagull). Aquí se plantean problemas de difícil solución como las perturbaciones generadas por los barcos en las aguas y su difícil modelaje.

•Adaptar los algoritmos ya implementados para que la navegación sea válida también para los tres cuadrantes restantes en los que los creadores de Google Earth dividieron la tierra.

•El modelo y todas sus aplicaciones están preparadas para trabajar con corrientes de mar variables, en esta primera versión se han tomado constantes. Una posible ampliación sería la adaptación de corrientes en tiempo real.