Plataforma de simulación para búsqueda y salvamento en alta mar

Autores:
Sergio González Sanz
David Rodilla Rodríguez
Carlos Sánchez García

Profesor director:
Jesús Manuel de la Cruz García

Proyecto de Sistemas Informáticos
Facultad de Informática
Universidad Complutense de Madrid

INDICE

Agradecimientos 5
Resumen del proyecto 6
Introducción al proyecto 7
  Organización de la memoria 9
  Problema 10
  Introducción a DEVS 11
    Especificación de un sistema DEVS 12
  XDevs 14
  Fases del proyecto 15
    Lista de tareas 15
    Evolución de las tareas: Diagrama de Gantt 16
  Evolución de los prototipos creados 17
    Prototipo 2D 17
    Visor 3D empleando Java-3D 18
Tema 1: Modelo simplificado de avión 22
  1.1. Modelo en vuelo horizontal 22
  1.2. Modelo de cambio de velocidad 26
  1.3. Movimiento vertical 28
  1.4. Composición de los movimientos horizontal y vertical 30
  1.5. Valores iniciales 30
  1.6. Entradas y salidas del modelo 31
    1.6.1. Entradas 31
    1.6.2. Salidas 32
  1.7. Control de rumbo 33
  1.8. Implementación del Avión 36
    1.8.1. AvionState 38
    1.8.2. ControladorRumboState 40
Tema 2. Modelo simplificado de barco 43
  2.1. Control del barco 45
  2.2. Implementación del barco 46
    2.2.1. Modelo de referencia 49
    2.2.2. Barco 51
    2.2.3. Controlador Rumbo 53
    2.2.4. Controlador Posición 55
Tema 3: Pruebas de control de rumbo de los vehículos 57
  3.1. Rumbo en cuadrado 57
  3.2. Rumbo en triángulo 58
  3.3. Rumbo en cruz 59
  3.4. Ruta en ocho 60
  3.5. Ruta en estrella 61
Tema 4: Modelos de los componentes 62
  4.1. Modelo del viento y oleaje 62
  4.2. Modelo de corrientes 63
  4.3. Modelo e implementación de los náufragos 64
  4.4. Mar 67

Tema 5: Otros modelos Devs 68
5.1. Mux 68
5.2. Operador 69
Tema 6: Integración numérica de los modelos 70
6.1. Método numérico de Euler 72
6.2. Método numérico de Runge-Kutta 73
Tema 7: Estructura de la simulación 74
7.1. Introducción 74
7.2. Esquema de la simulación 75
7.3. Controlador 76
Tema 8: Algoritmos de búsqueda de náufragos 77
8.1. Introducción 77
8.2. Estudio previo 78
8.2.1. Algoritmo 78
8.2.2. Viabilidad 79
8.2.3. Alcance 79
8.3. Simulaciones 79
8.4. Algoritmo 81
8.4.1. Espacio de búsqueda 81
8.4.2. Estructura de los individuos 83
8.4.3. Primera generación 84
8.4.4. Elección de los padres 85
8.4.5. Crossover 85
8.4.6. Mutación 86
8.4.7. Resultado final 86
8.4.8 Esquema 87
8.5. Diagrama UML 88
8.6. Pruebas 88
8.6.1. Simulación 1 89
Tema 9: Aprendizaje 96
9.1. Método de máxima verosimilitud 96
9.2. Implementación del aprendizaje 98
9.3. Ejemplo de aprendizaje 100
Tema 10: Interfaz de simulación 105
10.1. Introducción 105
10.2. Diseño XDEVS 106
10.3. Diseño y estructuración del manager de la vista 108
10.4. Diseño y estructuración de la interfaz 2D 110
10.5. Diseño y estructuración de la interfaz 3D 113
10.5.1. Diseño de la cámara 3D 116
10.6. Jerarquía de objetos 3D 118
10.6.1. TAfin 119
10.6.2. Color 119
10.6.3. Objeto3D 120
10.6.4. PV3D 120
10.6.5. VerticeNormal 121
10.6.6. Cara 121
10.6.7. Malla 121

10.6.8. ObjetoCompuesto3D 122
10.7. Visualización de los datos de las simulaciones 123
10.7.1. Ventana de visualización de probabilidades 123
10.7.2. Ventana de visualización de parámetros 124
Tema 11: Creación de terrenos y escenarios 125
11.1. Algoritmos de generación de terreno 128
11.1.1. Algoritmo de generación de terrenos de montaña y mar 128
11.1.2. Algoritmo de generación de terrenos de costa 128
11.2. Algoritmo de dibujo del terreno. 129
11.3. Diseño de la construcción de terrenos 130
Tema 12: Simulación en entornos reales: Google Earth 132
Tema 13: Integración de las partes 135
Apéndice A: Memoria XML 137
Introducción 137
Formato del documento XML 137
Apéndice B: Formato de los ficheros .kml 139
Encabezado de los ficheros .kml 139
Localización de posiciones (Placemarks) 139
Capas (GroundOverlays) 140
Caminos (Paths) 140
Polígonos (Polygons) 141
Estilos (Style) 141
Operaciones con la camara 142
Manejo del tiempo 143
Palabras clave para su búsqueda bibliográfica 144
Bibliografía 145
Autorización a la difusión 146

Agradecimientos

En primer lugar queremos agradecer la inestimable ayuda prestada por nuestro profesor director, Jesús Manuel de la Cruz sin la cuál la realización de este proyecto no hubiese sido posible.

También queremos agradecer la ayuda prestada por los profesores José Luís Risco Martín y Segundo Esteban San Román que tan amablemente han atendido nuestras dudas y que nos han prestado su ayuda, así como a Gonzalo Pajares Martinsanz que nos puso en contacto con Jesús Manuel de la Cruz.

Por último, agradecer a todo aquel que dedique su tiempo a revisar este proyecto, deseándole que le sea útil y le sirva de ayuda.

Resumen del proyecto

El objetivo del proyecto es crear escenarios en los que realizar simulaciones y planificación de tareas de salvamento y rescate en el mar.

Las simulaciones serán en mar abierto, disponiendo de tres tipos de móviles (aviones, barcos y náufragos), donde el objetivo será rescatar al máximo número de náufragos en el menor tiempo posible.

Se han creado modelos de vehículos autónomos aéreos y marinos y del movimiento de los náufragos mediante la tecnología xDevs. El movimiento de los náufragos es estocástico (influido por las corrientes marinas y el viento). Se han implementado algoritmos de control para los aviones y los barcos para que sigan las trayectorias de los algoritmos de planificación.

Los algoritmos de búsqueda de náufragos implementados por los aviones y los barcos son genéticos con aprendizaje en función de la velocidad y el rumbo medio de los náufragos.

La implementación gráfica del escenario, así como de los vehículos esta realizada en OpenGL y se pueden visualizar las simulaciones sobre escenarios reales con Google Earth.

The aim of this project consists of the creation of scenes in which to be able to realize simulations and planning of rescue tasks in the sea.

The simulations will be on the open sea, having three types of mobiles (planes, ships and shipwrecked persons), where the aim will be to rescue to the maximum number of shipwrecked on the least time posible.

Models of autonomous aereal and marine vehicles and have been created through xDevs technology. The movement of the shipwrecked is stochastic (influenced by the sea currents and wind). Control algorithms for aircraft and ships have been implemented to follow the trajectories of the planning algorithms.

Shipwrecked search algorithms implemented by planes and ships are genetic with learning depending on the speed and heading of the shipwrecked.

The graphic implementation of the scenes, this way like of the vehicles are realized in OpenGL and the simulations can be visualized on real stages with Google Earth.

Introducción al proyecto

El problema al que nos enfrentamos al iniciar el proyecto consiste en la creación de escenarios en los que realizar simulaciones y tareas de planificación de rescate y salvamento en el mar. Para ello, pretendemos crear una herramienta que permita de forma sencilla y fácil, la creación de estas simulaciones, así como la visualización de sus resultados, fundamental para la validación de las técnicas utilizadas, así como de los resultados obtenidos.

Las simulaciones creadas son en mar abierto, puesto que se trata de rescate de náufragos procedentes de un accidente aéreo en el mar o marítimo, donde las tareas de planificación son fundamentales a la hora de proceder al rescate, puesto que el tiempo resulta una variable fundamental para conseguir rescatar con vida al número máximo de náufragos posible.

Los vehículos que intervienen en este tipo de rescates son tanto aéreos como marinos, por lo que hemos procedido a la creación de modelos para representar este tipo de vehículos. Para ello, hemos utilizado herramientas como Matlab y Simulink para obtener modelos de los mismos, así como los algoritmos de control propios para conseguir que los vehículos creados sigan las rutas generadas por los algoritmos de planificación y de rescate. Para la creación de los distintos modelos, hemos utilizado la herramienta xDevs, que nos permite la implementación del paradigma de simulación DEVS sobre Java. Dado que el movimiento de los náufragos depende del movimiento de las mareas y de los viento, hemos implementado algoritmos estocásticos para simular su movimiento en función de estos parámetros.

Para poder rescatar al mayor número de náufragos en las simulaciones, hemos optado por la implementación de algoritmos genéticos para el cálculo de las rutas óptimas para los distintos vehículos, en los que están implementados mecanismos de aprendizaje, los cuales tienen en cuenta la posición de los náufragos encontrados para calcular la posición de los náufragos restantes. Para ello, se tiene en cuenta el rumbo y la velocidad que han tenido que seguir los náufragos encontrados para llegar a esa posición para poder aprender estas variables y poder calcular las mismas para los restantes náufragos.

La implementación gráfica de los escenarios de simulación se ha creado con OpenGL, una librería gráfica de libre distribución para Java. Se han creado dos vistas distintas, una vista en 3D y otra en 2D, que permiten un manejo fácil y sencillo de las simulaciones, permitiendo en cada momento la visualización de cada uno de los elementos de las simulaciones, así como su seguimiento por los escenarios. Además se visualizan los parámetros de las simulaciones para poder comprobar su corrección y poder valorar los resultados obtenidos.

Además, para poder visualizar las simulaciones sobre escenarios reales, se ha trabajo con la herramienta Google Earth y se ha hecho una extensión del proyecto que permite seguir los resultados de las simulaciones a la vez que se están realizando en la interfaz propia, sobre Google Earth. Esto permite llevar al proyecto a una nueva dimensión, gracias a que Google Earth implementa multitud de herramientas de georeferenciación y de visualización sobre entornos reales, con las ventajas que eso significa a la hora de planificar un rescate real. Además, actualmente Google Earth está teniendo una gran difusión en todos los ámbitos, en especial para la creación de aplicaciones que requieren el posicionamiento de vehículos en un entorno real, como es nuestro proyecto.

Con todo, hemos creado una herramienta completa que permite crear simulaciones reales de rescate y salvamento en el mar, a través de la tecnología xDevs.

Organización de la memoria

En esta introducción, en primer lugar presentamos el problema al que nos enfrentamos al afrontar este proyecto. Tras ello, procedemos a hacer una introducción a la tecnología DEVS, utilizada en el proyecto para construir los modelos de los diferentes componentes de las simulaciones. Presentamos en esta introducción también las fases desarrolladas dentro de este proyecto, así como diferentes prototipos creados durante la fase de desarrollo y que finalmente evolucionaron hacia otros prototipos más completos.

Tras la introducción, los temas 1 y 2 presentan el estudio y la construcción de los aviones y barcos de las simulaciones. El tema 3 se encarga de presentar una serie de pruebas para comprobar el correcto funcionamiento de estos vehículos.

El tema 4 presenta el estudio y la creación de modelos estocásticos para las simulaciones, como son los modelos de las olas o del mar y de los náufragos.

La creación de otros modelos necesarios para las simulaciones, como son el multiplexor o un modelo de operador binario parametrizable son tratadas en el tema 5.

En el tema 6 presenta el modelo de integración numérica implementado para la evolución en el tiempo de los elementos de la simulación. Por su parte, el tema 7 presenta la unión de todos los elementos necesarios para la simulación y su operación conjunta.

El tema 8 explica el algoritmo implementado por los vehículos de la simulación para la búsqueda de los náufragos objetos del proyecto. El tema 9 presenta el mecanismo de aprendizaje implementado en los vehículos.

La interfaz gráfica de la simulación se explica en el tema 10, mientras que la construcción de los terrenos de simulación y su visualización es tratada en el tema 11.

Por último en el tema 12 se explica el diseño realizado que permite la visualización de los resultados de la simulación en tiempo real sobre la herramienta Google Earth.

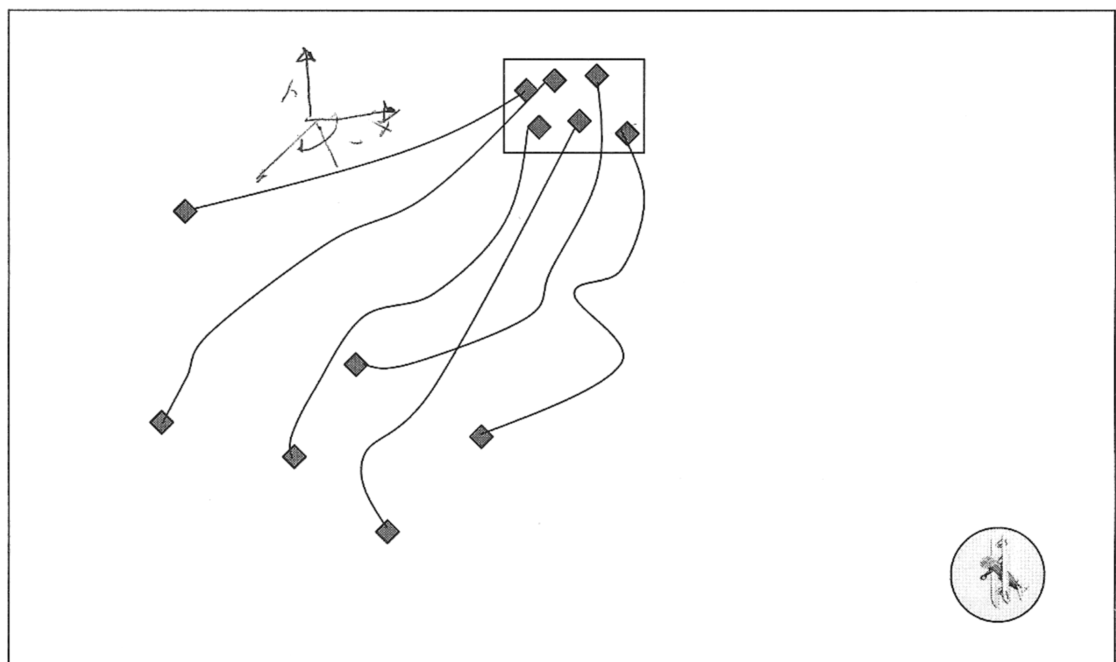
Además se incluyen dos apéndices: el primero dedicado a la construcción de sistema para transformar archivos XML a elementos xDevs y viceversa, el segundo de ellos, al formato de los ficheros KML necesarios para visualizar los resultados de las simulaciones sobre Google Earth.

Problema

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 de objetos en el menor 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?



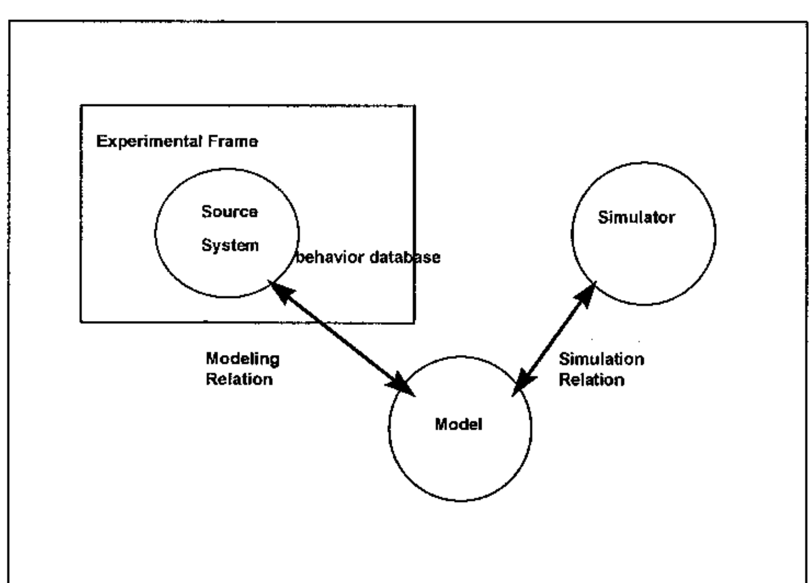
Introducción a DEVS

El formalismo Devs (Discrete Event Sistems) fue formulado por Bernard P. Zeigler en 1975.

EL formalismo Devs proporciona una manera matemática de especificar un objeto al que denominamos sistema y de simular su comportamiento en un entorno que le proporcionaremos.

En un momento dado, podemos caracterizar a un sistema por su estado, es decir, las características que hacen que ese sistema se comporte de una manera u otra en nuestra simulación, dependiendo de en qué aspectos nos estemos fijando, probablemente nos interese un estado más simple con menos características o uno más complejo, con más características del modelo.

El framework conceptual DEVS permite fácilmente separar el modelo que vamos a simular, de dónde lo vamos a simular (simulador).



Modelo: Que es un conjunto de instrucciones capaces de generar datos comparables al sistema real observable.

Simulador: Aquello que ejecuta las instrucciones del modelo, dándole un comportamiento y haciéndolo evolucionar en el tiempo.

Experimental Frame: Captura los resultados de la simulación.

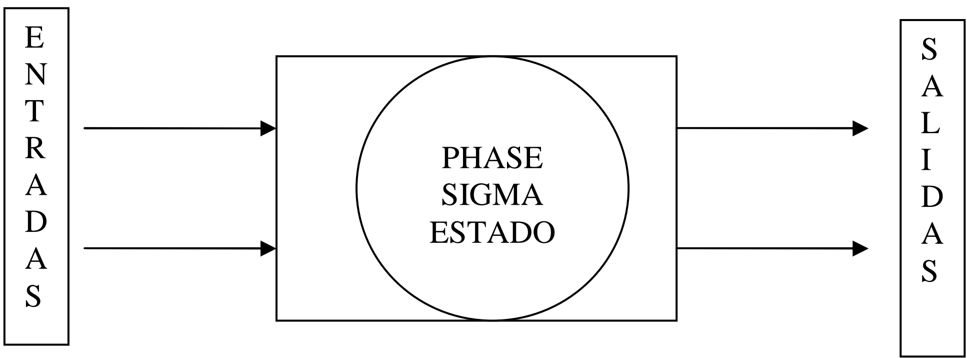
La estructura de un modelo debe estar representada en un lenguaje matemático llamado formalismo. Un formalismo debe definir como generar nuevos valores de un modelo y cuando generarlos.

Las principales ventajas de Devs, son que proporciona una gran modularidad a los modelos, debido a que los modelos pueden ser vistos, como veremos a continuación como cajas negras muy fáciles de manejar.

Además separa completamente los modelos del simulador, lo que hace que podamos ejecutar el simulador en una máquina o de forma distribuida de una forma muy sencilla.

Especificación de un sistema DEVS

Un sistema devs cuenta con una tupla:
$M = < X, S, Y, \delta_{int}, \delta_{ext}, \lambda, ta>$
X es el conjunto de entradas.
S es el conjunto de estados.
Y es el conjunto de salidas.
δ_{int}: S→ S - es la función de transición interna.
δ_{ext} : Q x X → S - es la función de transición externa.
λ: S→Y - es la función de salida.



<u>Entradas</u>: Valores con los que el modelo toma datos del exterior (otros modelos).

<u>Salidas</u>: Valores que el modelo da al exterior.

<u>Phase</u>: Estado actual del modelo, por ejemplo, en el caso del avión podría ser si esta sin inicializar, creado o destruido.

Estado: Variables que indican el estado del modelo, por ejemplo en el avión, podría ser la posición, la velocidad y los ángulos que indican el movimiento actual del avión.

Sigma: Sigma es un tiempo que queremos que pase desde un momento dado hasta el envió de un mensaje por las salidas, esta función se denomina lambda.

Los modelos Devs no varían solo cuando se produce una entrada como haría una máquina de Moore, sino que el modelo puede evolucionar en función de su estado cuando pasa un tiempo determinado, a esto se le denomina transición interna y esta función es llamada después de la función lambda.

En contraposición a las transiciones internas estarían las transiciones externas, esto es, cada vez que una entrada actualiza su valor, el simulador informaría al modelo de que esto ha ocurrido y el modelo sería quien tendría que decidir que decisión sería la adecuada a tomar.

Hay una tercera función de transición denominada función de transición de confluencia, esta función sería llamada por el simulador en caso de que aconteciesen al mismo tiempo una transición interna y otra externa, así podríamos decidir en cada caso que es lo que más nos interesa: realizar una y luego otra, realizar una e ignorar la otra…

En Devs hay dos tipos básicos de modelos, los atómicos y los acoplados.

Modelos atómicos:

Constan de un conjunto de puertos de entradas y salidas.

Modelos acoplados:

Constan de un conjunto de componentes (modelos atómicos o acoplados)

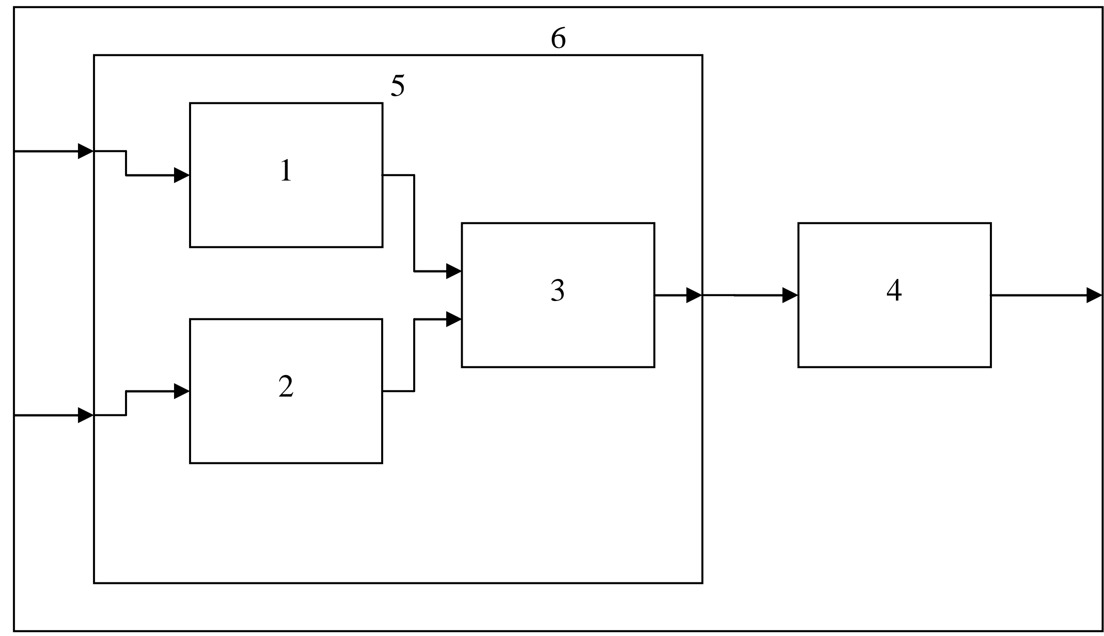
Conjunto de conexiones.

-IC: Conexiones internas (internal connections).

-EIC: Conexiones externas de entrada (external input c)

-EOC: Conexiones externas de salida (external output c)

Vamos a ver un ejemplo de lo dicho en Devs.



El componente más externo sería el bloque 6, que contendría al bloque 5 y al 4.

Conexiones EIC: que irían hasta los puertos de entrada del bloque 5.

Conexiones EOC: que irían desde el puerto de salida de 4 hasta la salida de 6.

Conexiones IC: que irían desde la salida de 6 hasta la entrada de 4.

El bloque 5 tendría igualmente lo siguiente:

Componentes: bloques 1, 2,3

Conexiones EIC: entrada hasta 1, entrada hasta 2.

Conexiones EOC: 3 hasta salida.

Conexiones IC: que irían desde 1 hasta 3 y desde 2 hasta 3.

XDevs

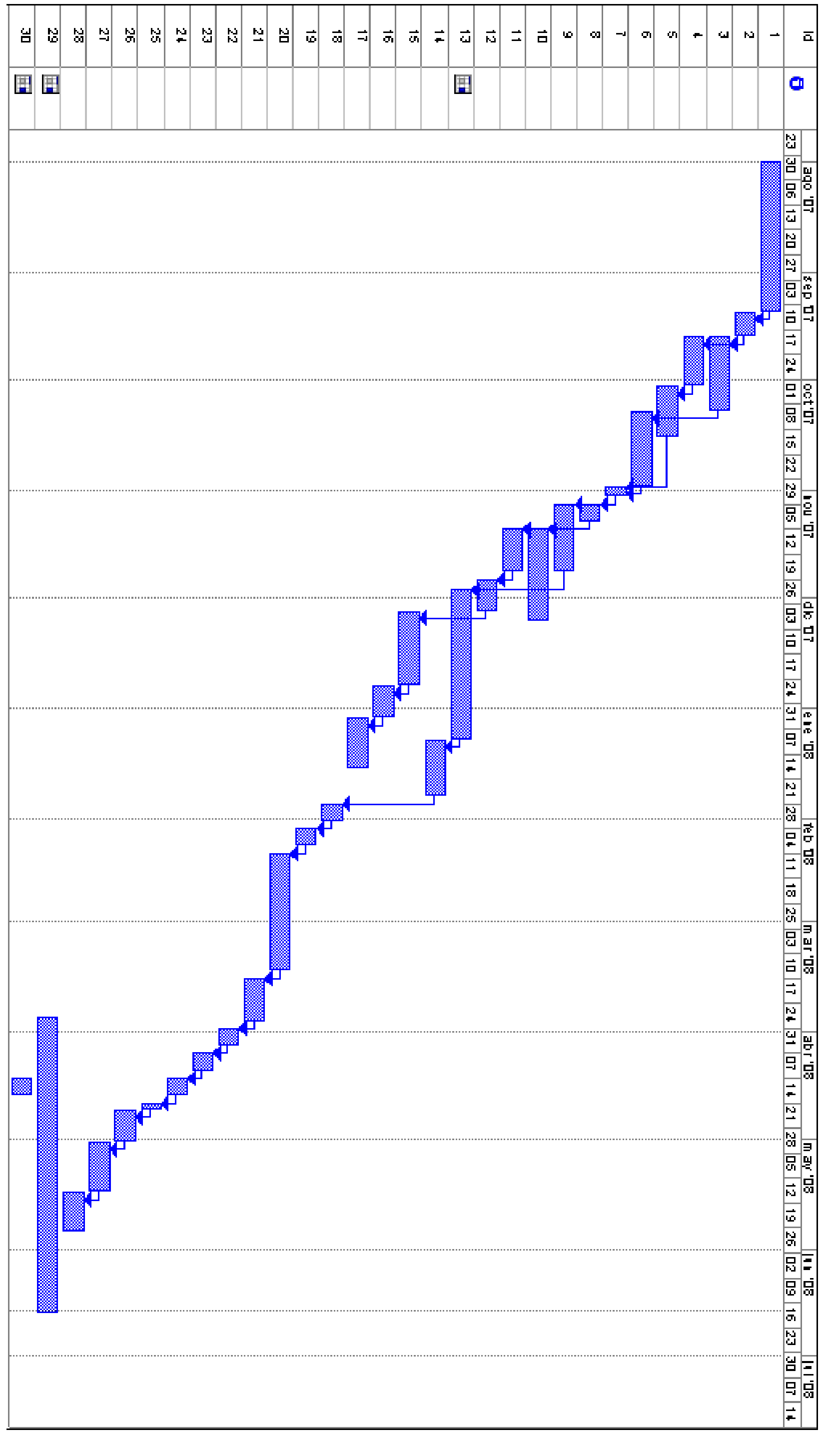
Xdevs consiste en una API devs realizada por el departamento de DACYA de la Universidad Complutense de Madrid, más concretamente por José Luís Risco Martín que consiste en una implementación del paradigma devs. Xdevs se encuentra en una librería java que se puede descargar desde: http://www.dacya.ucm.es/jlrisco/xdevs.htm

Fases del proyecto

Lista de tareas

1- Iniciación Devs
2- Primeros programas Devs
3- Modelo Avión
4- Prototipo para el manejo de 1 avión
5- Ampliación del prototipo para manejo de varios aviones
6- Controlador Avión
7- Comprobar el funcionamiento de lo anterior
8- Creación de rutas para los aviones
9- Creación de rutas para los aviones en XML
10- Creación de prototipo en java 3d
11- Transformación a atomicstate
12- Inclusión de métodos de integración en avión y controlador
13- Implementación transformación de .xml a xDev y viceversa
14- Modelo del barco
15- Transformación de lo creado en java3d a OpenGL
16- Información sobre creación de terrenos
17- Creación de terrenos
18- Modelo de referencia del barco
19- Modelo de controlador de rumbo del barco
20- Depuración de errores de los modelos del barco
21- Controlador de barco (ir a punto)
22- Modelo del mar
23- Modelo de corriente
24- Modelo de náufragos
25- Modelo de multiplexor
26- Creación de la simulación con aviones, barcos y náufragos
27- Creación del controlador de la simulación
28- Creación de un primer algoritmo
29- Cambio a vista con OpenGL
30- Investigación algoritmos de inteligencia artificial
31- Implementación algoritmo genético

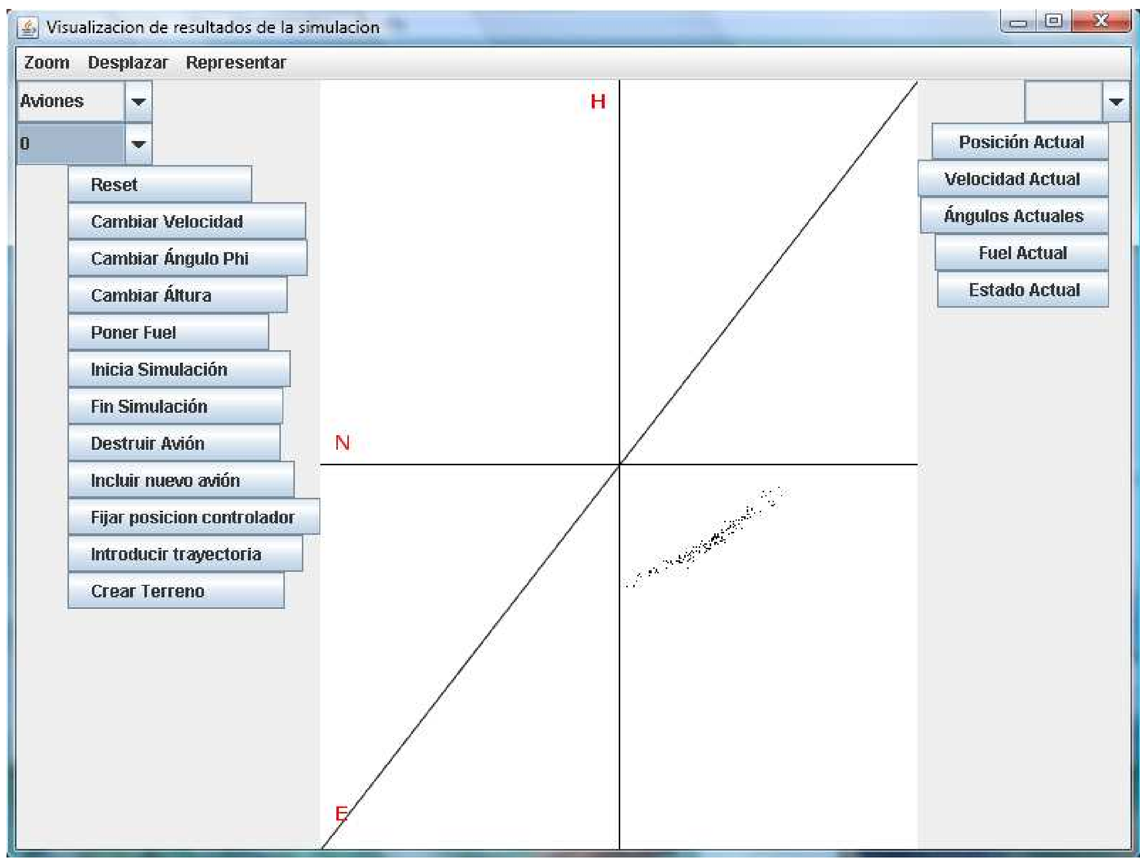
Evolución de las tareas: Diagrama de Gantt



Evolución de los prototipos creados

Prototipo 2D

Nuestro primer prototipo y con el que hemos trabajado un mayor tiempo, fue un prototipo orientado a la prueba de los modelos.



Este modelo visual tenía funcionalidades, que en la versión final no hemos incluido, debido a que estaban orientadas a la realización de pruebas y comprobación del funcionamiento, podíamos seleccionar la operación que deseásemos en los botones de la izquierda, de este modo, podíamos cambiar la velocidad de un vehículo, su ángulo phi, su altura, o destruirlo y ver si se obtenían los resultados deseados.

Si por ejemplo dábamos un punto de referencia, este era marcado en la posición que correspondía, de este modo, si el avión pasado un tiempo, llegaba a ese punto, veíamos que el comportamiento había sido el correcto.

También tenía otras funcionalidades gráficas, como por ejemplo podíamos hacer zoom o movernos libremente por el espacio.

Un punto débil que tenía esta representación, es que no permitía ver el terreno, además nos mostraba del mismo modo, (con una poli línea) los distintos vehículos, ya fueran aviones, barcos o náufragos. Lo que nos llevó a plantearnos la posibilidad de cambiar el visor por un posible visor 3d.

Visor 3D empleando Java-3D

Nuestro siguiente paso en cuanto al visor, se realizó gracias al API java-3d, éste API, simplifica enormemente la creación de entornos 3d, por lo que no nos llevó mucho tiempo la creación de un visor que mostrara un terreno generado por nosotros.

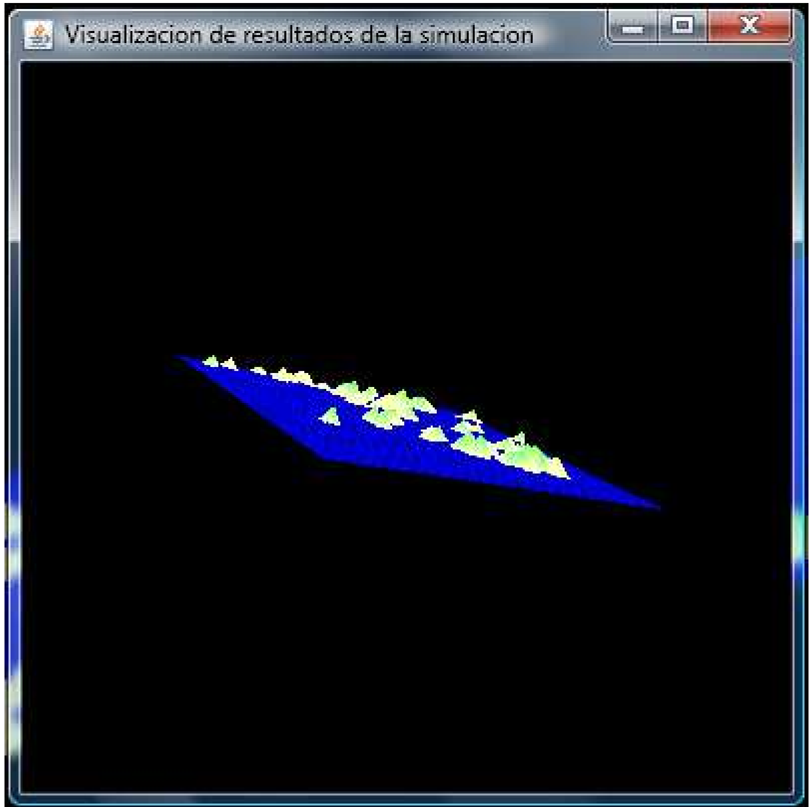
Nuestro algoritmo de generación de terrenos tenía 3 fases:

-1: Elección de puntos aleatorios: a estos puntos se les asignaba una altura aleatoria en función de las características del terreno.

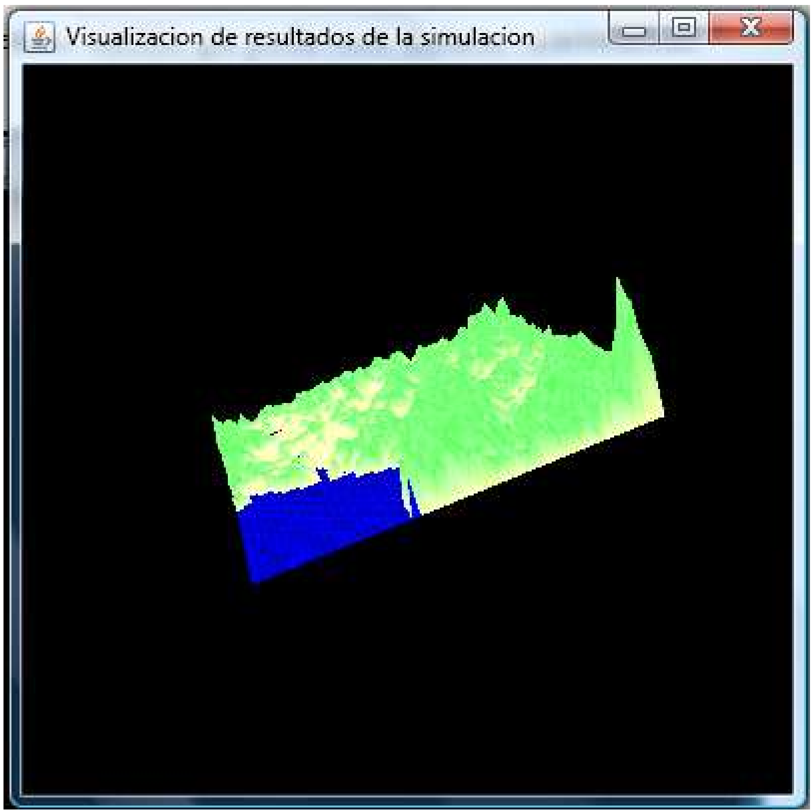
-2: Generación de resto de las casillas: en función de las características del terreno estimábamos las probabilidades de subida o bajada y subíamos un porcentaje con respecto a la posición inicial.

-3: Generación del mar: Debido a que nuestras misiones iban a ser principalmente en mar, pensamos que sería interesante la incorporación de mar siempre, a todo tipo de terreno, es por ello, que se buscaban las zonas de menor altura (unidas) y se sustituían por mar.

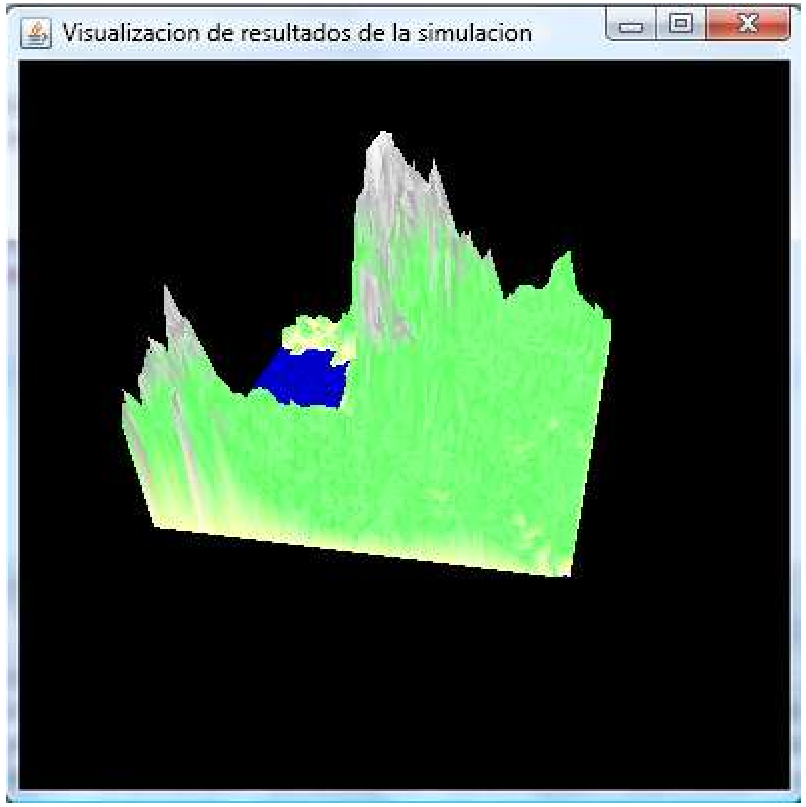
Con este sencillo algoritmo (nuestro, no como el que emplearíamos más adelante) obteníamos un terreno sencillo que podía tener distintas características.



Ejemplo de terreno de mar



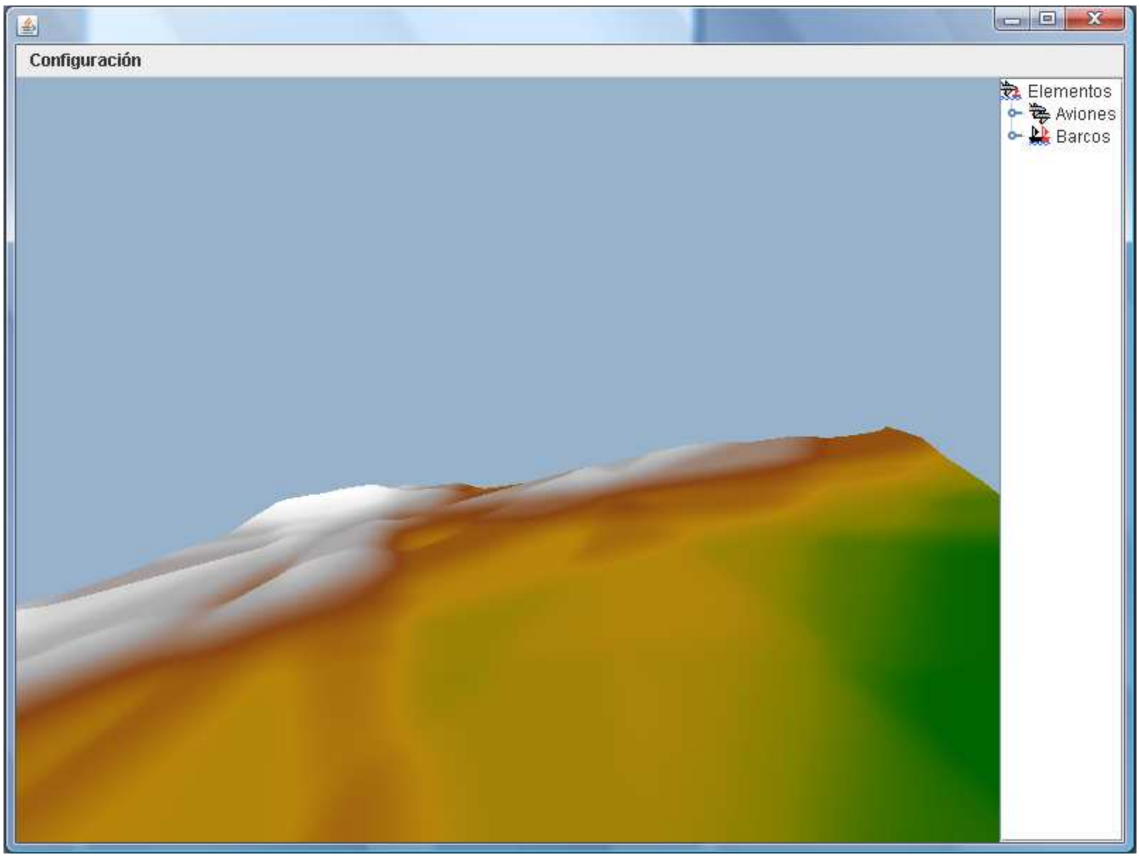
Ejemplo de terreno de altura intermedia



Ejemplo de terreno de montaña

Sin embargo, la sencillez a la hora de implementar un terreno en java-3d tiene su desventaja, debido a que el consumo de recursos es mucho mayor, es por ello, y debido a que una simulación de nuestro programa podría tener de 100 a 500 individuos, además de la creación de terreno que estimamos oportuno la creación de los terrenos, así como de los vehículos en OpenGL.

Del mismo modo sería sustituido nuestro algoritmo de creación de terrenos por un algoritmo de creación de terrenos real, lo que le da un aspecto mucho más realista al terreno.



Vista final del terreno

Tema 1: Modelo simplificado de avión

1.1. Modelo en vuelo horizontal

Vamos a describir el modelo del avión en vuelo horizontal.
La dinámica del avión moviéndose a una altura constante está dada por:

$$\begin{aligned} \dot{x} &= V\cos\psi \\ \dot{y} &= V\sin\psi \qquad (1) \\ \dot{\psi} &= w \end{aligned}$$

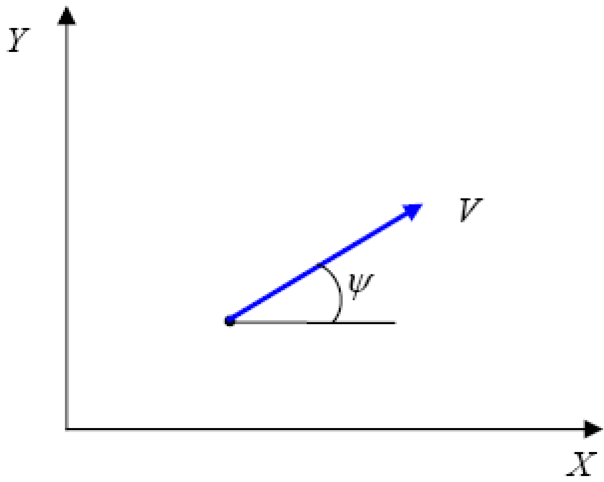


Figura 1. 1 Movimiento horizontal del avión

Donde *x* e *y* son las coordenadas cartesianas de la posición, V es el módulo de la velocidad y ϕ es el rumbo, y *w* es la señal de control o velocidad angular de entrada.
Vamos a determinar de forma aproximada la relación que liga el rumbo con el ángulo de alabeo del avión. La figura 2 muestra las fuerzas que aparecen en un giro.

La fuerza de empuje total *L* es perpendicular a las alas del avión. Su componente vertical L_V contrarresta el peso del avión *W* ya que suponemos que durante el giro no se produce movimiento en el plano vertical. Aparece una fuerza tangencial L_r que hace que el avión gire. Se verifica en estas condiciones

$$L_v = L\cos\phi = W \qquad (2)$$

$$L_r = \sqrt{L^2 - W^2} \qquad (3)$$

Se define el factor de carga, *n*, como la relación entre la fuerza de ascensión L y el peso:

$$n \equiv \frac{L}{W} \qquad (4)$$

El factor de carga se expresa en g's. Por ejemplo, un avión con una fuerza ascensorial total tres veces mayor que su peso experimenta un factor de carga de 3g. Al factor de carga también se le denomina "peso aparente".

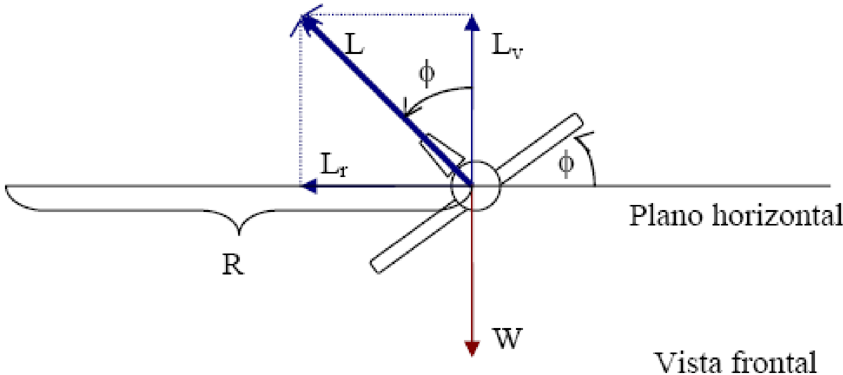


Figura 1. 2 Fuerzas en el movimiento lateral de un avión

Sustituimos el factor de carga en las ecuaciones de las fuerzas para obtener las relaciones:

$$n = \frac{1}{\cos\phi} \qquad (5)$$

$$L_r = W\sqrt{n^2 - 1} = W\tan\phi \qquad (6)$$

La ecuación (5) relaciona directamente el factor de carga con el ángulo de alabeo (roll o bank angle). Vemos que en la relación entre ambos no interviene la velocidad, luego un determinado ángulo de alabeo genera un valor fijo de factor de carga independientemente de la velocidad y el peso del avión.

Dada que *W = Mg,* siendo *M* la masa del avión, tenemos

$$L_r = Mg\tan\phi = Ma_r \qquad (7)$$

Donde a_r es la aceleración radial

$$a_r = g \tan\phi = g \text{ sqr}(n^2\text{-}1) \qquad (8)$$

La velocidad interviene en el radio de giro y en la velocidad angular del giro. La relación se encuentra mediante la fórmula que liga la aceleración radial con una velocidad lineal constante:

$$a_r = \frac{V^2}{R} \qquad (9)$$

La aceleración radial se debe a la fuerza L_r, luego relacionando ambas se obtiene la expresión

$$L_r = M\frac{V^2}{R} = \frac{W}{g}\frac{V^2}{R} \qquad (10)$$

Relacionando (9) y (6) – o (9) y (7)- obtenemos el radio de giro

$$R = \frac{V^2}{g\sqrt{n^2-1}} = \frac{V^2}{g\tan\phi} \qquad (11a)$$

$$R = \frac{V^2}{a_r} \qquad (11b)$$

La velocidad angular de giro, ω, está dada por la velocidad lineal dividido por el radio de giro, luego

$$\omega = \dot{\psi} = \frac{V}{R} = \frac{a_r}{V} = \frac{g\tan\phi}{V} = \frac{g\sqrt{n^2-1}}{V} \qquad (12)$$

El ángulo de alabeo está dado por:

$$\phi = \arctan\left(\frac{V\omega}{g}\right) \qquad (13)$$

Para las maniobras de evasión interesa que el avión gire lo más deprisa posible, recorriendo el mínimo espacio posible, lo cual nos dice –por las ecuaciones (11) y (12)- que el radio se debe minimizar, lo que se consigue con el máximo valor de carga, y la velocidad angular debe ser lo mayor posible, lo que se consigue con la menor velocidad lineal.

En resumen las ecuaciones que describen la evolución del modelo simplificado de avión son:

$$V_x = \dot{x} = V\cos\psi$$
$$V_y = \dot{y} = V\sin\psi$$
$$\dot{\psi} = w$$
$$\phi = \arctan\left(\frac{V\omega}{g}\right)$$

Para implementarlas en el computador usamos la aproximación discreta:

$$\begin{aligned} x_{k+1} &= x_k + \Delta t\, V\cos\psi_k \\ y_{k+1} &= y_k + \Delta t\, V\sin\psi_k \\ \psi_{k+1} &= \psi_k + \Delta t\, w_k \\ \phi_k &= \arctan\left(\frac{V\omega_k}{g}\right) \\ V_{xk} &= V\cos\psi_k \\ V_{yk} &= V\sin\psi_k \end{aligned} \qquad (14)$$

Δt es el intervalo de tiempo. La velocidad angular ω_k es una señal de entrada o de control. Las señales de salida del modelo son:

Posición: x_k, y_k
Velocidad: V_{xk}, V_{yk}
Orientación: ψ_k, ϕ_k

El eje X del modelo representa la dirección Norte, el eje Y representa la dirección Este.

El factor de carga, y por lo tanto la aceleración angular y el ángulo de alabeo, están limitados a unos valores máximos en función de la altura mediante la relación:

$$n_{max} = 5.3820e\text{-}9 * h^2 - 4.4291e\text{-}4 * h + 6.1000e\text{+}0 \qquad (15)$$

La altura máxima es de 14.000 m.

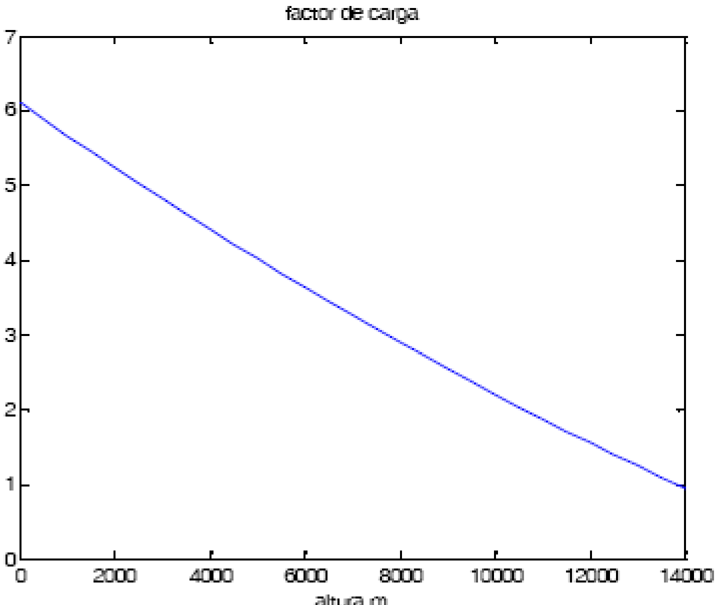


Figura 1.2 Factor de carga máximo

1.2. Modelo de cambio de velocidad

El modelo de cambio de velocidad del avión es:

$$\frac{dV}{dt} = a(h) \qquad (16)$$

Donde la aceleración *a (h)* está dada por la relación:

$$a(h) = \begin{cases} K_p(V_r - V) & \text{si } |K_p(V_r - V)| < a_{\max} \\ sign(K_p(V_r - V))\, a_{\max} & \text{en otro caso} \end{cases} \qquad (17)$$

Con

$$a_{max} = -4.0365e\text{-}9 * h^2 - 2.0505e\text{-}4 * h + 3.8000e{+}0 \qquad (18)$$

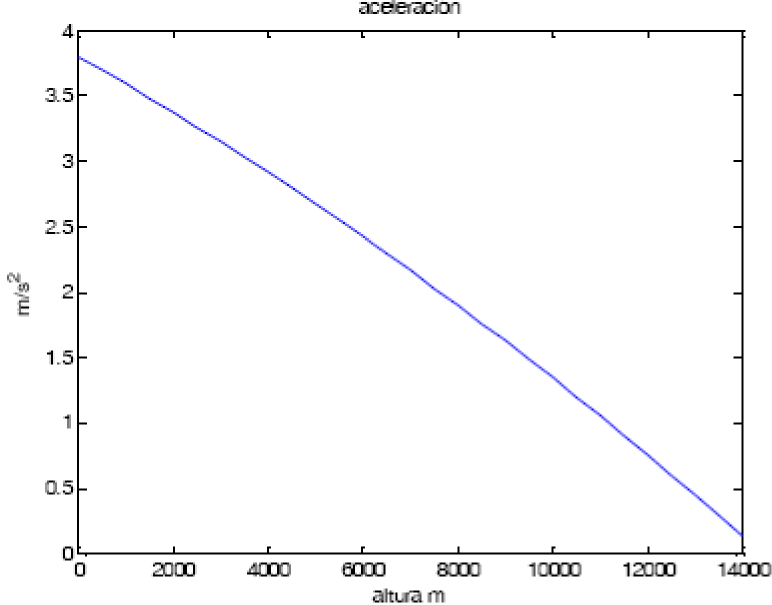


Figura 1.3 Aceleración en función de la altura

Las velocidades máxima y mínima posibles de un avión están limitadas por la altura de vuelo mediante las relaciones:

Donde la altura máxima es de 14000 m.

$$V_{min} = 7.8038e\text{-}7 * h^2 - 8.2021e\text{-}4 * h + 5.9000e{+}1 \qquad (19a)$$

$$V_{max} = -1.0764e\text{-}7 * h^2 - 8.2021e\text{-}4 * h + 3.3500e{+}2 \qquad (19b)$$

La figura 1.4 muestra los valores de las velocidades máximas y mínimas en función de la altura.

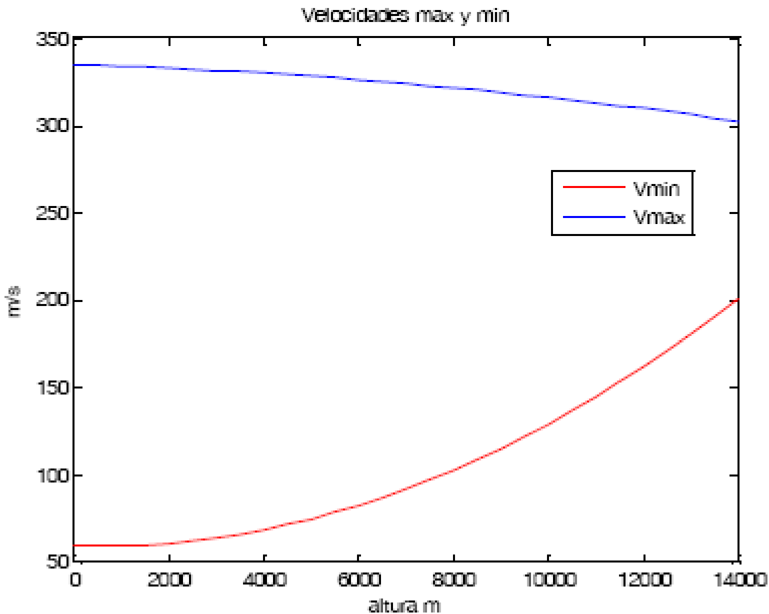


Figura 1.4 Velocidades máximas y mínimas

El modelo discreto para el cambio de velocidad es

$$V_{k+1} = V_k + \Delta t\, a(h)$$
$$a(h) = K_p (V_k - V_{rh}) \qquad (20)$$
$$\text{if } |a(h)| \geq a_{\max} \quad \text{then} \quad a(h) = sign(a(h)) * a_{\max} \quad \text{end}$$

Con valor de la ganancia $K_p = 1.25$.

1.3. Movimiento vertical

De forma aproximada la dinámica vertical o de cambio de altura del avión está dada por:

$$\ddot{h} = -a_1\, \dot{h} - a_2(h - h_c) \qquad (21)$$

Donde h representa la altura del avión, h_c la altura deseada, $V_h = \mathrm{d}h$ la velocidad de subida y a_1, a_2 son parámetros que dependen de las características del avión.

Utilizaremos los valores:

$$a_1 = 0.76666$$
$$a_2 = 0.58768 \qquad (22)$$

De forma discreta la ecuación (21) se puede aproximar en la forma:

$$h_{k+1} = h_k + \Delta t\, V_{hk}$$
$$V_{hk+1} = V_{hk} + \Delta t\, \left(-a_1\, V_{hk} - a_2\, (h_k - h_{ck})\right) \qquad (23)$$

La velocidad de ascensión está limitada en función de la altura. En subida la velocidad máxima está dada por la ecuación:

$$V_{h\max\uparrow} = 1.7403e-11 * h^3 - 4.8328e-7 * h^2 - 3.1700e-3 * h + 9.3257e+1 \quad (24)$$

Y en bajada

$$V_{h\max\downarrow} = 2.8064e-12 * h^3 - 2.5433e-7 * h^2 + 4.7390e-3 * h - 4.6005e+001 \quad (25)$$

La figura 1.5 muestra los valores de las educaciones (24) y (25) en función de la altura.

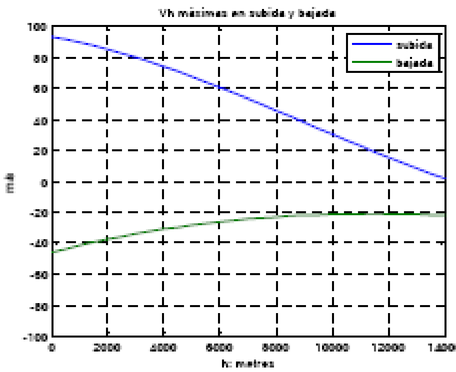


Figura 1.5 Valores máximos de V_h en subida y bajada

Las aceleraciones máximas de subida y bajada también están limitadas en función de la altura de vuelo. Para la subida la aceleración máxima es de 29.37 m/s2 entre 0 y 8000 m, y entre 8000 y 14000 m se tiene la relación:

$$a_{h\max\uparrow} = -4.7077e-3 * h + 6.6078e+1 \quad (26)$$

Para bajadas se tiene:

$$a_{h\max\downarrow} = -7.1275e-8 * h^2 + 1.7604e-3 * h - 2.0142e+1 \quad (27)$$

La figura (1.6) muestra las aceleraciones máximas en función de la altura.

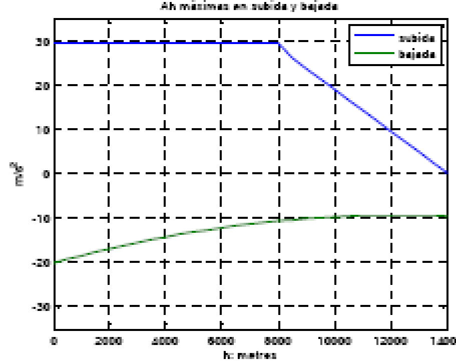
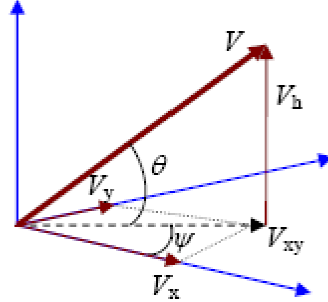


Figura 1.6 Valores máximos de la aceleración en subida y en bajada

Las limitaciones en altura y en velocidades y aceleraciones máximas en ascensión y en descenso hacen que se tengan que limitar los valores de la ecuación (23). La altura no debe superar los 14000 metros. La velocidad no puede superar o bajar de los valores dados por las ecuaciones (24) y (25), y el término (−a1Vhk−a2(hck−hk)) no puede superar o bajar de los valores dados por las ecuaciones (26) y (27).

1.4. Composición de los movimientos horizontal y vertical

La figura 1.7 muestra los componentes del vector velocidad en el espacio, y los ángulos de cabeceo y rumbo.



Se verifican las relaciones

$$\sin(\theta) = \frac{V_h}{|V|}$$
$$|V| = \sqrt{V_x^2 + V_y^2 + V_h^2} \qquad (28)$$
$$\tan(\psi) = \frac{V_y}{V_x}$$

1.5. Valores iniciales

El avión partirá de unas condiciones iniciales dadas:

· Componentes de la posición (Norte, Este y Altura) expresadas en metros.

· Componentes de la velocidad (Norte, Este y Altura) expresadas en metros por segundo.

Estos valores iniciales determinan el rumbo y el ángulo de cabeceo iniciales del avión.

Lo normal será que la velocidad de subida V_h sea cero. Suponemos que el ángulo de alabeo inicial es siempre cero (el avión no está girando).

Los cálculos se harán en metros, metros por segundo y radianes.

1.6. Entradas y salidas del modelo

1.6.1. Entradas

Reset(Posición, Velocidad)

Pone la posición y velocidad inicial del avión a los valores dados.

Posición: vector fila de tres componentes [P_N, P_E, P_h]

Velocidad: vector fila de tres componentes [V_N, V_E, V_h]

CambiarVelocidad(Vr)

Da la orden de que el avión cambie la velocidad al valor indicado por Vr.

Vr: valor de nueva velocidad en m/s

CambiarÁnguloPhi(ϕ_r*)*

Da la orden de que el avión cambie el ángulo de alabeo al valor ϕ_r.

Φ_r: valor de ángulo de alabeo en radianes (-pi, pi].

CambiarÁnguloTheta(θ_r*)*

Da la orden de que el avión cambie el ángulo de cabeceo al valor θ_r.

Θ_r: valor de ángulo de cabeceo en radianes (-pi, pi].

PonerFuel(F)

Pone el valor de fuel del avión.

IniciaSimulacion(dt)

Da la orden de que el avión empiece a evolucionar según su modelo discreto con intervalo de tiempo discreto dt.

FinSimulacion()

Da la orden de que el avión deje de evolucionar y permanezca en el estado actual.

DestruirAvion()

Da la orden de que el avión pase al estado destruido. No puede proporcionar ninguna información.

1.6.2. Salidas

GetPosicion()

Devuelve un vector fila con las tres componentes del vector de posición [P_N, P_E, P_h].

GetVelocidad()

Devuelve un vector fila con las tres componentes del vector de velocidad [V_N, V_E, V_h].

GetÁngulos()

Devuelve un vector fila con las tres componentes de los ángulos de vector de posición [ϕ , θ , ψ].

GetFuel()

Devuelve el valor del nivel de combustible.

GetEstado()

Devuelve el valor del estado: Destruido, Running, Parada.

1.7. Control de rumbo

Para hacer que el avión siga un rumbo determinado utilizamos una ley de control de la forma:

$$w = \frac{2V}{L}\sin\eta \qquad (1)$$

Donde V es la celeridad del avión, L es la distancia del avión a un punto de referencia situado en la trayectoria adelante del vehículo, y η es el ángulo que hay que girar el vector velocidad para hacerlo coincidir con la línea que une el avión con el punto de referencia, ver la figura 2.1.

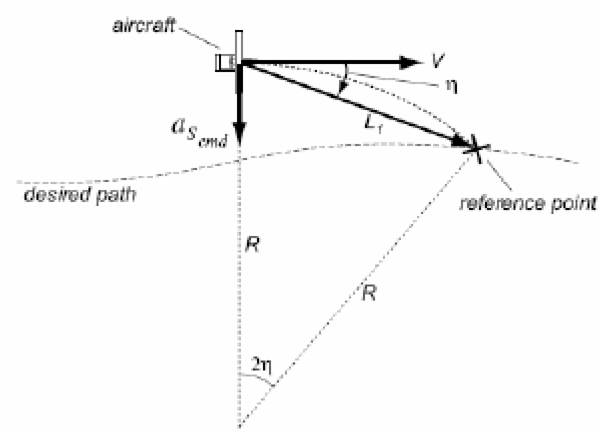


Figura 2.1 Diagrama para el control no lineal

La ley de control (1) equivale a introducir una aceleración centrípeta que haga tender al avión hacia el punto de referencia, con un radio:

$$R = \frac{V}{w} = \frac{L}{2\sin\eta} \qquad (2)$$

Y aceleración:

$$a_n = Vw = \frac{2V^2}{L}\sin\eta \qquad (3)$$

Claramente la velocidad angular está limitada por el máximo valor de la velocidad angular para la altura de vuelo en ese instante.

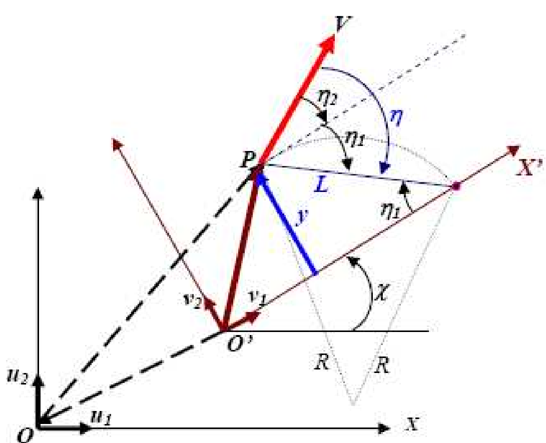


Figura 5.1 Modelo para control

Se supone que el avión situado en *P* y con vector velocidad *V* en el plano, debe seguir la trayectoria de referencia que coincide con el eje *X'* del sistema de referencia girado un ángulo χ con respecto al eje *X* del sistema inercial y desplazado al punto O'. El ángulo χ es positivo en los giros contrarios al sentido del reloj. Sobre el eje *X'* se sitúa el punto de referencia que dista *L* de la posición actual del avión. La distancia del avión al eje *X'* es su coordenada *y*. El punto *P* forma un ángulo η_1 con el punto de referencia sobre la trayectoria. El vector velocidad forma un ángulo η_2 con la trayectoria deseada.

El ángulo $\eta = \eta_1 + \eta_2$ indica el giro necesario para que el vector velocidad lleve la dirección necesaria para alcanzar el punto de referencia. Fijarse que el sentido dibujado al ángulo η (y en η_1 y η_2) es el necesario para alcanzar el punto de eferencia, y este sentido de giro es negativo, ya que lleva la dirección contraria a un giro positivo en el sistema de referencia (el sentido es positivo en el sentido contrario de las agujas del reloj, y negativo en sentido contrario).

La ley de control que se aplica corresponde a una aceleración velocidad centrípeta o velocidad angular de giro que llevara al avión al punto de referencia siguiendo un arco que una los puntos P y de referencia y con segmento circular L. El radio de giro está dado por:

$$R = \frac{L}{2\sin\eta}$$

Y la velocidad angular es:

$$w = \frac{|V|}{R} = \frac{2|V|}{L}\sin\eta \qquad (4)$$

La relación entre la velocidad angular y el ángulo de alabeo ϕ está dada por la ecuación (13) de la sección 1:

$$\phi = \arctan\left(\frac{V\omega}{g}\right)$$

De forma discreta tenemos para intervalo de tiempo:

$$\phi_k = \arctan\left(\frac{V_k\omega_k}{g}\right)$$

Hay que tener en cuenta que el ángulo de alabeo está limitado en función de la altura de vuelo debido a la limitación que existe en el factor de carga. La relación entre éste la velocidad angular y el ángulo de alabeo está dada por la ecuación (12) de la sección 1:

$$\omega = \dot{\psi} = \frac{V}{R} = \frac{a_r}{V} = \frac{g\tan\phi}{V} = \frac{g\sqrt{n^2-1}}{V}$$

El valor de *L* lo determinamos de modo que la frecuencia natural, del sistema lineal de segundo orden equivalente a la ley de control, sea de *0.5* rad/seg, mediante la relación

$$L = \frac{\sqrt{2}|V|}{w_n}$$

Valores de w_n más elevados producen maniobras más agresivas, y estas se suavizan con valores menores.

El cálculo de η_2 se realiza hallando el rumbo del vector velocidad respecto al sistema de ejes situados en la trayectoria, O'X'. Para ello se calculan las componentes del vector velocidad respecto del sistema de ejes rotados y se calcula el rumbo del avión respecto de este sistema de referencia. Para calcular el ángulo η_1 determinamos la distancia del avión con respecto al sistema de referencia O'X'. Esta distancia es la componente Y del vector posición expresado en el sistema de referencia O'X'. Valores positivos de estos ángulos indican sentidos de giro negativos (contrarios al sentido de giro de las agujas del reloj), y viceversa.

La velocidad angular de giro se debe limitar en valor absoluto a la máxima velocidad angular de giro que posee el avión a la altura de vuelo actual, lo que limita el factor de carga de la forma correspondiente.

En ausencia de error en el rumbo y en la posición del avión, éste debe estar situado sobre la trayectoria de referencia, luego la señal de rumbo que se debe dar al avión es: *rumbo* = $\chi + \eta$. En el caso de que el rumbo actual del avión respecto de la trayectoria sea superior en valor absoluto a $\pi/2$, damos como rumbo el valor χ de la dirección de la trayectoria y el factor de carga máximo que corresponde a la máxima velocidad angular posible.

1.8. Implementación del Avión

En nuestra implementación hay una clase UAV (modelo acoplado) que contiene los siguientes modelos atómicos.

Modelado propiamente dicho del avión

```
private AvionState _avión;
```

Encargado de realizar al avión las peticiones encargadas por el usuario (en nuestra versión final, debido a la falta de interacción con el usuario no tiene gran utilidad)

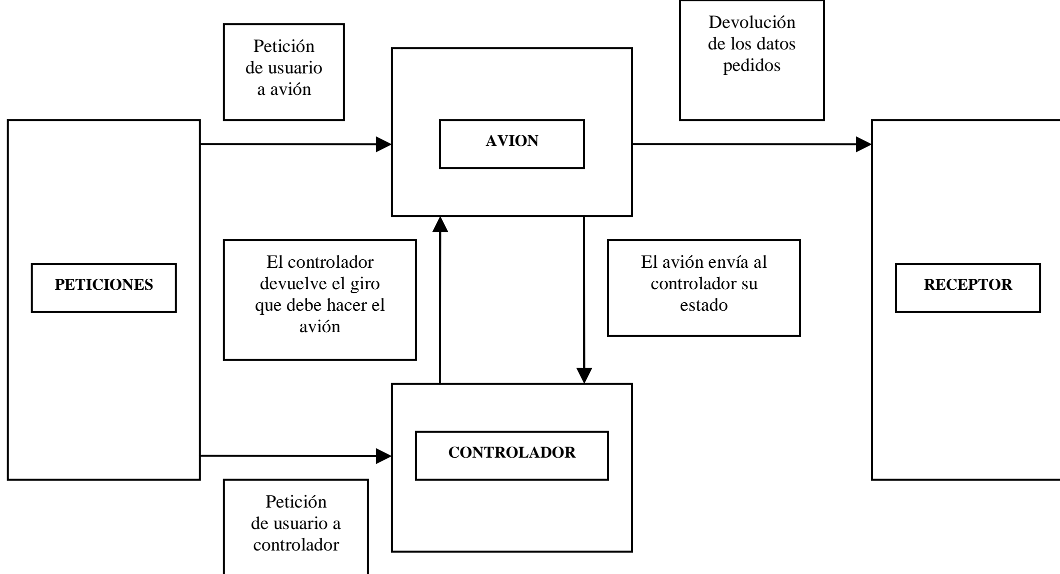
```
private PeticionesState _peticion;
```

Encargado de recibir aquello que devuelve el avión.

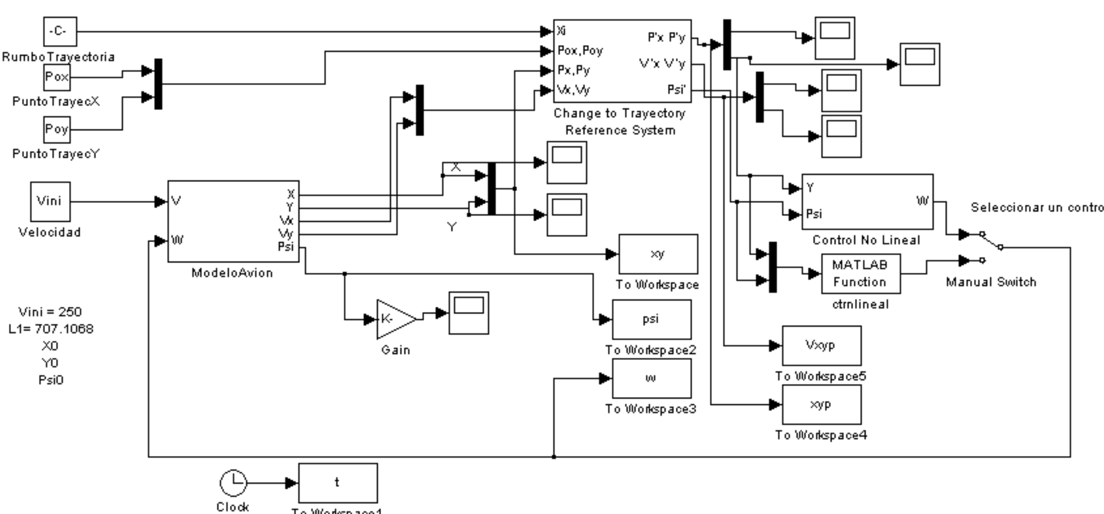
```
private ReceptorState _receptor;
```

Controlador encargado de guiar a un avión al punto deseado.

```
private ControladorRumboState _controlador;
```



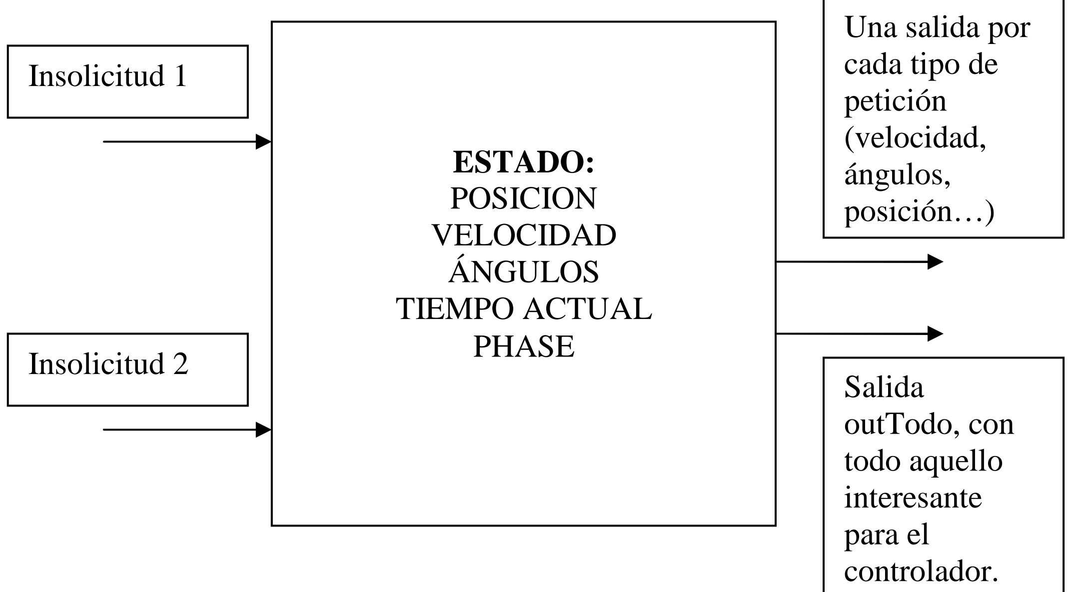
Hemos utilizado el esquema Simulink siguiente para comprobar el funcionamiento del controlador del avión:



Pasemos a desglosar las dos clases principales del UAV:

1.8.1. AvionState

Puertos:



Posibles valores de Phase:

Los valores que puede tomar phase son: parada (esto es, no ha sido aun generado), running (avión en movimiento) y destruido (pensamos que podría ser útil el tratamiento de colisiones con el terreno u otros vehículos, aunque finalmente no ha sido incluido)

Estado inicial:

La phase del avión en el instante inicial (construcción), es la de avión en estado de parada, esto es, en espera a que se produzca una inicialización del mismo. Le ponemos por tanto sigma infinito, o lo que es lo mismo, lo pasamos a estado passivate.

Función de transición interna:

La función de transición interna se encarga de comprobar si ha habido alguna petición en el instante anterior, de este modo puede desactivar el flag que indica un tipo de petición.

Función de transición externa:

En caso de recibir alguna petición externa, el avión lo que hace es calcular el estado (en función de cuanto tiempo haya pasado desde la última transición interna) y devolverlo al exterior por el puerto correspondiente.

Función de confluencia:

En caso de que ocurran simultáneamente la función de transición interna y la externa preferimos ejecutar antes la función de transición interna para que el avión continúe evolucionando, ejecutando justo después la función de transición externa.

Función lambda:

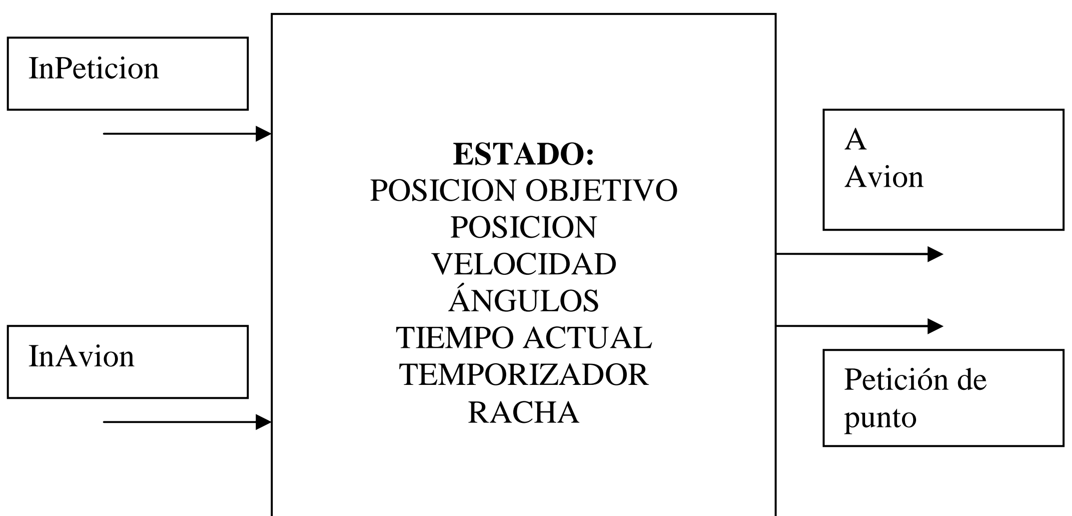
La función lambda es la encargada de realizar la integración del estado del avión, esto lo realiza gracias a que implementa la interfaz Iintegración, la integración del avión se produce en avanzaTiempo().

```
	public void avanzaTiempo() {
		double tiempo = this.getStateValue("dt").doubleValue();
	this.actualizaEstados(integrador.integra(this,
tiempo,this.getStateValue("tactual").doubleValue()));
	this.setStateValue("tactual",
this.getStateValue("tactual").doubleValue()+tiempo);
	}
```

Que se calcula gracias a las derivadas que devuelve la función dameDerivadas()

```
derivadas[0]=v*Math.cos(this.getStateValue("rumbo").doubleValue());
derivadas[1]=v*Math.sin(this.getStateValue("rumbo").doubleValue());
derivadas[2]= vh;
derivadas[3]= -a1*vh -a2*(h-hck);
```

1.8.2. ControladorRumboState



Puertos:

El control de rumbo está comunicado con el avión de modo que cada vez que el avión avanza, le envía por el puerto InAvion al controlador un mensaje que contiene todos los datos necesarios para que el controlador calcule el próximo paso.

Por el puerto InPeticion pueden llegar peticiones externas, esto es, por ejemplo peticiones que desee hacer un usuario desde la aplicación o peticiones que realice el algoritmo.

Por el puerto de salida al avión se le indica la acción que debe tomar el avión para llegar al estado deseado, como pueda ser girar, acelerar, etc.

Por el puerto de salida PeticionDePunto el controlador pide en caso de ser necesario un nuevo punto de la ruta al controlador.

Posibles valores de Phase:

Los posibles valores de la phase del controlador son los mismos que los del avión: no_iniciado (equivalente a parada), iniciado (equivalente a running) o destruido.

Estado inicial:

El estado inicial, es también similar al empleado para el avión, es decir, en este caso tendríamos phase no_iniciado y sigma que sería nuevamente infinito.

Función de transición interna:

Con los datos actualizados de la última transición externa, el controlador calcularía el próximo evento que necesitaría hacer el avión y añadiríamos al mensaje aquella información que debiésemos enviarle al avión.

Además tenemos que tener en cuenta, que puede ser necesario que pidamos un nuevo punto de ruta, para tener control sobre los puntos de ruta tenemos que tener en cuenta dos cosas:

1-Tenemos que fijar un tiempo mínimo.
2-Tenemos que fijar un tiempo máximo.

Cómo acotar estos tiempos puede ser tema de estudio, nosotros lo hemos hecho del siguiente modo:

1- El tiempo mínimo que hemos fijado es aquel que tardaría el avión desde la posición inicial (cuando recibe por primera vez la orden de ir a otro punto) hasta la posición objetivo yendo en línea recta, lo lógico es que al principio tenga que girar y tarde más que éste tiempo y como hemos podido probar realizando múltiples pruebas, éste tiempo es suficiente como para que el avión pueda realizar un giro cualquiera y comience a acercarse al punto.

2- Nosotros hemos estimado, que si el control es suficientemente bueno, una vez se pase la primera etapa (que puede producir un alejamiento del punto origen) lo normal es que el avión se vaya acercando cada ciclo más al punto objetivo, por tanto hemos fijado un limite que se puede variar fácilmente (en principio es de 3) que indica que en el momento que se aleje 3 turnos seguidos del punto de referencia lo tomaremos como que ya ha llegado al punto de referencia, debiéndose por tanto pedir un nuevo punto al controlador que gestione las rutas.

Función lambda:

La función lambda simplemente se encarga de enviar el mensaje con el contenido que le haya incluido la función de transición interna.

Función de confluencia:

De nuevo ejecutamos primero la función de transición interna y luego la externa.

La función de control del avión queda del siguiente modo:

```
public void controlUAV() {
if (getStateValue(activo).intValue()==1) {
            Vector<Double> aux = new Vector<Double>(2,0);
      aux.add(new      Double      (getStateValue("pe").doubleValue()-
getStateValue("poe").doubleValue()));
      aux.add(new      Double      (getStateValue("pn").doubleValue()-
getStateValue("pon").doubleValue()));
      Vector<Double>                        pn                    =
transformarEjes(this.getStateValue("Xi").doubleValue(), aux);{
            aux = new Vector<Double>(2,0);
            aux.add(getStateValue("ve").doubleValue());
            aux.add(getStateValue("vn").doubleValue());
      Vector<Double>                        vn                    =
transformarEjes(this.getStateValue("Xi").doubleValue(), aux);
                  double eta2 = Math.atan2(vn.get(1),vn.get(0));
                  double va = Math.sqrt(
      getStateValue("ve").doubleValue()*
getStateValue("ve").doubleValue() +
      getStateValue("vn").doubleValue()*
getStateValue("vn").doubleValue()
                                    );

      double l = Math.sqrt(2)*va/(wn);
      double y = pn.get(1).doubleValue();
      double eta1 = Math.atan(y/l);
      double eta = -(eta1+eta2);
      double w = 2*va/l*Math.sin(eta);
      double nmax = nmaxcarga(getStateValue("ph").doubleValue());
      double wmax = 9.8*Math.sqrt((nmax*nmax)-1)/va;
      double wmin = 0.0;
      if (Math.abs(eta2)>=90*Math.PI/180) {
            w = Math.signum(eta)*wmax;
      }
      if (Math.abs(w)>wmax) {
            w = Math.signum(eta)*wmax;
      }
      if (Math.abs(w)<wmin) {
            w = 0;
      }
      double phiValue=0;
      phiValue = Math.atan(va*w/9.8);
      setStateValue("phi",phiValue);
      Vector solicitud = new Vector(2,0);
      solicitud.add(new Integer(AvionState.CambiarÁnguloPhi));
            solicitud.add(new Double (phiValue));
            _msg.add(OutPeticion,solicitud);
            }
      }
}
```

Tema 2. Modelo simplificado de barco

El modelo está extraído de [1] y en la figura 1 se muestran las variables utilizadas para describir el movimiento en el plano horizontal. Representamos por:

U: Vector velocidad del barco.

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

v: Velocidad de través (eje transversal) (sway)

ψ: 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.

La dinámica del buque esta definida por las ecuaciones:

$$\begin{aligned} &T_1T_2\ddot{r}(t) + (T_1 + T_2)\dot{r}(t) + r(t) = K(T_3\dot{\delta}(t) + \delta(t)) \\ &\dot{\psi}(t) = r(t) \end{aligned} \qquad (1)$$

$$\begin{aligned} &\dot{x}(t) = u(t)\cos(\psi(t)) - v(t)\sin(\psi(t)) \\ &\dot{y}(t) = u(t)\sin(\psi(t)) + v(t)\cos(\psi(t)) \\ &U(t) = \left(u^2(t) + v^2(t)\right)^{1/2} \end{aligned} \qquad (2)$$

A veces se utiliza un modelo de primer orden simplificado de (1):

$$\begin{aligned} &T\dot{r}(t) + r(t) = K\delta(t), \quad T = T_1 + T_2 - T_3 \\ &\dot{\psi}(t) = r(t) \end{aligned}$$

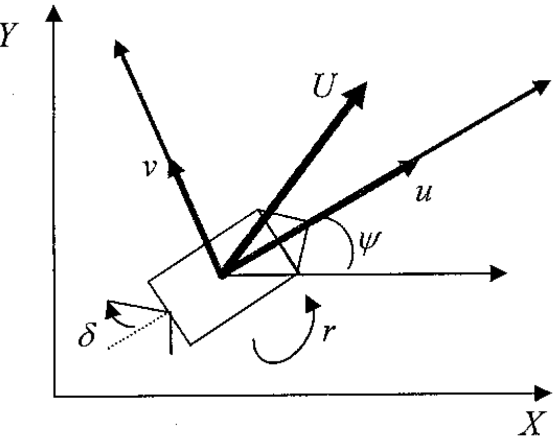


Figura 1. Variables que describen el movimiento horizontal del barco

La tabla siguiente da los valores de los parámetros del modelo (1) para dos tipos de barco.

	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

Modelo del timón:

El modelo del funcionamiento del timón se describe en la figura 2.

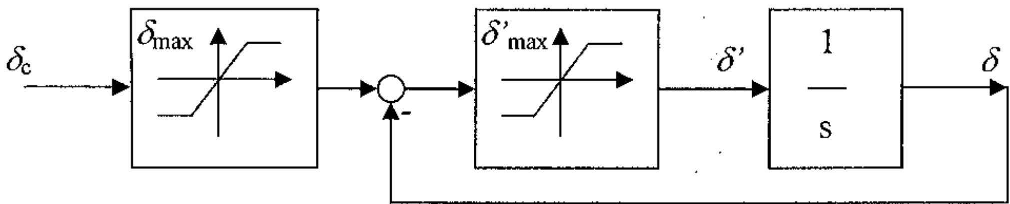


Figura 2. Modelo del control del timón

Al bloque le llega la señal de control δc y se limita a los valores máximos de +20º y -20º. En algunos barcos el ángulo puede llegar hasta +35º y -35º. Existe también una limitación en la velocidad de movimiento de timón de +5º/s y -5º/s. El modelo lo podemos describir de la forma:

$$\dot{\delta}(t) = \left[\left[\delta_c(t)\right] - \delta(t)\right]$$

Donde con los corchetes se indica que la señal se satura a sus valores máximo y mínimo

En el diagrama del modelo de timón y del barco es el de la figura 3.

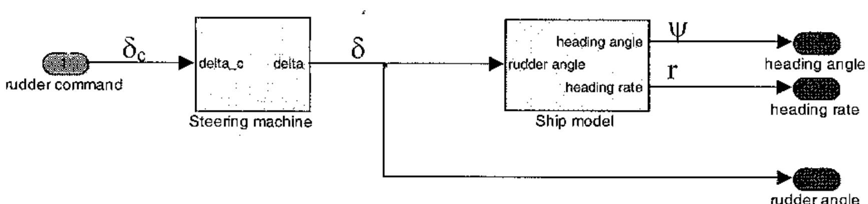


Figura 3. Diagrama de entrada-salida del conjunto timón y barco

2.1. Control del barco

El diagrama de la figura 4 muestra un esquema de control del barco. El autopiloto recibe el rumbo deseado y el rumbo y la velocidad angular del barco y genera la señal de control.

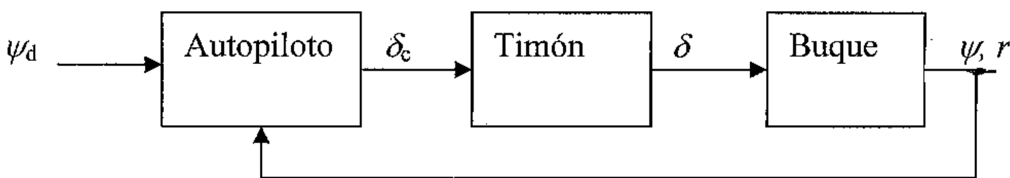


Figura 4. Diagrama de control del buque

El autopiloto tiene la estructura que se muestra en la figura 5. El rumbo deseado llega al modelo de referencia que genera valores de referencia para el rumbo, su velocidad angular y aceleración angular. Estos valores se llevan al controlador que genera la señal de control.

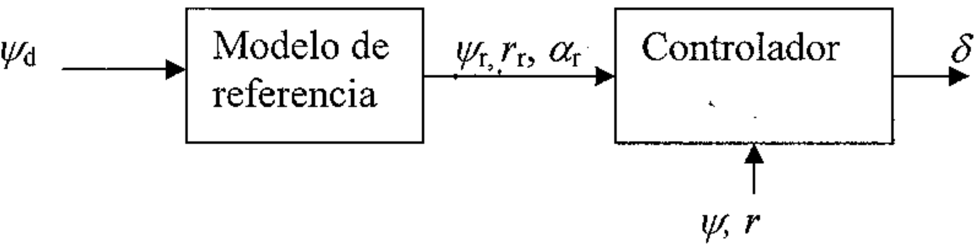


Figura 5. Estructura del autopiloto

El modelo de referencia está descrito por las ecuaciones siguientes:

$$\begin{aligned}
\dot{x}_1(t) &= x_2(t) \\
\dot{x}_2(t) &= -w_r^2 x_1(t) - w_r \zeta_r x_2(t) + x_3(t) \\
\dot{x}_3(t) &= -w_r x_3(t) + \psi_d(t) \\
\psi_r = x_1, \quad r_r = x_2, &\quad \alpha_r = \dot{x}_2 = -w_r^2 x_1 - w_r \zeta_r x_2 + x_3
\end{aligned} \qquad (5)$$

Tomaremos wr= 0.1,ζ = 1. Utilizaremos un control discreto equivalente con un periodo de muestreo de 5 segundos. Además limitaremos la velocidad de referencia en todos los cálculos a valores extremos de +- 1 rad/s.

El controlador tiene como ecuaciones:

$$\dot{x}_1(t) = \frac{K_p}{T_i}(\psi_r(t) - \psi(t))\Big|_{[-\pi,\pi]}$$

$$\delta_c(t) = x_1(t) + K_p T_d (r_r(t) - r(t)) + \frac{1}{K_N}(T_N \alpha_r(t) + r_r(t)) + K_p (\psi_r(t) - \psi(t))\Big|_{[-\pi,\pi]}$$

Al igual que con el modelo de referencia, utilizaremos un control discreto equivalente con periodo de muestreo de 5 segundos. Utilizaremos los valores de parámetros siguientes: Tn = 107.3, Kn=0.185, Kp=2, Td=50, Ti=100.

2.2. Implementación del barco

Nuestra implementación en java del barco consiste en un modelo acoplado que contiene los siguientes modelos atómicos:

- PeticionesState: Módulo encargado de realizar las peticiones al barco desde la interfaz de usuario, en la versión final de nuestra aplicación no es utilizada, pero sería de mucha utilidad en caso de cualquier tipo de interacción con el usuario, como por ejemplo darle nuevos puntos a un barco de forma manual.

- Receptor: Módulo encargado de recoger los datos que pidiese el usuario y mostrarlos por pantalla, al igual que PeticionesState en la versión final no es usado.

- BarcoState: Contiene el comportamiento del barco propiamente dicho, esto es, como reacciona a un cambio de rumbo, de velocidad, etc.

- ModeloReferencia: Módulo encargado del suavizado del comportamiento del barco, para obtener un control más refinado.

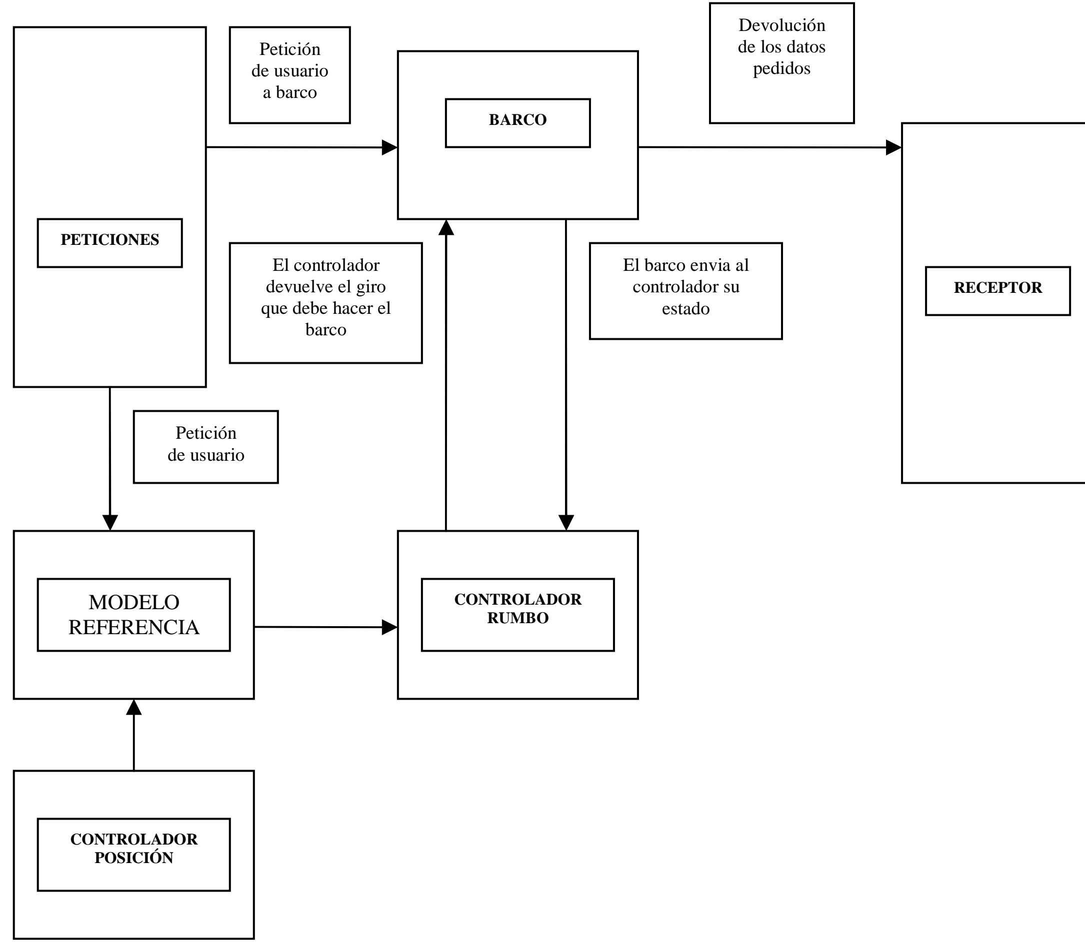
En el barco además hay dos tipos de control:

- ControladorRumboBarco: dadas unas condiciones del barco y un rumbo, llevarlo a dicho rumbo

- ControladorBarco: éste controlador se aprovecha de que el controlador de rumbo ya puede darle el rumbo deseado, para irle ordenando un rumbo u otro y de este modo, poder llegar a un punto destino. El controlador del barco envía una señal cuando considera que necesita un nuevo punto, de este modo,

el encargado (El controlador de la simulación en nuestro caso), podrá juzgar cuando debe entregarle otro punto.

El esquema que hemos utilizado para un barco es muy similar al del avión, siendo como se muestra a continuación:

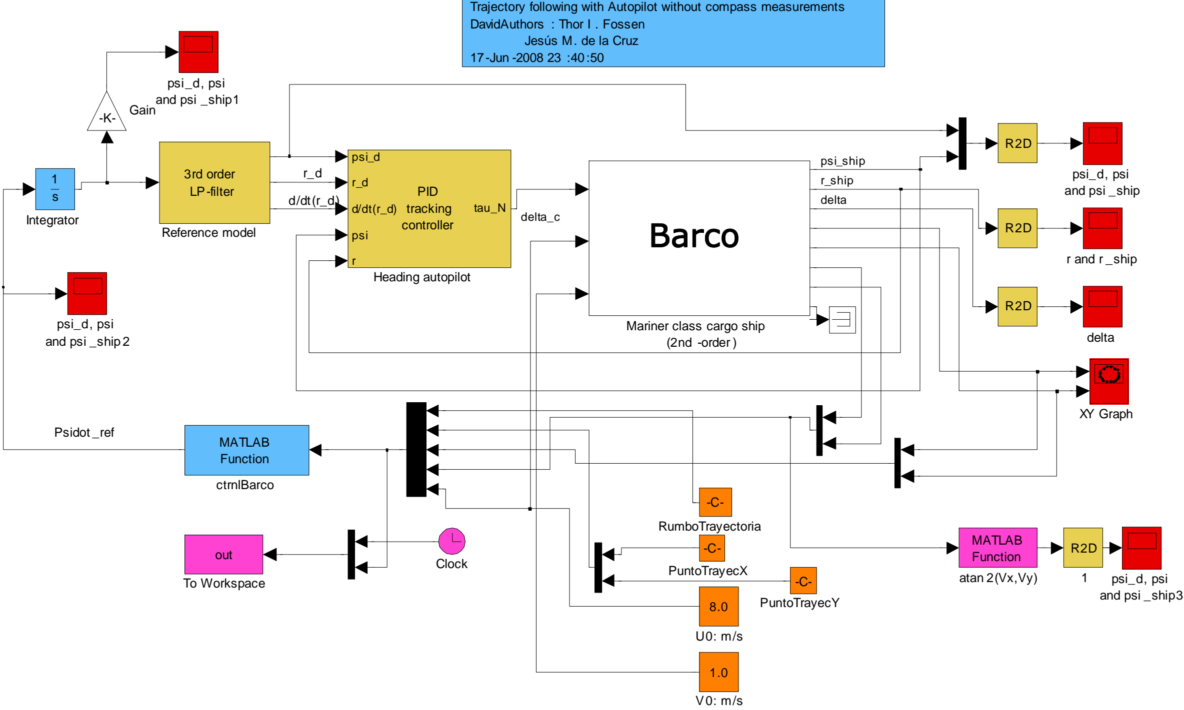


El barco además tiene un identificador único, que le identifica, de este modo, cuando un barco se mueve y dicho movimiento llega al controlador, el controlador es consciente de que el barco que se ha movido es éste y no otro que estuviera muy cerca.

Las conexiones entre los modelos internos de éste modelo son:
EIC:
De barco a BarcoState y a controlador
IC:
De Controlador a BarcoState, ModeloReferencia,
De petición a controlador y barco
De BarcoState a Controlador y ControladorRumbo

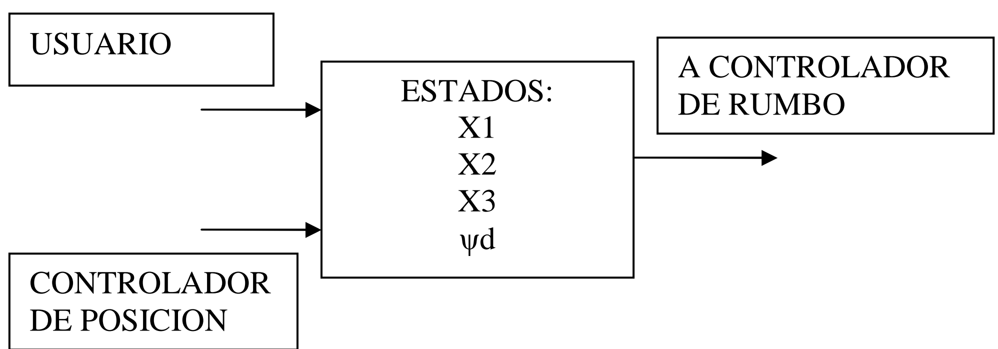
De ModeloReferencia a ControladorRumbo
De ControladorRumbo a BarcoState

Pueden verse más claramente las conexiones en el modelo simulink que se muestra a continuación:



Dicho modelo simulink, lo usamos para comprobar que el funcionamiento era el adecuado antes de empezar con la programación del modelo en xdevs.

2.2.1. Modelo de referencia



Puertos:

El modelo de referencia tiene un solo puerto de entrada al que llegan tanto las peticiones del usuario, cuando se desea cambiar manualmente el rumbo del barco, como las peticiones del controlador, cada vez que requiere un cambio de rumbo.

Posibles valores de Phase:

Debido a que el modelo de referencia es un sistema de entrada-salida inmediata, que por si mismo no tiene un uso especifico, sino que para una entrada devuelve los valores de rumbo, velocidad angular y aceleración angular deseados, hemos considerado que siempre esta activo.

Estado inicial:

El estado inicial, sería activo con rumbo cero.

Función de transición interna:

El modelo de referencia integraría los modelos con un sigma de 5 y un periodo de integración de también 5 segundos.

Función de transición externa:

La función de transición externa lo que produciría sería un cambio en el valor del ángulo objetivo que deseamos alcanzar, esto es, en el ψd y pondría el tiempo de la simulación del modelo de referencia a cero. De este modo, la próxima función de transición interna integraría con el nuevo valor del ángulo.

Función lambda:

Calculamos en función de x1,x2,x3 los valores del rumbo deseado, la velocidad angular deseada y la aceleración angular deseada y la enviariamos al controlador.

Estados: Wr, dt, tactual, Zetar, x1,x2, x3
Obtenemos el valor de la derivada en la función dameDerivadas

```
derivadas[0]=x2;
derivadas[1]=(-(wr*wr)*x1-2*wr*zr*x2+x3);
derivadas[2]=(-wr*x3+wr*wr*wr*psid);
return derivadas;
```

De este modo al realizar la función lambda, se llama a la función avanzaTiempo() que consiste en lo siguiente.

```
public void avanzaTiempo() {
this.actualizaEstados(integrador.integra(this,
5,this.getStateValue("tactual").doubleValue()));
this.setStateValue("tactual",
this.getStateValue("tactual").doubleValue()+5);
}
```

Dentro de esta función se realiza la integración que llama a la función dameDerivadas del Modelo de referencia, obteniendo las derivadas indicadas arriba, y adquiriendo de este modo los nuevos estados para x1,x2,x3.

Controlador de Rumbo:
Derivadas:

```
vpsir=this.getStateValue("psir").doubleValue();
vpsi=this.getStateValue("psi").doubleValue();
vkp=this.getStateValue("Kp").doubleValue();
vti=this.getStateValue("Ti").doubleValue();
derivadas[0]=(vkp/vti)*(normalizar(vpsir-vpsi));
```

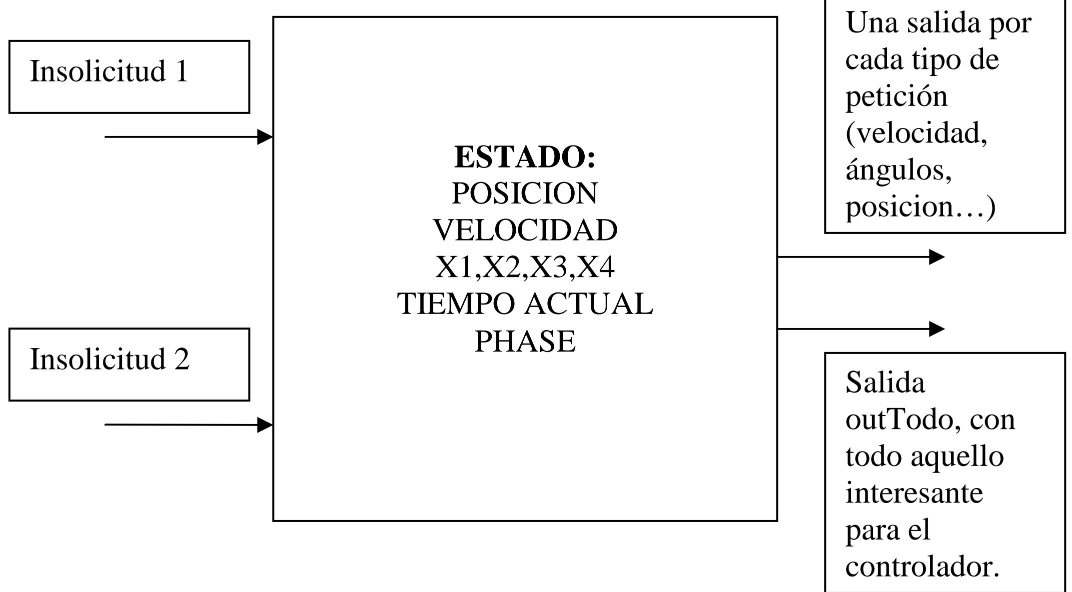
Siendo la función normalizar:

```
public double normalizar(double ángulo){
            double ángulo_normalizado = ángulo;
            if(ángulo > Math.PI){
ángulo_normalizado=normalizar(ángulo-2*Math.PI);
}
            if(ángulo < -Math.PI){
ángulo_normalizado=normalizar(ángulo + 2*Math.PI);}
            return ángulo_normalizado;
            }
}
```

Es decir, es una función que dado un ángulo lo convierte en su ángulo equivalente entre –pi y pi.

2.2.2. Barco

Puertos:



Donde:
X1 representa el ángulo de guiñada.
X2 representa la velocidad de guiñada.
X3 representa la aceleración de guiñada.
X4 representa el ángulo del timón.

Posibles valores de Phase:

Los valores que puede tomar phase son(al igual que vimos en el avión): parada (esto es, no ha sido aun generado), running (barco en movimiento) y destruido

Estado inicial:

La phase del barco en el instante inicial (construcción), es la de barco en estado de parada, esto es, en espera a que se produzca una inicialización del mismo. Le ponemos por tanto sigma infinito, o lo que es lo mismo, lo pasamos a estado passivate.

Función de transición interna:

La función de transición interna se encarga de avanzar el sistema hasta el momento actual.

Función de transición externa:

En caso de recibir alguna petición el barco lo que ocurre es que se le envía al barco la orden de realizar la operación correspondiente.

Función de confluencia:

En caso de que ocurran simultáneamente la función de transición interna y la externa preferimos ejecutar antes la función de transición interna para que el barco continúe evolucionando, ejecutando justo después la función de transición externa.

Función lambda:

La función lambda es la encargada de devolver los estados actualizados en caso de que así se pida.

Debido a problemas que encontramos en el control del barco, estimamos que lo más oportuno era pasarlo a canónica de control, quedando como se muestra a continuación:

```
double a1 = (T1+T2)/(T1*T2);
double a2 = (1)/(T1*T2);
double b1 = (K*T3)/(T1*T2);
double b2 = (K)/(T1*T2);
double[][] A={{1.8712,-0.8722},{1,0}};
double[][] B={{0.0625},{0}};
double[][] C={{0.0571,-0.0541}};
double[][] D={{0}};
ecuacionDif=new mlved(A,B,C,D);
```

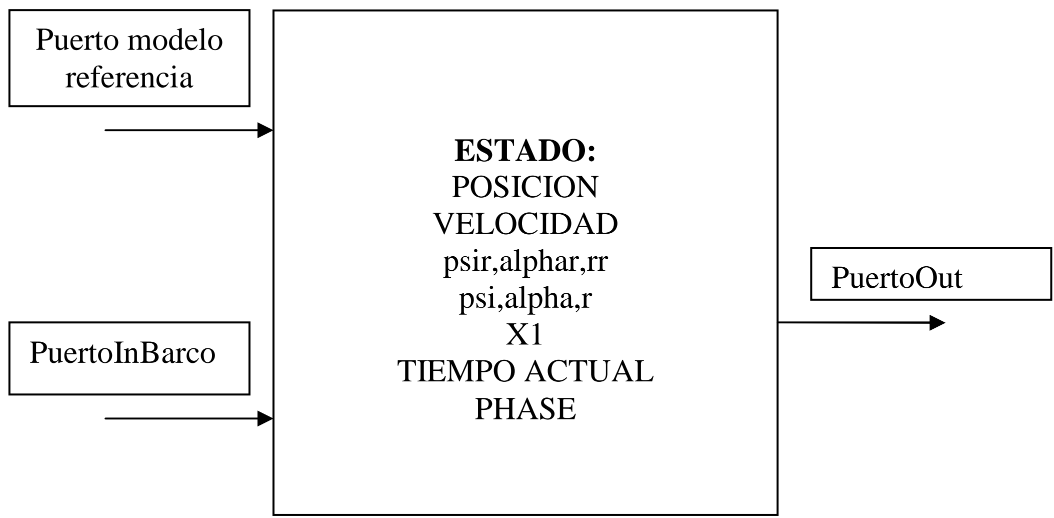
Teniendo la clase mlved un metodo update:

```
public double[][] update(double u){
        double[][] matrizU= mlved.inicializar(1, 1, u);
    y=mlved.sumaMatrices(mlved.multiplicaMatrices(C,               x),
mlved.multiplicaMatrices(D, matrizU));
    x=mlved.sumaMatrices(mlved.multiplicaMatrices(A,               x),
mlved.multiplicaMatrices(B, matrizU));
        return y;
}
```

Así el metodo avanzaTiempo() sería.

```
public void avanzaTiempo() {
        double tiempo = 1;
    this.setStateValue("x2",ecuacionDif.update(this.getStateValue("x
4").doubleValue())[0][0] *tiempo);
    this.actualizaEstados(integrador.integra(this,
tiempo,this.getStateValue("tactual").doubleValue()));
    this.setStateValue("tactual",
this.getStateValue("tactual").doubleValue()+tiempo);        }
```

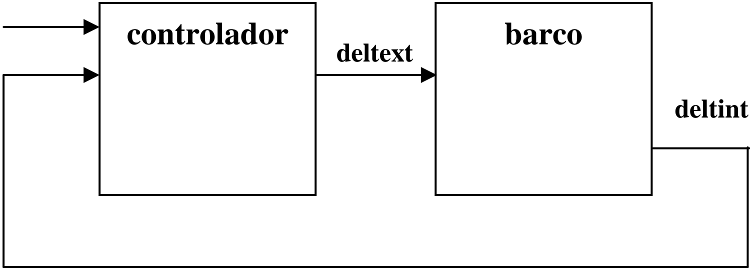
2.2.3. Controlador Rumbo



Resumen del comportamiento:

Mediante el puertoInBarco, llegan las posiciones reales del barco, esto son psi, alpha y r, mediante el puerto del modelo de referencia llegan las entradas del comportamiento esperado para un barco durante un momento dado, estas señales son psir, alphar y rr.

El controlador de rumbo, lo que trata, es de darle al barco de nuevo unas ordenes de modo que cumpla el comportamiento esperado, de este modo se produce una realimentación entre el controlador y el barco.



Posibles valores de Phase:

El controlador va a estar en este caso siempre pasivo, esto es, a la espera de que llegue una nueva señal del modelo de referencia o del barco. En el momento que llega una nueva señal de uno de estos dos elementos, el controlador realiza los cálculos oportunos y envía las órdenes necesarias de nuevo al barco.

Estado inicial:

La phase del controlador de rumbo en el instante inicial consiste en la espera de la llegada de mensajes como ya dijimos antes, esto es, espera en pasivo.

Función de transición interna:

La función de transición interna simplemente vuelve a poner en pasivo el controlador.

Función de transición externa:

La función de transición externa es la encargada de cuando llega un mensaje, producir los cambios oportunos de modo que cree el mensaje que le indique al barco que cambios tiene que realizar para modificar su trayectoria.

Función lambda:

Es la encargada de avanzar el tiempo del controlador y de enviar el mensaje al barco.

2.2.4. Controlador Posición

Puerto In

Puerto In algoritmo

Puerto InBarco

ESTADO:
POSICION
VELOCIDAD
psir,alphar,rr
psi,alpha,r
X1
TIEMPO ACTUAL
PHASE

Puerto Out

Puerto Conexión exterior

Puerto In Peticiones:

Por éste puerto llegan las peticiones de nuevo way-point por parte del usuario.

Puerto In algoritmo:

Por este puerto llegan las peticiones de nuevo way-point por parte del algoritmo.

Puerto In Barco:

Por este puerto llega el estado actualizado del barco para posibilitar el control sobre el mismo.

Puerto Out:

Por este puerto se envía al barco un nuevo mensaje que le indica lo que debe realizar en el siguiente paso para acercarse al control.

Puerto Conexión Exterior:

Por este puerto se pide un nuevo way-point al algoritmo.

Posibles valores de Phase:

Los posibles valores de la phase del controlador son los mismos que los del controlador del avión: no_iniciado (equivalente a parada), iniciado (equivalente a running) o destruido.

Estado inicial:

El estado inicial, es también similar al empleado para el controlador del avión, es decir, en este caso tendríamos phase no_iniciado y sigma que sería nuevamente infinito.

Función de transición interna:

Al igual que hacíamos en el controlador del avión, con los datos actualizados de la última transición externa, el controlador calcularía el próximo evento que necesitaría hacer el barco y añadiríamos al mensaje aquella información que debiésemos enviarle al barco.

Además tenemos que tener en cuenta, que puede ser necesario que pidamos un nuevo punto de ruta, para tener control sobre los puntos de ruta tenemos que tener en cuenta dos cosas:

1. Tenemos que fijar un tiempo mínimo.
2. Tenemos que fijar un tiempo máximo.

Estos dos puntos los solucionamos del mismo modo que en el avión.

Función lambda:

La función lambda simplemente se encarga de enviar el mensaje con el contenido que le haya incluido la función de transición interna.

Función de confluencia:

De nuevo ejecutamos primero la función de transición interna y luego la externa.

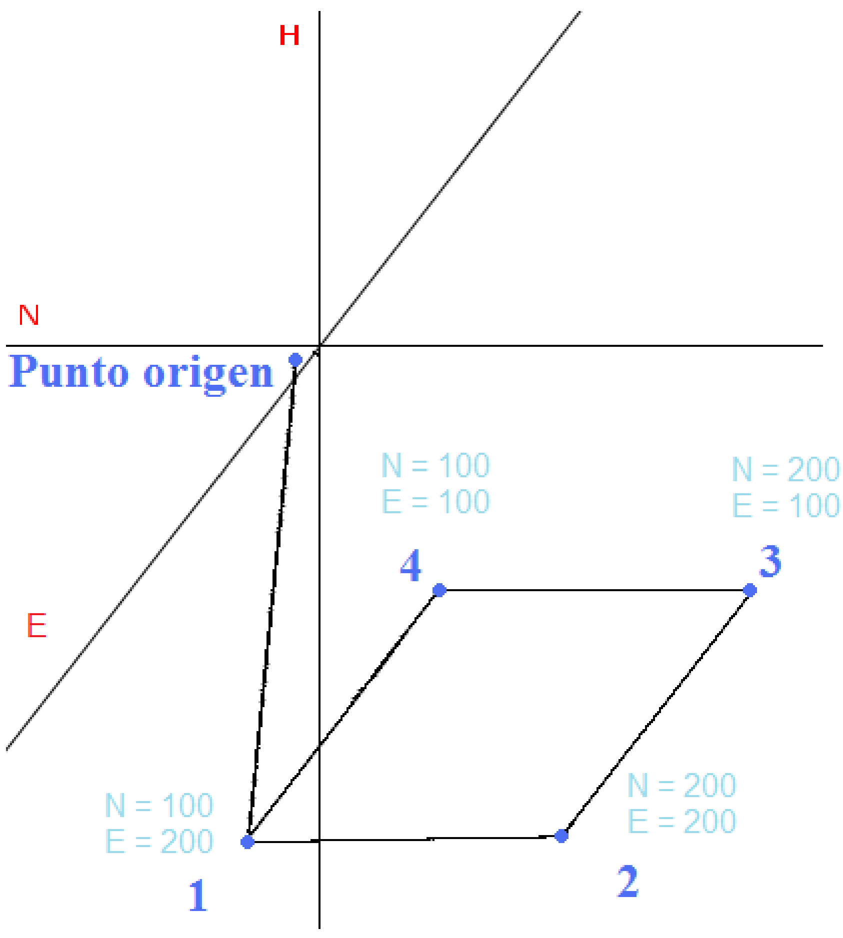
Tema 3: Pruebas de control de rumbo de los vehículos

Las pruebas de control de rumbo las hemos realizado con el primer prototipo de vista de nuestro proyecto.

Podemos observar como en todos los rumbos, primero, el vehículo se acerca a la posición inicial de las distintas figuras siguiendo luego correctamente los distintos way-points.

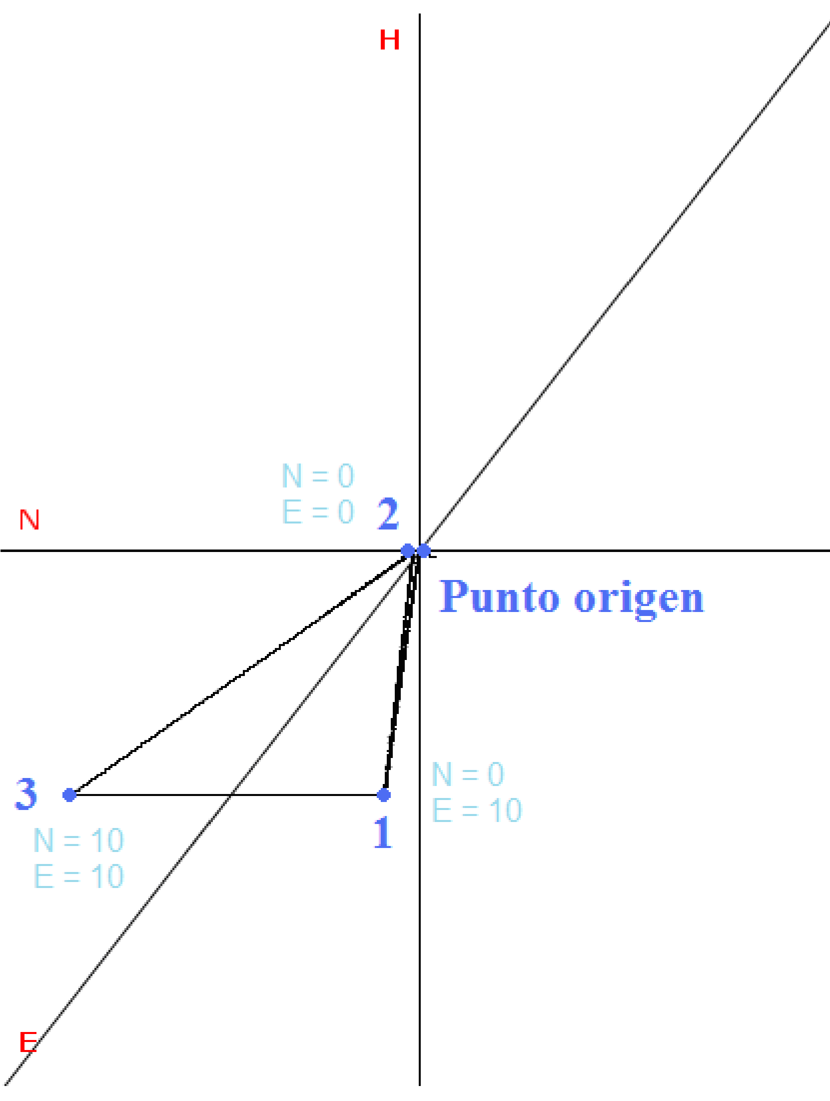
3.1. Rumbo en cuadrado

En la siguiente imagen podemos observar un vehículo realizando un recorrido con trayectoria de cuadrado en el plano N-E. Las coordenadas de los puntos de la trayectoria se encuentran en kilómetros.



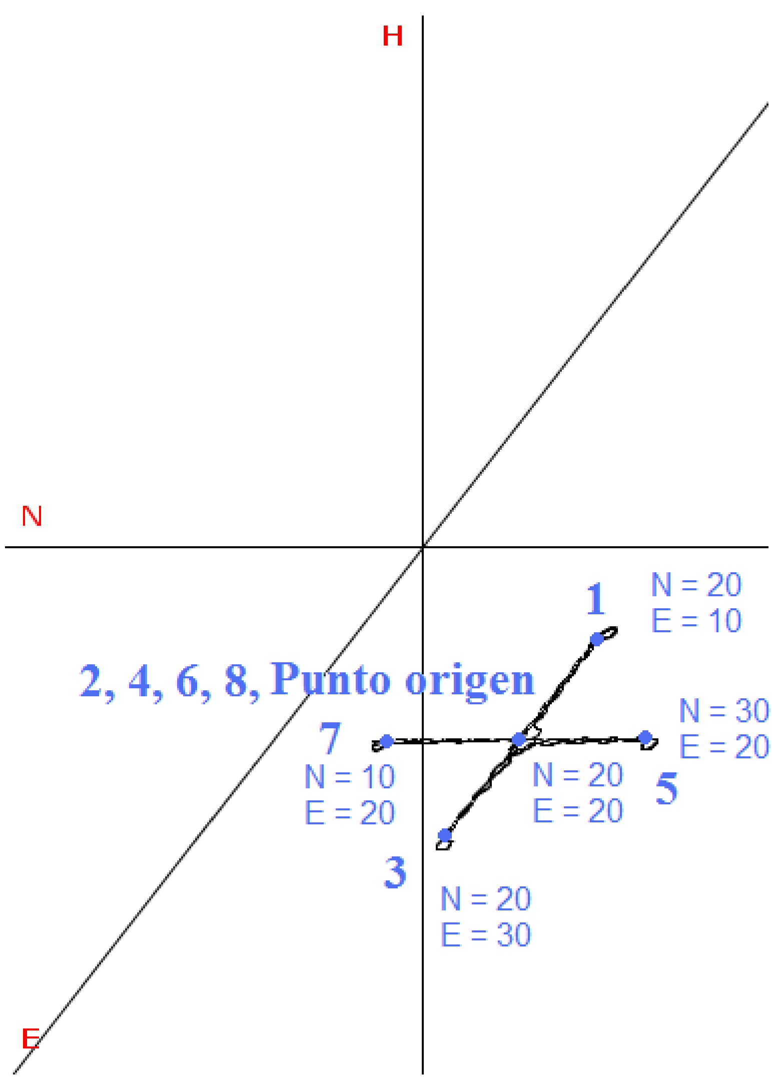
3.2. Rumbo en triángulo

Para la siguiente prueba optamos por una ruta en triángulo en el plano N-E. El orden de recorrido de los puntos de la ruta se encuentran en la imagen y las coordenadas, como en el ejemplo anterior se encuentran en kilómetros.



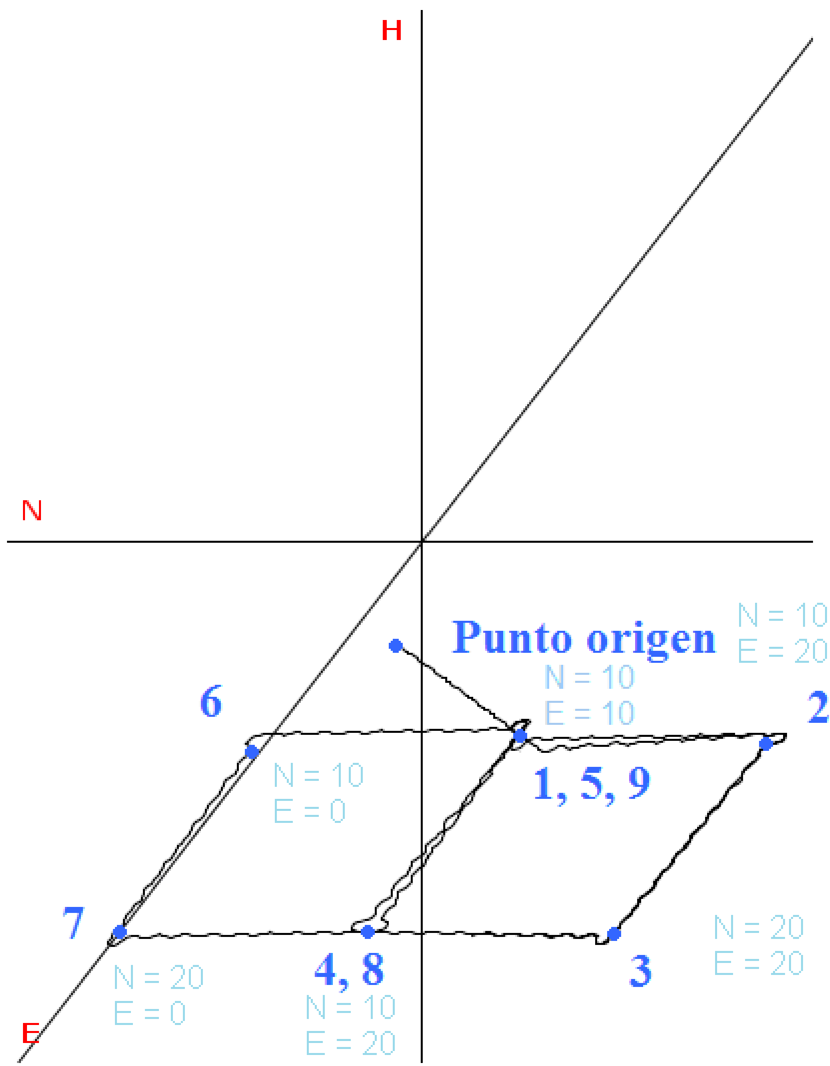
3.3. Rumbo en cruz

Para esta prueba utilizamos una trayectoria en forma de cruz. Las coordenadas se encuentran en km. Los puntos 2, 4, 6 y 8 coinciden con el punto de origen, puesto que el avión vuelve a pasar por ahí para realizar la cruz.



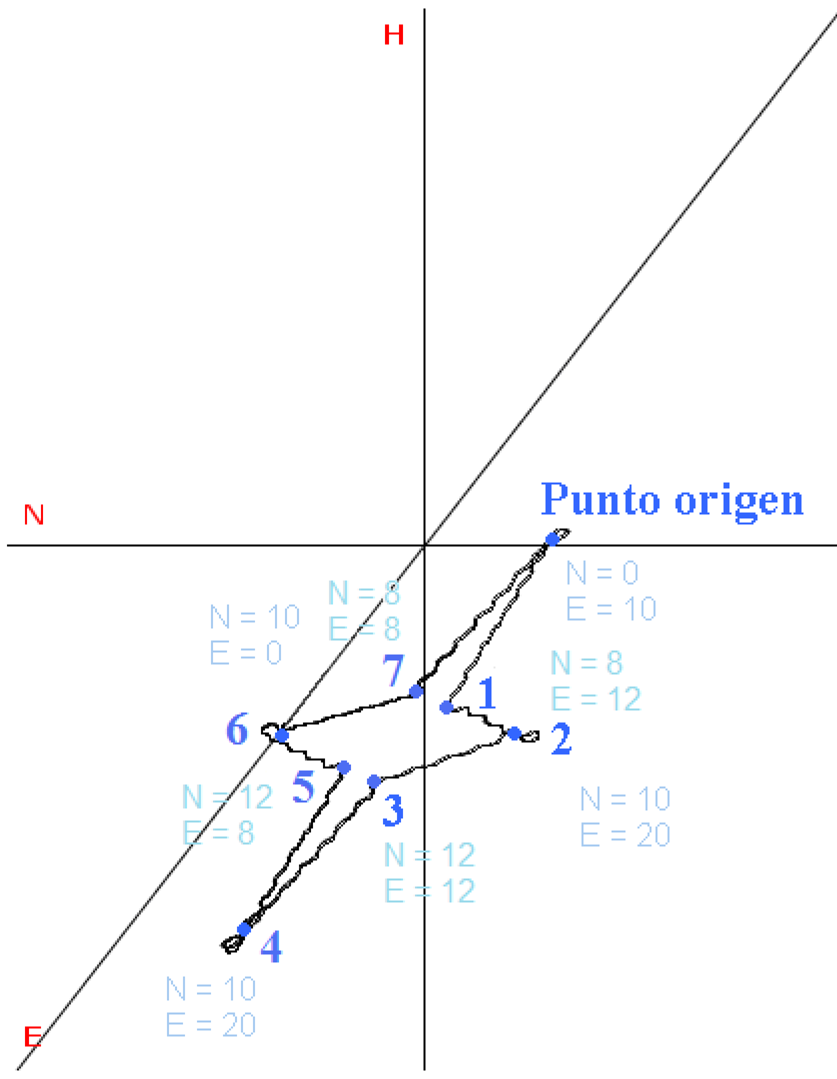
3.4. Ruta en ocho

Para realizar una prueba compleja, utilizamos una nueva ruta en forma de 8, obteniendo el siguiente resultado (el orden de recorrido de los puntos y las coordenadas se encuentran indicadas en la grafica):



3.5. Ruta en estrella

Para la última prueba, escogemos una trayectoria dada por una ruta con puntos en estrella. El resultado de la simulación es el siguiente:



Tema 4: Modelos de los componentes

4.1. Modelo del viento y oleaje

El viento induce un oleaje que desplazará un objeto situado sobre la superficie mediante la ecuación:

Uw(t) = Vw(t) cos Ψw(t)

Vw(t) = Vw(t) cos Ψw(t)

Donde Vw representa la celeridad con la que se mueve el objeto, y que será una cierta fracción de la velocidad del viento a, por ejemplo 10 metros de altura, y Ψw es el rumbo que lleva el viento y las olas. No tenemos información detallada de cómo son estos valores, por lo que debemos hacer suposiciones. La más sencilla es que ambos valores son constantes, lo que dará un comportamiento determinista. Para introducir un comportamiento estocástico podemos proceder de distintas formas. En fossen [2, pp. 137], se utiliza un módulo para las fuerzas como un proceso de Markov de 2º orden junto con una deriva aleatoria. Introduce un cero en el origen. Este mismo modelo lo utiliza en la librería gnc desarrollada por Trodheim para modelar las perturbaciones de rumbo en el barco Mariner. Sin embargo, nosotros eliminamos el cero en el origen para generar el rumbo. De modo que el modelo que usamos es:

Donde Ww y Nw son señales de ruido blanco gausiano; dw representa la deriva, y el valor inicial de su integrador nos daría el rumbo dominante; σ es la ganancia del espectro de las olas, w0 es la frecuencia de pico del espectro, λ es el amortiguamiento relativo y K es la amplitud de la ola. Podemos usar los valores.

σ = 0.5 m, w0 = 1.2 rad/seg, λ =0.1, K=3

El rumbo está dado en grados. El rumbo dominante del viento es el que fijemos: [0º, 360º].El ruido blanco gaussiano es en ambos cados de media nula y varianza 10 (podemos modificarlo y ver distintos supuestos).

La velocidad del viento se puede dar como una señal aleatoria de ruido blanco con una media y unas varianzas adecuadas, o bien como un proceso de Markov de primer orden. Podemos considerar lo primero que es más fácil. Para el proceso de Markov procederíamos como veremos en las corrientes.

$$\psi_w(s) = \frac{2\lambda w_o \sigma K}{s^2 + 2\lambda w_o s + w_o^2} w_w + d_w$$

$$\dot{d}_w = n_w$$

4.2. Modelo de corrientes

Las corrientes tienen un modelo similar al del viento:

$U(t)_c = V_c(t) \cos \Psi_c(t)$

$V(t)_c = V_c(t) \sin \Psi_c(t)$

Donde V_c es la velocidad de la corriente y Ψ_c es el rumbo. Al igual que con el viento podemos utilizar modelos muy sencillos de velocidad y rumbo constante, o bien con características aleatorias (función de distribución) dadas.

En Fossen [2, pp. 138] se utiliza el siguiente modelo: la velocidad se considera un proceso de Markov de primer orden:

$dV_c + \mu V_c = w_c$

donde w_c es un ruido blanco gaussiano y μ (mayor o igual a cero) es una constante. El integrador debe tener un saturador para limitar la velocidad entre unos valores máximo y mínimo, por ejemplo:

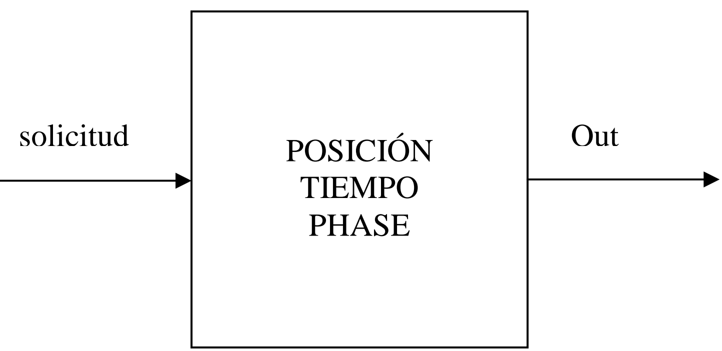
$V_{min} <= V_c <= V_{max}$

Con V_{min} = 0.5 m/s, V_{max} = 3 m/s.

Al rumbo Ψ_c le podemos asociar también una dinámica. Podemos usar una función determinista de la posición, una función de distribución de probabilidad determinada, o hacerle un proceso de Markov de 1º o 2º orden. Quizás lo mejor sea hacerle una función determinista de posición.

4.3. Modelo e implementación de los náufragos

La dinámica del náufrago corresponde a la suma de las acciones de los vientos y las corrientes. Su implementación se realiza de la siguiente forma:



Puertos

Solicitud: Cambio externo que se realiza en alguna de las características del náufrago, como la corriente o la posición.

Out: Por este puerto el náufrago envía su estado actualizado.

Posibles estados:
Un náufrago puede estar encontrado o no_encontrado, si esta encontrado, es que ya ha sido rescatado por los vehículos de rescate, sino, es que los vehículos lo están buscando.

Función de transición interna:
Se avanza en la integración del modelo, esto es, como progresaría el náufrago en el mar.

Función de transición externa:
Se realiza la petición realizada desde el exterior, como por ejemplo, cambiar la posición o rescatar a un náufrago.

Función lambda:
Encargada de enviar el mensaje con el estado del barco.

Los náufragos, en nuestra implementación son unos modelos atómicos que constan de lo siguiente:

- Un modelo de Corrientes
- Un modelo de Mar

Las variables que indican su estado.

El tiempo actual de la simulación.
```
public static final String tactual = "tactual";
```

Posicion norte del náufrago
```
public static final String n = "n";
```

Posicion Este del náufrago
```
public static final String e = "e";
```

El tiempo de integración
```
public static final String tiempo ="tiempo";
```

El tiempo inicial de simulacion
```
public static final String tinicial = "tinicial";
```

La fase del náufrago, esto es, si ha sido inicializado o no.
```
public static final String Phase = "phase";
```

Las derivadas de este modelo son las componentes de la corriente:

```
derivadas[0]=corriente.dameUc();
derivadas[1]=corriente.dameVc();
```

Así en la función AvanzaTiempo se produce la integración de las ecuaciones diferenciales.
```
public void avanzaTiempo() {
      double dtiempo=30;
      corriente.avanzaTiempo(dtiempo);
      this.actualizaEstados(integrador.integra(this,
      dtiempo,this.getStateValue("tactual").doubleValue()));
this.setStateValue(tactual,
this.getStateValue(tactual).doubleValue()+dtiempo);
}
```

En nuestra implementación, el mar es una función de la posición, es por tanto que una vez sabemos en que posición estamos, pedimos la corriente marina, que la añadimos al oleaje obteniendo el oleaje actual de dicha posición.

Esto podemos verlo en la función de transición interna:

```
public void deltint() {
	if(this.getStateValue("phase").intValue()==Náufrago.ON){
		mar.avanzaTiempo();
		corriente.ponOleaje(mar.dameSalidaActual());
		this.avanzaTiempo();
```

En nuestra aplicación hemos creado distintos tipos de corriente, todos bastante sencillos para observar el funcionamiento en los distintos tipos de corriente.

Estas distintas corrientes son fácilmente modificables, debido a que es una nueva clase denominada FunciónCorriente.

Veamos un ejemplo:

```
public class FunciónCorriente {
	public static double constante0 =0;
	public static double constante1 =1;
	public static double variable=2;
	public static double loco=4;
public static double dameCorriente(double tipoCorriente,double x, double y) {
		double devolver =0;
		if(tipoCorriente==0){
			devolver=0;
		}
		else if(tipoCorriente==1){
			devolver=1;
		}
		else if(tipoCorriente==2){
			devolver= dameCorrienteCasilla(x,y);
		}
		else if(tipoCorriente==4){
			Random randomizador=new Random();
			devolver = 3;
		}
		return devolver;
	}
	private static double dameCorrienteCasilla(double x, double y){
		double corriente=0;
		int casillaX = (int)(x/10)+1;
		int casillaY = (int)(y/10)+1;
		corriente = (3*casillaX);
		return corriente;
	}
	private static double mpipi(double valor){
		if(valor<-Math.PI){valor=mpipi(valor+2*Math.PI);}
		if(valor>Math.PI){valor=mpipi(valor-2*Math.PI);}
		return valor;
	}
}
```

De este modo, podemos definir nuevos tipos de corrientes y añadirlos fácilmente.

La clase corriente tiene por tanto una FunciónCorriente , y las derivadas mostradas a continuación:

```
public double[] dameDerivadas(double tiempo, double[] estados,
```

```
            double[] control) {
            double[] derivadas=new double[1];
            Random randomizador=new Random();
            derivadas[0]=randomizador.nextGaussian()-
mu*velocidad_actual;
            return derivadas;}
```

que representa la ecuación:

$dVc + \mu Vc = w_c$

4.4. Mar

El avanzar tiempo del mar, se realiza integrando y habiendo actualizado w con un correspondiente ruido gaussiano, es decir:

```
      public void avanzaTiempo(){
            w=randomizador.nextGaussian();
      this.actualizaEstados(integrador.integra(this,
this.tiempo_integracion,tiempo_actual));
            tiempo_actual+=tiempo_integracion;
      }
```

Las derivadas que se corresponden con el filtro de segundo orden son las siguientes:

```
      public double[] dameDerivadas(double tiempo,double[]
      estados,double[] control){
            double[] derivadas = new double[3];
            derivadas[0]=estados[1];
            derivadas[1]=3*2*amortig_relativo*frec_pico_espectro*
            gan_espectro_olas*w -   2*amortig_relativo*
            frec_pico_espectro* estados[1]- frec_pico_espectro*
            frec_pico_espectro * estados[0] + dw;
            derivadas[2]=randomizador.nextGaussian();
      return derivadas;
      }
```

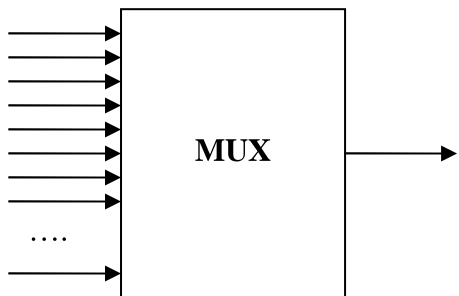
Que se corresponde con el paso del filtro a una ecuación diferencial, esto es:

$y'' + 2 \lambda w_o y + w_o^2 y = 2 \lambda w_o \sigma w(t)$

Tema 5: Otros modelos Devs

5.1. Mux

Este módulo tiene n entradas y solo una salida.



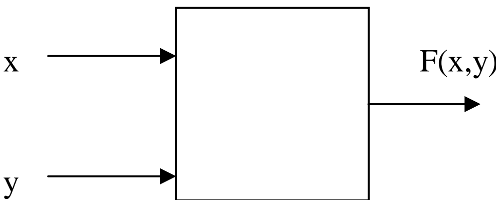
Una limitación de momento de xdevs, es que tiene problema en la unión de varios puertos de salida a un mismo puerto de entrada, es por ello que creamos este módulo al que denominamos multiplexor (o mux abreviadamente) debido a su semejanza con el comportamiento del multiplexor electrónico.

El funcionamiento de este módulo consiste en que cada vez que se produce una transición externa, esto es, cada vez que entra un dato por uno de los puertos de entrada, el mux recorre todos los puertos de entrada (debido a que es posible que hayan llegado varias transiciones externas simultáneamente) ,guardando todas los datos que hayan llegado en las transiciones externas conjuntamente con el puerto en el que se ha producido ésta transición en una lista. Ésta lista tiene objetos de tipo DatoMux, que es simplemente una clase que contiene dos atributos, un entero que indica el número de puerto en el que se ha producido y un Object que contiene el elemento que ha llegado por dicho puerto, de este modo, el multiplexor es independiente de la representación de los elementos, simplemente, le llegan unos Object y los devuelve unidos en una lista junto con el puerto por el que han llegado.

Nos pareció interesante implementar el multiplexor, debido a que creímos que había en ocasiones que podía resultar interesante implementar de alguna forma la llegada de muchos puertos a uno, y no que la relación entre puertos siempre fuera de 1 a 1, o de 1 a muchos, pero nunca de muchos a 1.

5.2. Operador

Un nuevo módulo que hemos realizado debido a que hemos considerado interesante es el operador binario. Éste módulo posee dos puertos de entrada y uno de salida.



El resultado del valor obtenido en el método de salida, depende tanto de los valores de las entradas como de la función seleccionada para el operador. Las funciones posibles deben de implementar la interfaz IOperadorBinario, que tiene una función:

```
public Double opera(Double a, Double b);
```

El operador que hemos usado nosotros es un operador, que dadas dos entradas, devuelve como salida la suma de ambas, esto es F = + , lo que implicaría F(x,y) = x+y, pero sería del mismo modo muy sencillo crear cualquier tipo de operación, numérica, ya fuera el logaritmo en base x de y, la raíz x de y, o lo que quisiéramos, puesto que el bloque procesaría del mismo modo las entradas, llamaría a la correspondiente función y enviaría el resultado por la salida.

Nuestra clase, función sumador binario, tendría tan poca complejidad como mostramos a continuación:

```
public class SumadorBinario implements IOperadorBinario{
	public SumadorBinario(){
	}
	public Double opera(Double a, Double b) {
		return a+b;
	}
}
```

Tema 6: Integración numérica de los modelos

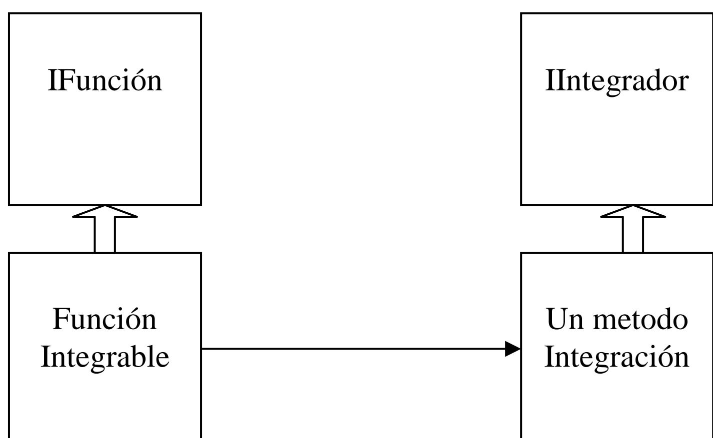
Para la integración de los elementos móviles hemos implementado el patrón Strategy, que le da una mayor capacidad de cambio a los modelos.

Todos nuestros modelos que necesitan ser integrados (avión, barco, etc) implementan la interfaz IFunción.

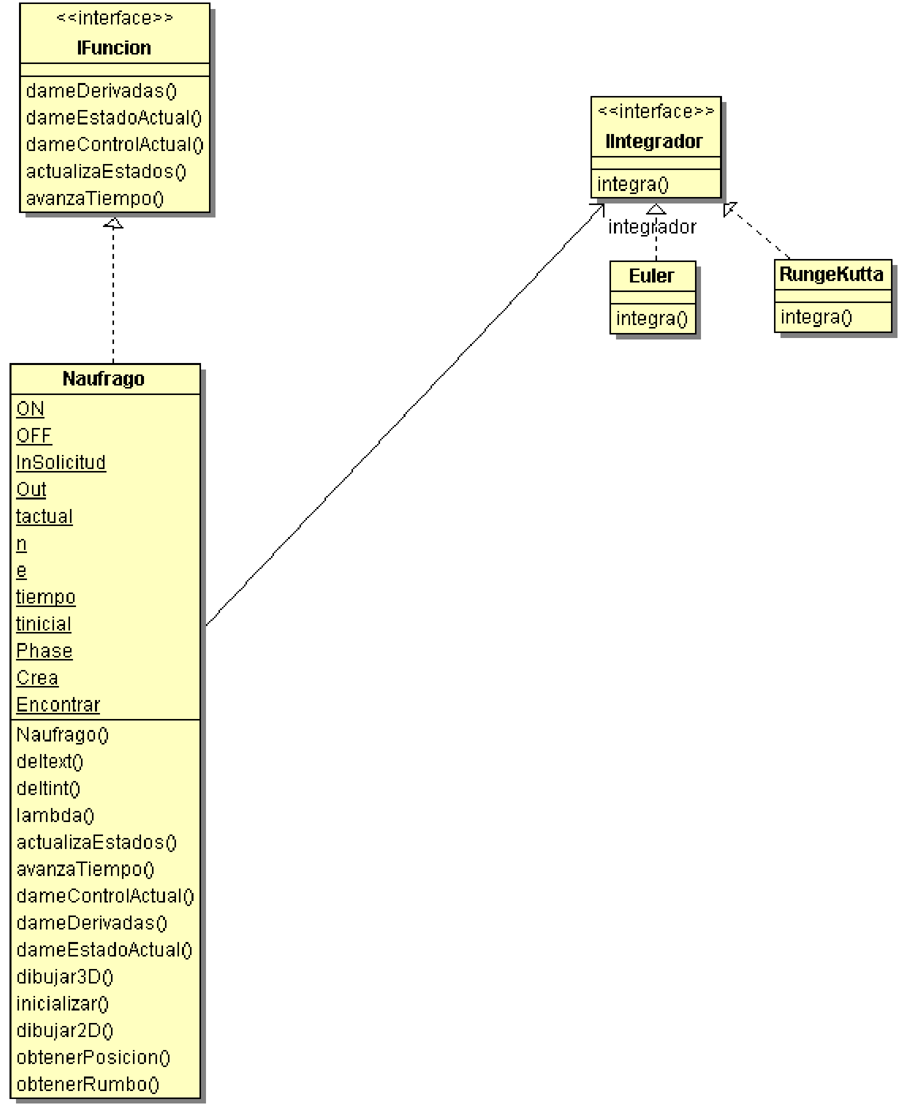
Esta interfaz tiene los siguientes métodos:

```
public double[] dameDerivadas(double tiempo,double[] estados,double[] control);
public double[] dameEstadoActual();
public double[] dameControlActual();
public void actualizaEstados(double[] estadosActuales);
public void avanzaTiempo();
```

La función dameControlActual, finalmente no la hemos utilizado, pero las demás, son necesarias debido a que toda función integrable tiene que tener un integrador que usa dichas funciones, es decir.



Veamos por ejemplo, el diagrama UML de cómo quedaría el diseño de la integración de los náufragos:



La interfaz IIntegrador tiene una única función que es:

```
public double[] integra(IFunción función,double tiempo,double tiempo_actual);
```

Esto es, dando la función el tiempo de integración y el tiempo actual, el integrador devuelve el valor de los nuevos estados integrados.

De este modo sería muy sencilla la creación de nuevos métodos de integración para nuestras funciones, nosotros hemos implementado dos métodos numéricos clásicos:

Euler y Runge-Kutta.

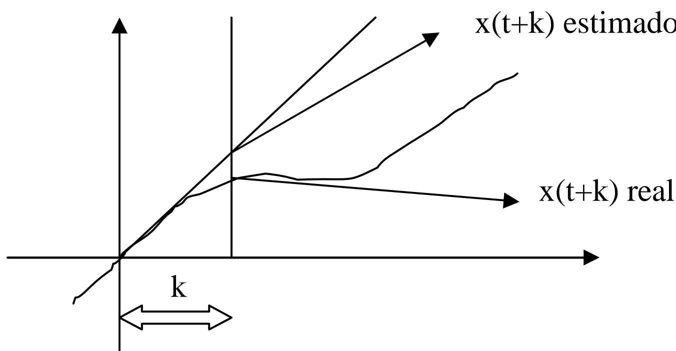
6.1. Método numérico de Euler

La idea del método de Euler es muy sencilla, y esta basada en la noción de lo que es la derivada.

Una derivada seria la tangente a una función en un punto.

El método de Euler lo que hace es, dada una función x y su derivada dx, predice que la derivada no va a variar en un intervalo de tiempo prefijado, esto hace que el incremento sea igual a la derivada multiplicada por el intervalo de tiempo. Es decir:

$$x(t+k) = x(t) + dx(t)*k$$



Una vez obtenido x(t+k) (o el vector de soluciones, en caso de ser más de una), podríamos obtener de nuevo las derivadas en un punto dado, mediante las ecuaciones diferenciales de los modelos.

La ventaja de este método es su gran sencillez, esto hizo que en un principio trabajáramos con él, ya que nos permitía solucionar ecuaciones diferenciales de una manera muy fácil, sin embargo, si queríamos aumentar los periodos de integración para acelerar la simulación por ejemplo mientras se mueven inicialmente los náufragos, pensamos que este método no sería los suficientemente robusto, esto es, que el error sería demasiado grande, debido a que en este método si k es muy pequeño, el funcionamiento es exactamente el mismo que el de la derivada, el problema es que a medida que aumenta k, el error es mucho mayor,

6.2. Método numérico de Runge-Kutta

Los métodos de Runge-Kutta son una familia de métodos muchos más precisos que el método de Euler, debido a que consiste en coger varias derivadas y asignarles un distinto peso a cada una de ellas, de este modo se puede compensar el resultado de una derivada e intentar predecir el futuro comportamiento de la ecuación.

El método numérico de Runge-Kutta de nivel cuatro viene dado por:

$$y_{n+1} = y_n + \frac{h}{6}(k_1 + 2k_2 + 2k_3 + k_4)$$

$$k_1 = f\left(t_n, y_n\right)$$

$$k_2 = f\left(t_n + \frac{h}{2}, y_n + \frac{h}{2}k_1\right)$$

$$k_3 = f\left(t_n + \frac{h}{2}, y_n + \frac{h}{2}k_2\right)$$

$$k_4 = f\left(t_n + h, y_n + hk_3\right)$$

$$\text{pendiente} = \frac{k_1 + 2k_2 + 2k_3 + k_4}{6}.$$

De este modo del mismo modo que conseguíamos con Euler, ahora podemos obtener una nueva pendiente asignando un mayor peso a unos elementos que a otros, de este modo, ahora nos basamos en cuatro derivadas y no solamente en una, consiguiendo una mayor robustez y un menor error en los cálculos, lo que nos posibilita un incremento de tiempo mayor del que podíamos con Euler.

Finalmente, aunque están implementados los dos métodos de integración, y sería muy fácilmente expandible añadiendo nuevos métodos, todos los modelos integrados tienen añadidos en su constructor un integrador del tipo Runge-Kutta, puesto que lo hemos considerado suficientemente bueno.

Tema 7: Estructura de la simulación

7.1. Introducción

El controlador es un elemento que está comunicado con todos los aviones, barcos y náufragos, de este modo, tiene conocimiento en todo momento de la situación de todos ellos. Una función del controlador sería por tanto, hacer que cuando un barco encontrase a un náufrago, esté cambiase su estado, dejando por tanto de evolucionar, es decir moverse.

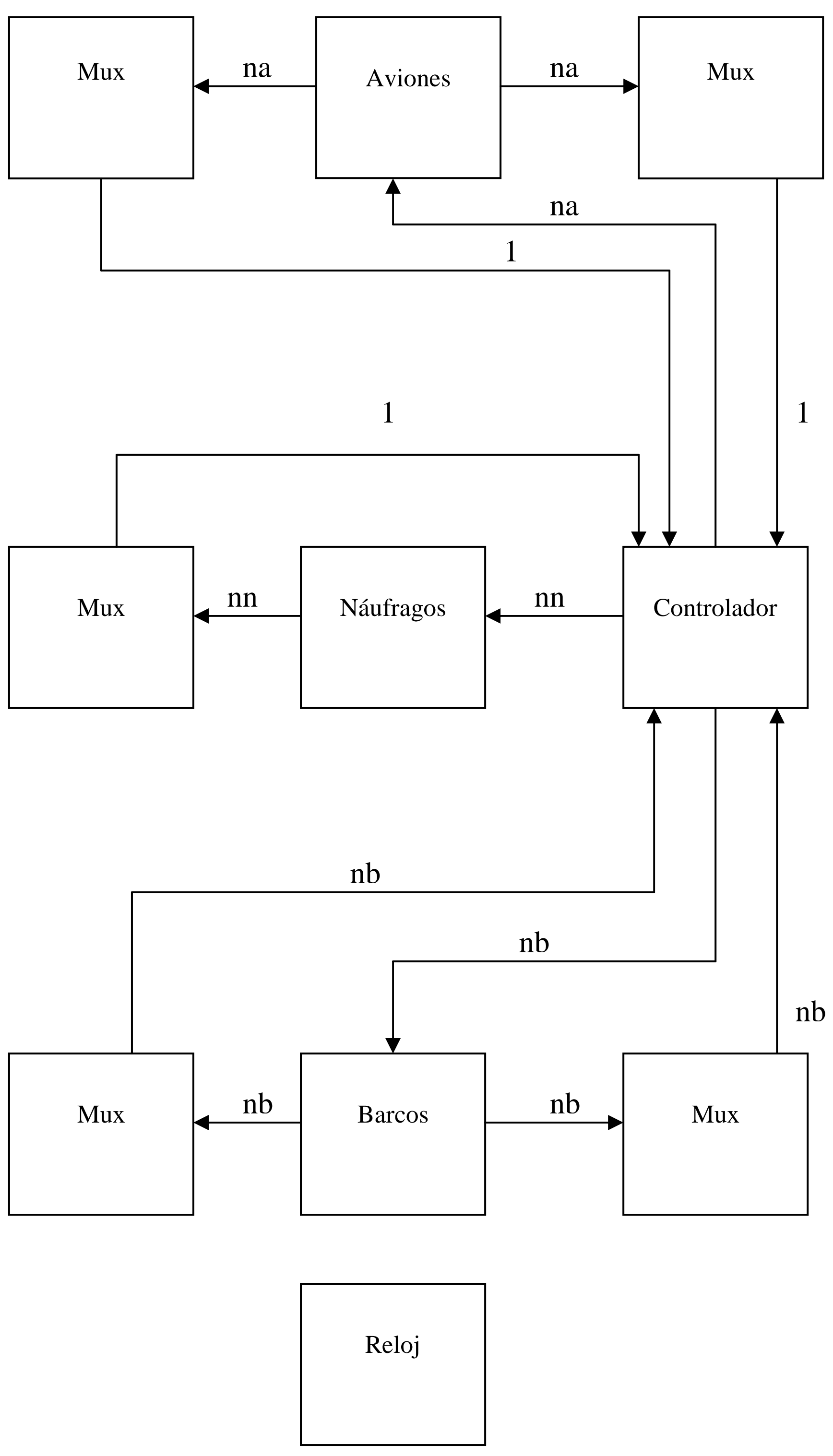
Para la realización de ésta comunicación entre el controlador con los distintos elementos hemos preferido que el controlador tenga unos puertos fijos, esto es, que el numero de puertos del controlador no dependa del numero de aviones, barcos y náufragos. Para llevar a cabo este tipo de implementación con xDevs, nos ha sido necesaria la creación de un nuevo modulo, al que hemos llamado mux (multiplexor) debido a las semejanzas que tiene con los multiplexores reales, debido a que como explicaremos más adelante, cuando llega una señal de entrada a cualquiera de sus entradas, se produce una señal de salida que contiene una lista con todas las señales que se han producido simultáneamente, de este modo, luego podemos obtener en que puertos se han producido señales, y cuáles han sido esas señales.

Los multiplexores los utilizamos para unir las salidas de los aviones, controladores de los aviones, barcos, controladores de barcos y náufragos con el controlador de la simulación.

La comunicación entre los móviles y el controlador de la simulación se produce siempre incluyendo el nombre del móvil (un número único en su tipo de móvil) que unido a que sabemos que tipo de móvil es (náufrago, barco, avión) debido a que sabemos por que puerto llega, nos da la información necesaria para saber a quién nos referimos.

La simulación en xDevs se produce a una alta velocidad, nuestro reloj es el encargado de parar la simulación cada cierto tiempo, así en función del tiempo del reloj la simulación será más o menos rápida.

7.2. Esquema de la simulación



7.3. Controlador

La clase controlador es la encargada de gestionar los vehículos entre los distintos vehículos, o en caso de otro tipo de misiones, sería la encargada de gestionar por ejemplo los choques de los vehículos con el suelo, en un principio incluimos esta posibilidad, de hecho el controlador posee un terreno, pero finalmente al ser misiones en mar abierto lo consideramos poco útil.

Todos los vehículos envían al moverse un mensaje al controlador, de este modo, el controlador tiene en todo el momento una lista, que es un objeto de la clase ListaPosicion, que contiene una lista de DatosTipoPosición, que contiene información sobre el tipo de vehículo, el nombre del vehículo, la posición y la ruta. ListaPosición además posee una función de filtro por tipo de vehículo (si queremos obtener una lista por ejemplo con todos los náufragos), una función de búsqueda por tipo de vehículo y nombre del vehículo, de este modo, dado un vehículo podemos saber que posición ocupa, o funciones de inserción y eliminación.

La clase controlador además tiene las probabilidades que vamos obteniendo a medida que nos vamos encontrando a los náufragos. Las probabilidades que vamos a usar son 2, el ángulo y la velocidad, éstas probabilidades van a ser distribuciones normales con una media y una varianza, que vamos modificando a medida que nos encontramos náufragos en una u otra posición.

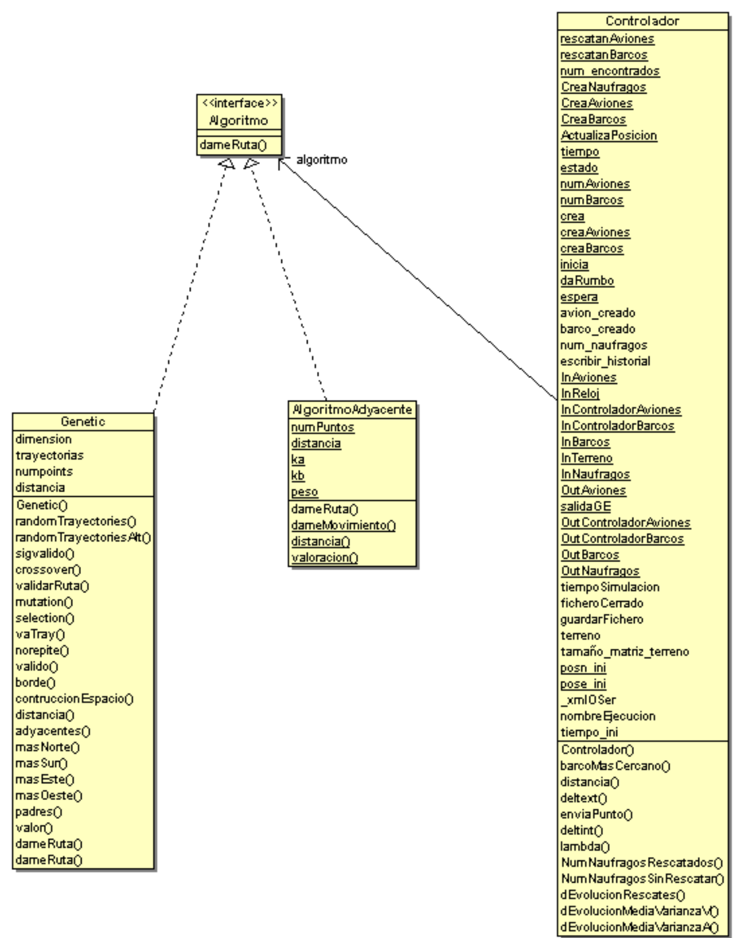
El controlador, además, posee una lista con todos los mensajes que va a enviar, esta lista es del tipo ListaEnviar, que contiene elementos del tipo ParEnviar que contiene el puerto por el que vamos a enviar la información, así como que información vamos a enviar, de este modo, al realizar la transición lambda, simplemente tenemos que ir recorriendo la lista e ir enviando los mensajes.

Tema 8: Algoritmos de búsqueda de náufragos

8.1. Introducción

En esta sección se pasará a explicar como se han implementado un algoritmo genético para la búsqueda de náufragos basándose en unas simulaciones previas.

Para finalizar el desarrollo de este proyecto se decidió crear un algoritmo que generara rutas para aviones y barcos principalmente en el mar, a una altura uniforme. El principal objetivo era conseguir una ruta buena, ya que obtener la optima seria excesivamente costoso y apenas tendría sentido. La elección del tipo de algoritmo de Inteligencia Artificial viene comentada en el apartado de estudio previo, junto con otros aspectos importantes de coste, viabilidad y alcance de esta parte del proyecto.



8.2. Estudio previo

8.2.1. Algoritmo

Para llevar a cabo la generación de rutas se llevo a cabo un estudio de varios tipos de algoritmos de Inteligencia Artificial, para elegir uno adecuado con nuestro espacio de búsqueda (en este caso casillas con náufragos generadas con simulaciones previas que se detallaran mas tarde) y con un tiempo de ejecución y resultado bueno.

Se estudiaron algoritmos tales como “Temple simulado” (Simulated Annealing), búsqueda A*, búsqueda Tabú y algoritmos genéticos.

En una primera criba se desecharon el temple simulado por la complejidad, y la búsqueda A* ya que esta requería de una heurística con unas características que no podía tener nuestro espacio de búsqueda.

Una vez que solo quedaban genético y tabú, y viendo que prácticamente las ventajas y el resultado de usar cualquiera de los dos serian similares, se optó por el genético. Este algoritmo proporciona una buena respuesta sin tener una complejidad excesiva para el sistema.

8.2.2. Viabilidad

Como antes de desarrollar un proyecto o una parte de él como en este caso se debe de hacer un estudio de la viabilidad basándose en cuatro factores:

- Tecnología: En este caso no hay problema con la plataforma eclipse sobre la que trabajamos en java es posible el desarrollo.
- Financiación: Al ser un proyecto de Sistemas Informáticos no requiere de financiación.
- Tiempo: Se puede desarrollar en el tiempo disponible que era un mes aproximadamente, pero hay que ver en el alcance cuantas de las posibles funcionalidades se pueden desarrollar.
- Recursos: Los ordenadores particulares que ya tenemos nos valen para ejecutar la aplicación perfectamente.

8.2.3. Alcance

En este apartado del estudio previo simplemente se valora el tiempo y recursos tanto humanos como materiales. Haciendo unas pequeñas estimaciones tanto por analogía como por puntos de función se vio que seria factible conseguir la ruta para un avión sin cooperar con otros.

8.3. Simulaciones

Para poder llevar a cabo el algoritmo se necesitaba de información de como estarían aproximadamente los náufragos. Esto lo hemos conseguido gracias a unas simulaciones que se realizan antes de llamar al algoritmo.

En este proceso recreamos las condiciones en las que se ha producido el naufragio tales como zona del mar, mareas y vientos,

para así conseguir la mayor similitud posible al caso real que ha ocurrido.

Durante las simulaciones, se guarda información sobre donde han estado los náufragos y esto se guarda cada cierto tiempo en una estructura de ArrayList, para acceder solo a la información que se necesita en un tiempo concreto.

La clase que realiza estas simulaciones aparte de guardarlo en esta estructura, también lo guarda en un .xml por si es necesario acceder desde fuera de la aplicación o se quiere guardar.

8.4. Algoritmo

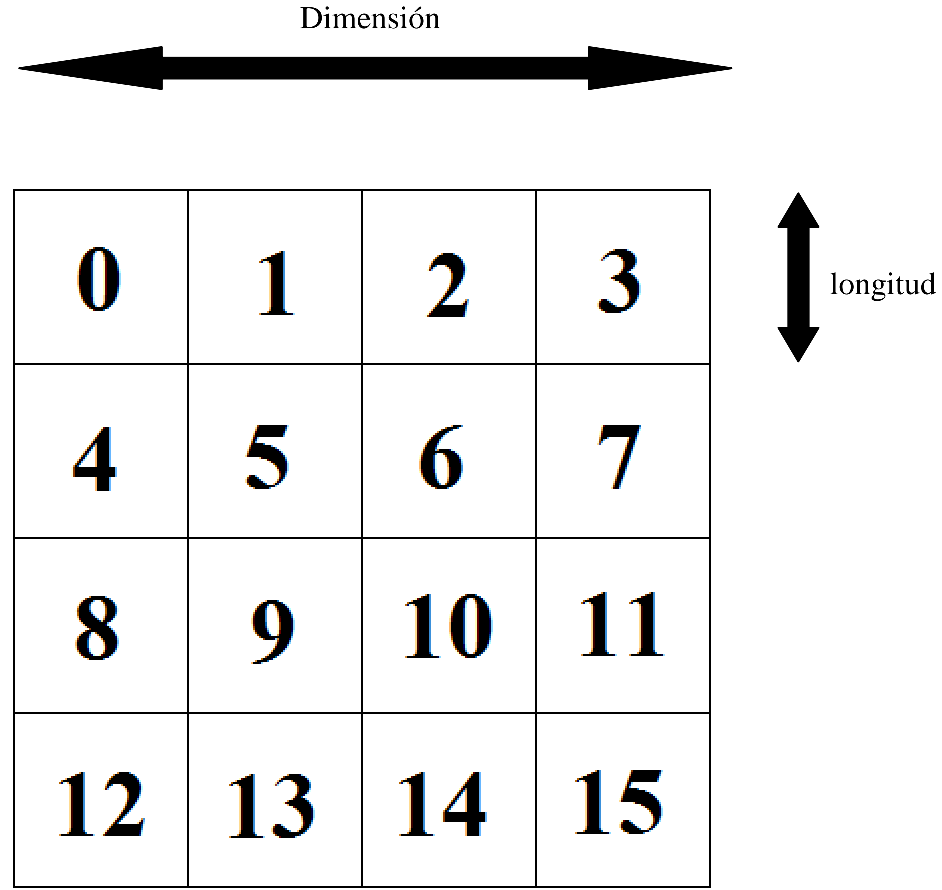
Como se ha comentado previamente, el algoritmo elegido para generar la ruta del avión es un algoritmo genético. Este tipo de algoritmo consta de múltiples fases, las que pasaremos a explicar a continuación.

8.4.1. Espacio de búsqueda

El principal problema al que nos enfrentamos fue como generar el espacio de búsqueda, estaba claro que había que basarse en los datos obtenidos de las simulaciones, pero había que delimitarlo y elegir una representación adecuada para que el algoritmo fuera eficaz.

Se optó por utilizar un cuadrado de casillas, ya que daba muchas facilidades al tener la misma dimensión en ambas direcciones. Para la construcción de este cuadrado había que conseguir la dimensión, y en que puntos colocarlo, para ello se obtenía toda la información de las simulaciones en un tiempo determinado. Esta simulación devuelve las casillas en las que han estado náufragos así que lo que se hizo fue recorrerlas todas para obtener los puntos más al este, oeste, norte y sur. Una vez se tiene esta información se situaba la esquina superior izquierda del cuadrado en estos puntos más este y más norte, y a partir de ahí, se iba sumando en ambas direcciones la longitud que se quería utilizar para cada casilla hasta que cubriera tanto los limites oeste como sur. Con esto obtendríamos la dimensión del cuadrado, que no es más que el número de casillas por borde, al fin y al cabo una matriz.

Un ejemplo de dimensión 4 con su numeración de casillas:



Una vez que se tiene creado el espacio de búsqueda hay que rellenarlo con la información correspondiente, que no es más que situar el número de náufragos que hay en cada casilla, obteniéndolo de la estructura de las simulaciones. Con esto hecho se puede pasar a crear la primera generación.

8.4.2. Estructura de los individuos

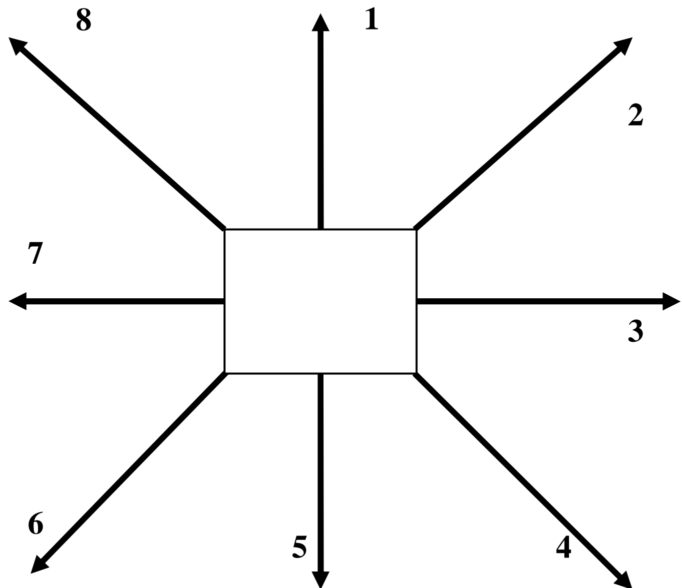
Para representar a cada individuo(que será una ruta) se ha elegido un estructura vector, en la cual la primera posición indica la casilla inicial según la numeración ya establecida en el espacio de búsqueda, y las casillas que le siguen indicaran según un número asociado de 1 a 8 una dirección a tomar. Estas direcciones de son de 1 a 8 ya que el número máximo de casillas adyacentes que puede tener otra casilla es de 8.

La longitud de la ruta y por lo tanto de los individuos vendrá elegida por el usuario.

La siguiente imagen muestra la correspondencia de las direcciones a sus números asociados.

Un ejemplo de individuo sería:

23	1	2	5	3	1	7	3	1	6	4	3



8.4.3. Primera generación

En un algoritmo genético la primera generación debe ser creada aleatoriamente, así pues se ha creado una función que la obtiene.

Lo primero a tener en cuenta es de que número de individuos vamos a querer cada generación. En este caso se ha hecho proporcional a la dimensión del espacio de búsqueda.

A pesar de que supuestamente tienen que ser completamente aleatorios los individuos, para este caso concreto de generación de rutas se han tenido que restringir ciertas cosas.

En primer lugar se ha limitado las casillas de origen, es decir, las primeras de cada ruta. Hay dos casos diferenciables, cuando el avión se encuentra dentro de el propio espacio de búsqueda, entonces las casillas serán solo las adyacentes; y cuando se encuentra fuera, que se cogerán un número determinado de casillas, pero siendo las mas cercanas a donde se encuentra el avión. Esto ultimo es debido a que no tiene sentido que para llegar al punto inicial tengo que atravesar el espacio de búsqueda, ya que no se podría tener en cuenta lo que ya ha visitado y valorar bien la ruta.

A parte de la casilla de origen, al generar las siguientes y por tanto las direcciones a tomar, si se dejaba a completa aleatoriedad (pseudo-aleatoriedad ya que los computadores no pueden hacer algo completamente aleatorio) la generación era muy mala y tardaba mucho ya que generaba gran cantidad de individuos no válidos que se salían del espacio de búsqueda, o bien repetían muchas casilla, algo que no queremos ya que la superficie recorrida es un importante valor de las rutas.

Por esto, se hizo una función que según se iban generando las rutas, daba a elegir la siguiente dirección posible tal que no se saliese del espacio, y si había direcciones que no repitieran casilla primero le dejaba elegir entre estas, para así premiar la máxima superficie recorrida. En caso de que hubiese que repetir porque no existiesen otras, elegía entre estas direcciones.

Una vez hecha esta función se comprobó que no solo las rutas eran muchísimo mejores, sino que su coste en tiempo era tremendamente menor.

Después de tener esta primera generación empieza el proceso genético, que consiste en una repetición de crear nuevas generaciones, lo cual se consigue eligiendo padres, cruzándolos y así obteniendo sus hijos, y matándolos. Todos estos pasos están explicados a continuación.

8.4.4. Elección de los padres

Para crear una nueva generación se tienen que elegir padres (selección) para que se crucen, y para que se vayan mejorando las rutas, lo suyo es ir eligiendo los mejores individuos como padres. Pero no es realmente así, los padres se eligen con el método de la ruleta, que consiste en agruparlos todos según valores de 0 a 360, y se tira un número aleatorio, y en el rango que caiga elegirá un padre u otro. Aquellos individuos que tienen más valor tendrán mas rango y viceversa, consiguiendo así nuestro objetivo.

La función de valoración de las rutas esta hecha de tal manera que tienen mayor valor aquellos que salvan mas náufragos y recorren más espacio, pero de tal manera que premia a los que consiguen náufragos más rápido, es decir puede ser mejor una ruta que salva a 95 náufragos en los primero 10 instantes que la que salva 110 a los 50 instantes.

Con los padres elegidos se pasará a la parte del crossover o cruzamiento.

8.4.5. Crossover

Esta operación esta considerada como aquella que hace diferente a los algoritmos genéticos de otros algoritmos como programación dinámica.

Este es el proceso mediante el cual dos padres generan dos hijos, hay múltiples variantes dentro de los algoritmos genéticos, pero para este problema específico se ha decidido usar el "One-point Crossover" que intercambia las colas de los dos genes padres. Un ejemplo en binario por su simplicidad seria:

- Parent 1 : **1 0 0 0 1 0 0 1 1 1 1**
- Parent 2: 0 1 1 0 1 1 0 0 0 1 1
- Hijo 1: **1 0 0 0 1** 1 0 0 0 1 1
- Hijo 2: 0 1 1 0 1 **0 0 1 1 1 1**

8.4.6. Mutación

En este proceso todos los individuos de la población con chequeados bit a bit y estos valores son cambiados de forma aleatoria de acuerdo a un ratio ya especificado. La operación de mutación fuerza al algoritmo a buscar nuevas áreas. Finalmente ayuda al algoritmo a evitar convergencia prematura y a encontrar una solución global buena.

Un ejemplo de mutación a nivel binario sería:

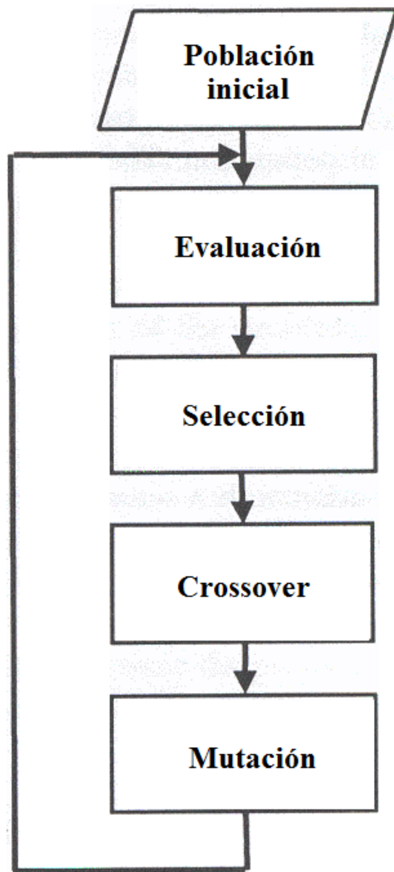
- Viejo: 1 1 0 0 **0** 1 0 1 1 1 0
- Nuevo: 1 1 0 0 **1** 1 0 1 1 1 0

Los procesos de crossover y mutación son realizados muchas veces, de hecho cuantas más veces se repitan, mejores resultados se pueden obtener, nosotros hemos obtenido buenas soluciones ejecutándolo 1000 veces pero es un valor parametrizable según la precisión que se desee obtener.

8.4.7. Resultado final

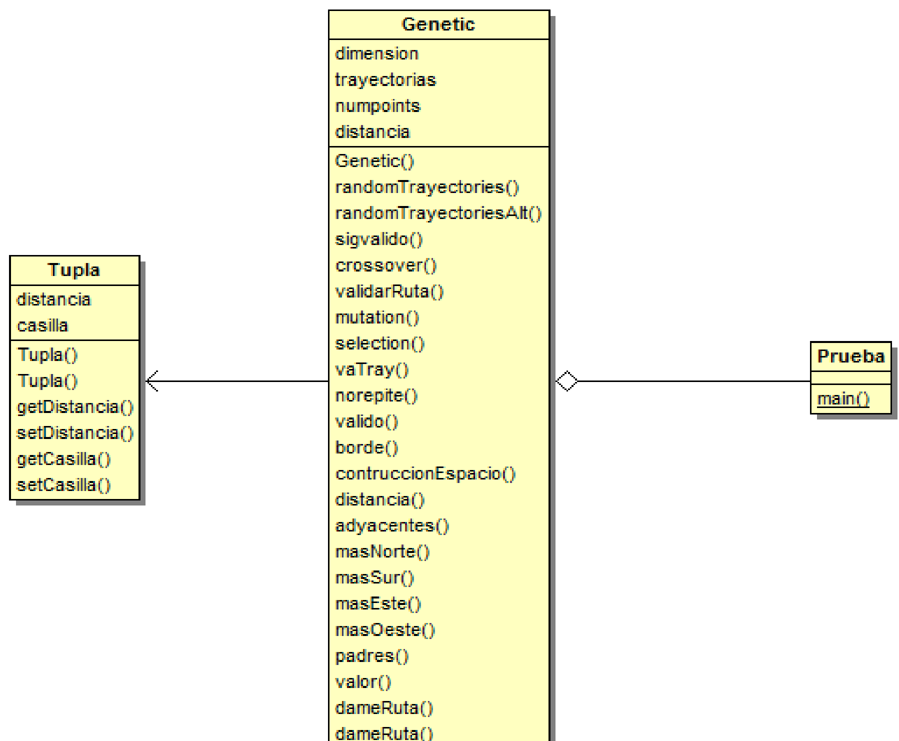
Para finalizar simplemente se coge toda la última generación y se devuelve el individuo de mejor valor. Este individuo aun estará en formato de gen, pero es simple pasarlo a la clase Ruta que será la finalmente devuelta.

8.4.8 Esquema



8.5. Diagrama UML

El diagrama UML del algoritmo genético es el siguiente:



8.6. Pruebas

Para comprobar la corrección del algoritmo, se han realizado varias pruebas manuales, más una pequeña automatización de pruebas.

Todas las pruebas han sido realizadas en el mar, ya que este algoritmo requiere que sea una superficie con una altura uniforme. Se ha utilizado un avión y del orden de 100-1000 náufragos, lo cual nos crea un espacio de búsqueda de una dimensión aproximada de 68 casillas, a 1km de longitud casilla.

Lo habitual en las pruebas sobretodo las que se han realizado cuando el avión podía llegar en un espacio corto de tiempo era que encontrara el orden del 90% de los náufragos ya que se encuentran mas cercanos y acordes a lo que la simulación puede informar. Al pasar el tiempo están más dispersos y la influencia del ruido blanco de las mareas es mayor por lo que la efectividad se reduce.

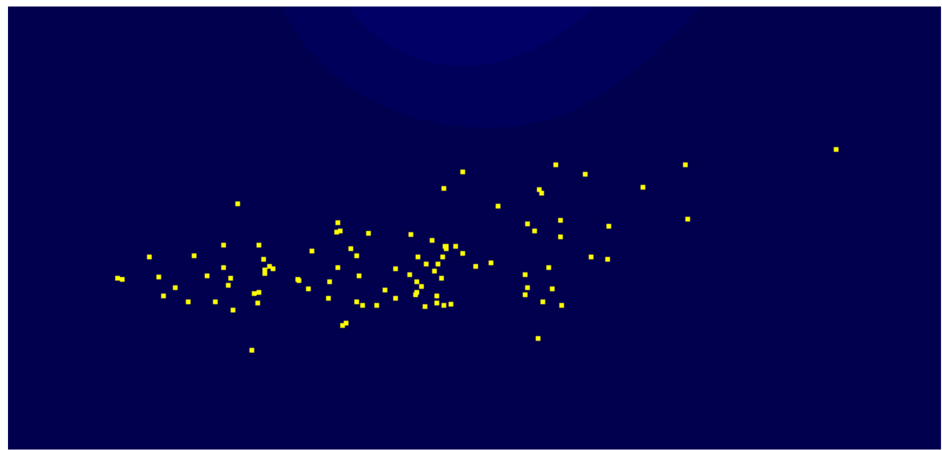
8.6.1. Simulación 1

Parámetros: 10 puntos por ruta (a los 10 puntos vuelve a recalcular), 1 vehículo de cada tipo.

Estado inicial:

A	B	C	D	E	F	G	H
0.0	0.0	0.0	0.0	0.0	0.0	0.00277...	0.00277...
0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.00277...
0.0	0.0	0.0	0.0	0.00554...	0.0	0.0	0.0
0.0	0.0	0.00554...	0.00554...	0.00277...	0.0	0.00138...	0.0
0.0	0.0	0.0	0.00416...	0.0	0.0	0.00832...	0.0
0.0	0.0	0.00416...	0.01109...	0.00554...	0.00970...	0.01248...	0.0
0.0	0.0	0.01525...	0.02496...	0.02357...	0.00416...	0.00832...	0.0
0.0	0.0	0.01386...	0.02635...	0.03190...	0.00416...	0.0	0.0
0.0	0.00554...	0.01386...	0.01664...	0.03190...	0.0	0.0	0.0
0.00554...	0.00554...	0.01525...	0.02357...	0.02357...	0.01109...	0.0	0.0
0.00554...	0.00554...	0.01525...	0.04022...	0.03051...	0.00554...	0.0	0.0
0.0	0.01803...	0.00832...	0.04576...	0.04299...	0.01248...	0.00554...	0.0
0.0	0.01664...	0.04160...	0.03606...	0.01109...	0.0	0.0	0.0
0.0	0.00554...	0.02219...	0.05270...	0.01109...	0.00554...	0.0	0.0
0.0	0.01941...	0.02080...	0.00416...	0.00416...	0.0	0.0	0.0
0.0	0.00416...	0.01386...	0.01386...	0.00832...	0.00416...	0.0	0.0
0.0	0.0	0.0	0.00416...	0.0	0.0	0.0	0.0
0.0	0.0	0.00138...	0.02080...	0.01803...	0.0	0.0	0.0
0.0	0.00277...	0.0	0.00970...	0.0	0.00277...	0.0	0.0
0.0	0.0	0.0	0.00554...	0.0	0.0	0.0	0.0
0.0	0.0	0.00277...	0.0	0.0	0.00277...	0.0	0.0
0.0	0.00138...	0.00416...	0.00277...	0.0	0.0	0.0	0.0

Donde vemos como la mayor probabilidad se encuentra en torno a las columnas D y E en las filas centrales, a partir de ahí van disminuyendo circularmente, esto se correspondería en la simulación real con:



A partir de los valores de la tabla anterior, empieza a trabajar el algoritmo genético, buscando una ruta que sume el mayor porcentaje de probabilidad, puesto que suponemos que a mayor probabilidad para un náufrago, también obtendremos una mayor probabilidad para muchos náufragos, lo que resultará en el salvamento del mayor número de los mismos.

Con el tiempo, la dispersión de los náufragos será mayor, así por ejemplo, pasada media hora de simulación, la distribución será la siguiente:

A	B	C	D	E	F	G	H
0.0	0.0	0.0	0.0	0.0	0.0	0.0	2.68024...
0.0	0.0	0.0	0.0	0.0	0.0	2.68024...	0.0
0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
0.0	0.0	0.0	2.68024...	0.0	0.0	5.36049...	0.0
0.0	0.0	0.00187...	8.04073...	0.0	0.0	0.0	0.0
0.0	0.0	0.00134...	0.00134...	0.0	5.36049...	0.0	0.0
0.0	0.0	0.00241...	0.0	0.0	0.00160...	5.36049...	0.0
0.0	0.00616...	0.00375...	0.00750...	0.00509...	0.00536...	0.0	0.0
0.0	0.00696...	0.01393...	0.03135...	0.00509...	0.00375...	0.00134...	0.0
0.0	0.00643...	0.01527...	0.04341...	0.00321...	0.0	0.0	0.0
0.00294...	0.01259...	0.02948...	0.02975...	0.00857...	0.0	0.0	0.0
0.01447...	0.02948...	0.03430...	0.02867...	0.01500...	0.00750...	0.0	0.0
0.01179...	0.01045...	0.03645...	0.05494...	0.02090...	0.0	0.0	0.0
0.00375...	0.03082...	0.03537...	0.04100...	0.02492...	0.00348...	0.0	0.0
0.01045...	0.01822...	0.08898...	0.02546...	0.00857...	0.0	0.0	0.0
0.0	0.01500...	0.05896...	0.00804...	0.00670...	0.00241...	0.0	0.0
0.00428...	0.02519...	0.00938...	0.00589...	0.00482...	0.0	0.0	0.0
0.00294...	0.00375...	0.01259...	0.00402...	0.00321...	0.0	0.0	0.0
0.0	0.0	0.00268...	0.0	0.0	0.0	0.0	0.0
0.0	0.0	0.00777...	0.00402...	0.00187...	0.0	0.0	0.0
0.0	0.00107...	0.00160...	0.00107...	0.00134...	0.0	0.0	0.0
0.0	0.0	2.68024...	2.68024...	0.0	0.0	0.0	0.0
0.0	5.36049...	0.0	0.0	0.0	5.36049...	0.0	0.0
0.0	2.68024...	2.68024...	0.0	0.0	0.0	0.0	0.0
0.0	0.0	2.68024...	0.0	0.0	0.0	0.0	0.0

Los valores que vemos superiores a uno están multiplicados por 10^{-4}, aunque por claridad (ya que sino no nos entraría la tabla, lo hemos dejado así).

Las probabilidades seguirían evolucionando como se muestra a continuación.

Pasada una hora:

A	B	C	D	E	F	G	H
0.0	0.0	0.0	0.0	0.0	0.0	0.0	7.31796...
0.0	0.0	0.0	0.0	0.0	0.0	7.31796...	0.0
0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
0.0	0.0	0.0	7.31796...	0.0	0.0	7.31796...	0.0
0.0	0.0	0.00256...	0.00146...	0.0	0.0	0.0	0.0
0.0	0.00182...	0.0	0.00219...	0.0	7.31796...	0.0	0.0
0.0	0.0	0.01024...	0.0	0.00292...	0.00219...	0.00146...	0.0
0.0	0.00951...	0.0	0.01061...	0.00439...	0.01244...	0.0	0.0
0.0	0.00695...	0.01975...	0.02341...	0.0	0.00219...	0.00329...	0.0
0.0	0.0	0.01061...	0.03732...	0.0	0.0	0.0	0.0
0.0	0.02414...	0.02744...	0.02671...	0.0	0.0	0.0	0.0
0.02195...	0.04244...	0.03183...	0.02305...	0.01280...	0.00914...	0.0	0.0
0.02341...	0.01500...	0.0	0.04866...	0.02671...	0.0	0.0	0.0
0.0	0.02927...	0.03256...	0.03073...	0.0	0.0	0.0	0.0
0.01536...	0.03915...	0.06439...	0.04061...	0.0	0.0	0.0	0.0
0.0	0.02122...	0.05195...	0.00512...	0.01317...	0.0	0.0	0.0
0.01939...	0.02378...	0.01683...	0.00365...	0.00292...	0.0	0.0	0.0
0.00439...	0.00548...	0.02341...	0.01061...	0.00402...	0.0	0.0	0.0
0.0	0.0	0.00365...	0.0	0.0	0.0	0.0	0.0
0.0	0.00329...	0.01061...	0.00878...	0.0	0.0	0.0	0.0
0.00109...	0.0	0.00292...	0.00256...	0.00109...	0.0	0.0	0.0
0.0	0.0	0.00109...	7.31796...	0.0	0.0	0.0	0.0
0.0	7.31796...	0.0	0.0	0.0	7.31796...	0.0	0.0
0.0	0.00146...	0.0	0.0	0.0	0.0	0.0	0.0
0.0	0.0	7.31796...	0.0	0.0	0.0	0.0	0.0

Pasada hora y media:

A	B	C	D	E	F	G	H
0.0	0.0	0.0	0.0	0.0	0.0	0.0	3.78787...
0.0	0.0	0.0	0.0	0.0	0.0	0.0	3.78787...
0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
0.0	0.0	0.0	0.0	3.78787...	0.0	0.0	7.57575...
0.0	0.0	0.00113...	0.00151...	0.00303...	0.0	0.0	0.0
0.0	0.0	0.00189...	0.0	0.0	0.0	7.57575...	0.0
0.0	0.0	0.0	0.00568...	0.00151...	0.0	0.00454...	0.0
0.0	0.0	0.01060...	0.01136...	0.00530...	0.01401...	0.0	0.0
0.0	0.0	0.00757...	0.03409...	0.00795...	0.0	0.0	0.00151...
0.0	0.0	0.0	0.02310...	0.02196...	0.0	0.0	0.0
0.0	0.0	0.02651...	0.04356...	0.01363...	0.0	0.0	0.0
0.00833...	0.01212...	0.06174...	0.03333...	0.03106...	0.0125	0.0	0.0
0.00909...	0.01287...	0.01628...	0.01742...	0.04848...	0.01287...	0.0	0.0
0.0	0.01287...	0.01553...	0.05037...	0.01515...	0.0	0.0	0.0
0.00833...	0.01136...	0.06287...	0.05151...	0.02007...	0.0	0.0	0.0
0.0	0.0	0.03409...	0.04128...	0.01060...	0.00416...	0.0	0.0
0.0	0.02159...	0.03409...	0.00871...	0.00757...	0.0	0.0	0.0
0.0	0.00492...	0.01780...	0.00568...	0.00530...	0.00416...	0.0	0.0
0.0	0.0	0.00189...	0.00189...	0.0	0.0	0.0	0.0
0.0	0.0	0.00492...	0.00606...	0.00643...	0.0	0.0	0.0
0.0	0.00151...	0.0	0.00454...	0.00113...	0.00113...	0.0	0.0
0.0	0.0	0.0	3.78787...	0.0	0.0	0.0	0.0
0.0	0.0	7.57575...	0.0	0.0	7.57575...	0.0	0.0
0.0	3.78787...	3.78787...	0.0	0.0	0.0	0.0	0.0
0.0	0.0	0.0	3.78787...	0.0	0.0	0.0	0.0

Pasadas dos horas:

A	B	C	D	E	F	G	H	I
0.0	0.0	0.0	0.0	0.0	4.4385...	0.0	0.0	8.8770...
0.0	0.0	0.0	4.4385...	0.0	0.0	0.0	0.0	0.0
0.0	0.0	0.0	8.8770...	0.0	8.8770...	0.0	0.0	0.0
0.0	0.0	0.0	0.0013...	0.0	0.0013...	0.0	0.0	0.0
0.0	0.0	0.0	0.0017...	8.8770...	0.0	0.0	0.0	0.0
0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
0.0	0.0	0.0035...	0.0	0.0075...	0.0066...	0.0	0.0	0.0
0.0	0.0	0.0053...	0.0053...	0.0	0.0	0.0035...	0.0	0.0
0.0	0.0	0.0	0.0399...	0.0186...	0.0031...	0.0	0.0	0.0
0.0	0.0	0.0102...	0.0319...	0.0	0.0	0.0	0.0	0.0
0.0	0.0	0.0261...	0.0133...	0.0	0.0	0.0	0.0	0.0
0.0	0.0137...	0.0	0.0319...	0.0150...	0.0128...	0.0	0.0	0.0
0.0124...	0.0155...	0.0173...	0.0177...	0.0337...	0.0	0.0	0.0	0.0
0.0	0.0332...	0.0186...	0.0572...	0.0177...	0.0	0.0	0.0	0.0
0.0	0.0	0.0559...	0.0474...	0.0173...	0.0	0.0	0.0	0.0
0.0	0.0	0.0173...	0.0705...	0.0488...	0.0	0.0	0.0	0.0
0.0	0.0275...	0.0457...	0.0	0.0142...	0.0	0.0	0.0	0.0
0.0	0.0	0.0124...	0.0124...	0.0057...	0.0044...	0.0	0.0	0.0
0.0	0.0181...	0.0195...	0.0146...	0.0	0.0	0.0	0.0	0.0
0.0	0.0066...	0.0292...	0.0071...	0.0062...	0.0	0.0	0.0	0.0
0.0	0.0	0.0102...	0.0048...	0.0	0.0	0.0	0.0	0.0
0.0	0.0	0.0	0.0	0.0031...	0.0	0.0	0.0	0.0
0.0	0.0	0.0048...	0.0044...	0.0022...	0.0	0.0	0.0	0.0
0.0	0.0	0.0013...	0.0013...	0.0013...	0.0	0.0	0.0	0.0
0.0	0.0	0.0013...	4.4385...	4.4385...	0.0	0.0	0.0	0.0
0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
0.0	0.0	8.8770...	0.0	0.0	0.0	0.0	0.0	0.0

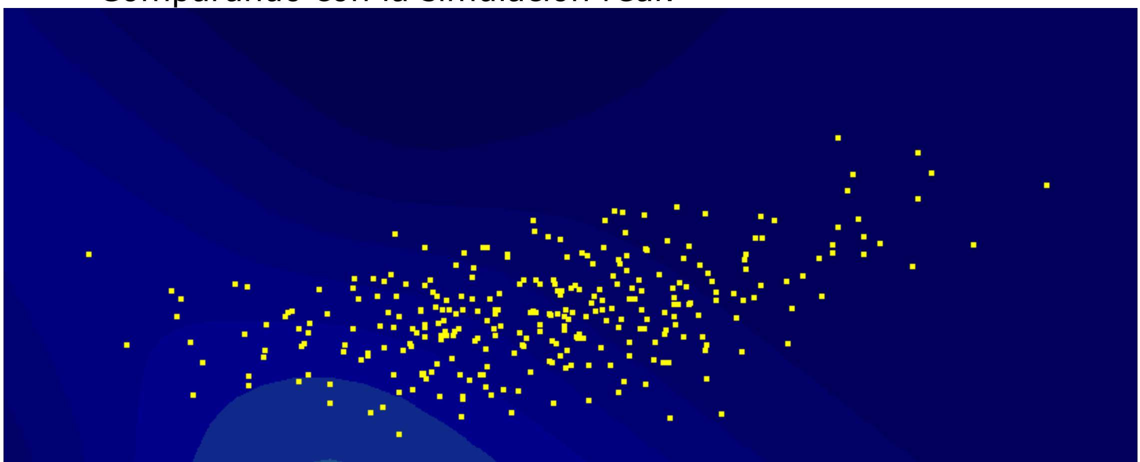
Seguimos observando cada vez una mayor dispersión en el sistema cada vez de una forma más desordenada y no tan concéntrica como podía verse en un inicio.

Pasadas dos horas y media:

A	B	C	D	E	F	G	H	I	J
0.0	0.0	0.0	0.0	0.0	0.0	8.605...	0.0	0.0	0.001...
0.0	0.0	0.0	0.0	0.0	0.0	0.001...	0.0	0.0	0.0
0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
0.0	0.0	0.0	0.004...	0.001...	0.001...	0.0	0.0	0.0	0.0
0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
0.0	0.0	0.0	0.0	0.003...	0.0	0.0	0.0	0.0	0.0
0.0	0.0	0.0	0.004...	0.0	0.0	0.0	0.0	0.0	0.0
0.0	0.0	0.006...	0.0	0.006...	0.005...	0.0	0.0	0.0	0.0
0.0	0.004...	0.004...	0.0	0.003...	0.011...	0.0	0.0	0.0	0.0
0.0	0.0	0.0	0.060...	0.0	0.009...	0.007...	0.0	0.0	0.0
0.0	0.0	0.0	0.047...	0.014...	0.0	0.0	0.0	0.0	0.0
0.0	0.009...	0.030...	0.0	0.018...	0.0	0.0	0.0	0.0	0.0
0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
0.0	0.0	0.029...	0.0	0.0	0.0	0.0	0.0	0.0	0.0
0.055...	0.030...	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
0.0	0.033...	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
0.0	0.0	0.090...	0.017...	0.0	0.0	0.0	0.0	0.0	0.0
0.0	0.0	0.033...	0.0	0.0	0.0	0.0	0.0	0.0	0.0
0.026...	0.0	0.061...	0.0	0.0	0.0	0.0	0.0	0.0	0.0
0.0	0.025...	0.0	0.024...	0.0	0.0	0.0	0.0	0.0	0.0
0.0	0.042...	0.021...	0.020...	0.018...	0.0	0.0	0.0	0.0	0.0
0.0	0.033...	0.034...	0.0	0.0	0.0	0.0	0.0	0.0	0.0
0.024...	0.0	0.006...	0.012...	0.011...	0.0	0.0	0.0	0.0	0.0
0.0	0.0	0.029...	0.009...	0.0	0.0	0.0	0.0	0.0	0.0
0.0	0.0	0.014...	0.006...	0.0	0.0	0.0	0.0	0.0	0.0
0.0	0.0	0.002...	0.006...	0.0	0.0	0.0	0.0	0.0	0.0
0.0	0.0	0.0	0.003...	0.0	0.0	0.0	0.0	0.0	0.0
0.0	0.006...	0.002...	0.0	0.0	0.0	0.0	0.0	0.0	0.0
0.0	0.0	0.001...	0.0	0.0	0.0	0.0	0.0	0.0	0.0
0.0	0.0	0.0	0.0	0.001...	0.0	0.0	0.0	0.0	0.0
0.0	0.0	8.605...	8.605...	0.0	0.0	0.0	0.0	0.0	0.0
0.0	0.001...	0.0	0.001...	0.0	0.0	0.0	0.0	0.0	0.0

Como se ve en la evolución de las gráficas, la posibilidad con ésta marea y vientos de que haya náufragos, es máxima en una zona aproximadamente concéntrica, esto es, que lleva una velocidad media similar, y su rumbo esta fijado entre un rumbo mínimo y un rumbo máximo.

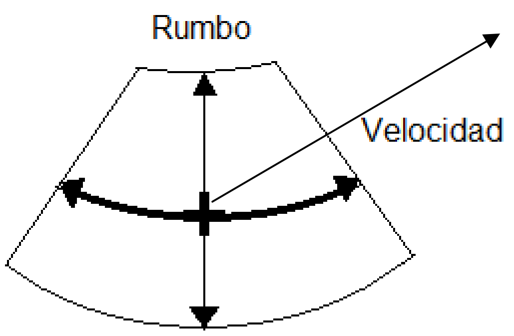
Comparando con la simulación real:



Observamos que realmente el comportamiento es el esperado. Esto unido a que las probabilidades obtenidas mediante simulaciones previas podían resultar interesantes para una primera aproximación pero que debido al comportamiento estocástico del sistema no tenía por que ser preciso del todo, nos llevo a la inclusión de aprendizaje como corrección de éstos parámetros.

Esto nos llevo a pensar que los dos parámetros que debíamos introducir para que el aprendizaje completara las probabilidades encontradas por el método de Montecarlo eran la búsqueda de la velocidad y del rumbo medios de los náufragos, puesto que como podemos ver la distribución está en torno a un punto máximo que en función del tipo de marea y vientos tendrán una velocidad y un rumbo distintos.

Zona de máxima probabilidad



Tema 9: Aprendizaje

9.1. Método de máxima verosimilitud

Consideramos una función de densidad de probabilidad que depende de un conjunto de parámetros $\alpha = (\alpha 1, \alpha 2, \ldots)$. Por ejemplo, si suponemos la función de distribución normal de media m y varianza s^2:

$$\mathrm{p(x)} = \frac{1}{\sqrt{2\pi s^2}} e^{\frac{(x-m)^2}{\sqrt{2\pi s^2}}}$$

El vector de parámetros sería: α = (m , s).

Escribimos la función de densidad en la forma $p(x \mid \alpha)$. Suponemos que tenemos también un conjunto de datos X={x1,x2,…,xn}. Si estos datos se generan de forma independiente por la función de probabilidad, la probabilidad conjunta de obtener ese conjunto de datos es:

$$\mathrm{p(X} \mid \alpha) = \prod p(x_i|\alpha) \equiv \mathrm{L}(\alpha)$$

Donde, para un conjunto de datos X dado, $L(\alpha)$ puede verse como una función de α. A L se le denomina función de verosimilitud (likelihood) para los datos X. Se trata de determinar el vector de parámetros α que hace máxima a la función L. EN la práctica se utiliza el negativo del logaritmo neperiano de L y se determina el mínimo de este valor:

$$\mathrm{E} = \text{- log } \mathrm{L}(\alpha) = \prod p(x_i|\alpha) \equiv L(\alpha)$$

El resultado que se obtiene es el mismo que si maximizamos a L ya que el negativo de del logaritmo es una función monótona decreciente.

Para la función de densidad normal hallamos la solución derivando analíticamente para obtener:

$$\hat{m} = \frac{1}{N} \sum x_i$$

$$\hat{s}_k^2 = \frac{1}{N} \sum (x_i - \hat{m})^2$$

Las ecuaciones suponen que se dispone de todos los datos para realizar la estima, pero si se van obteniendo de uno en uno se pueden obtener expresiones recursivas en la forma:

$$\hat{m_k} = \hat{m}_{k-1} + \frac{1}{k}(x_k - \hat{m}_{k-1})$$

$$\hat{s}_k^2 = \hat{s}_{k-1}^2 + \frac{1}{k}((x_i - \hat{m}_{k-1})^2 - \hat{s}_{k-1}^2)$$

Como valores iniciales

$$\hat{m}_0 = 0,\ \hat{s}_0^2 = 0$$

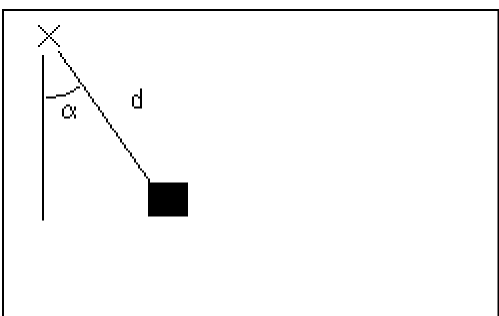
9.2. Implementación del aprendizaje

Como es conocido que las simulaciones previas no pueden dar una predicción exacta de la posición de los náufragos, sino sólo algo mas bien orientativo, se ha desarrollado un modulo de aprendizaje.

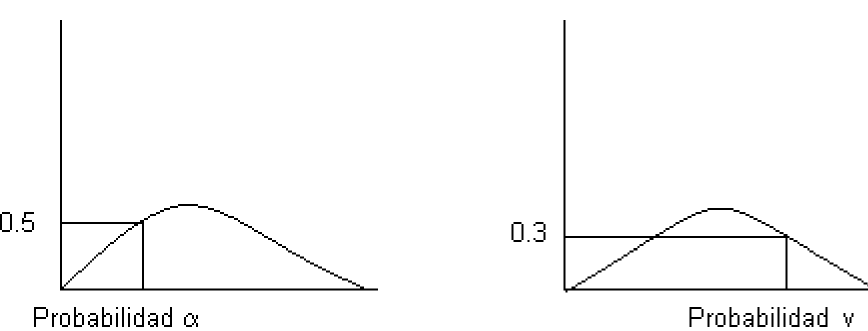
Este modulo, cada vez que un avión encuentra un náufrago, recoge información de su ángulo con respecto al punto inicial y de su velocidad media de desplazamiento desde el momento del naufragio.

Para evaluar un punto se procede del siguiente modo.

Tenemos el punto inicial y un punto a evaluar a una distancia d y a un ángulo α.



Como desde el naufragio ha pasado un tiempo t, la velocidad que lleva el náufrago será d / t. Una vez tenemos la velocidad y el ángulo que llevaría un náufrago en caso de estar en dicha posición, podemos calcular la probabilidad de que realmente haya un náufrago.



La probabilidad total sería el producto de las probabilidades de la velocidad y del ángulo, de éste modo en el caso que vemos sería 0.15.

Para incluir el aprendizaje en el algoritmo genético, simplemente se recalcula el valor de cada casilla con la información proporcionada por el modulo. Esto se hace de la siguiente manera:

*prob_casilla = prob_fichero + (prob_fich*aprendizaje*razon_de_aprendizaje)*

Tras esto, normalizando a uno las probabilidades de las casillas, para lo cuál se calcula la suma total de las probabilidades y se dividen las probabilidades de las casillas entre la probabilidad total.

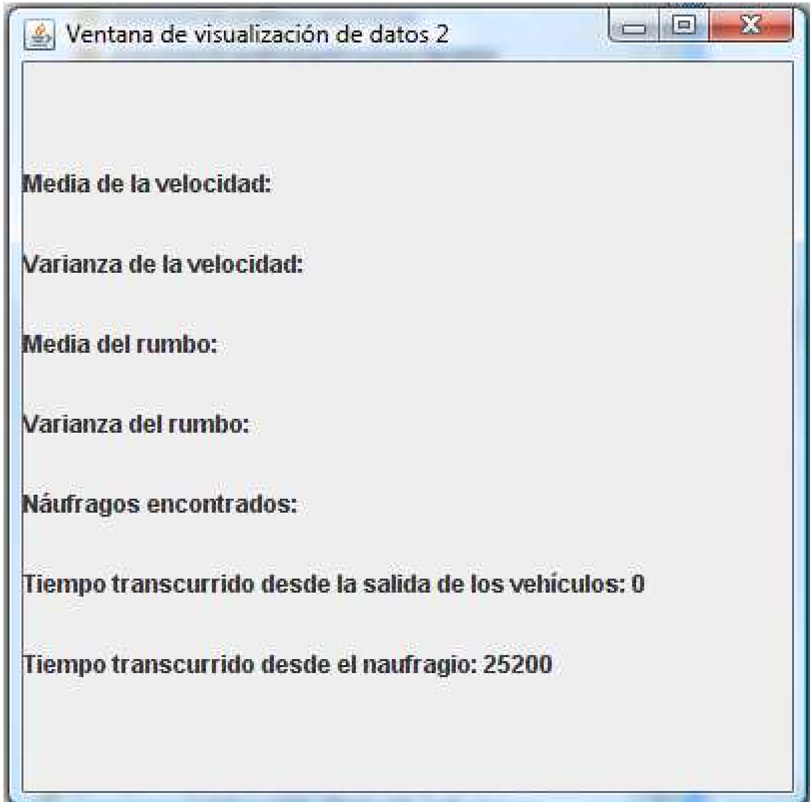
Siendo prob_fichero el valor obtenido mediante la simulación y el aprendizaje la probabilidad estimada a partir del conocimiento real del sistema.

9.3. Ejemplo de aprendizaje

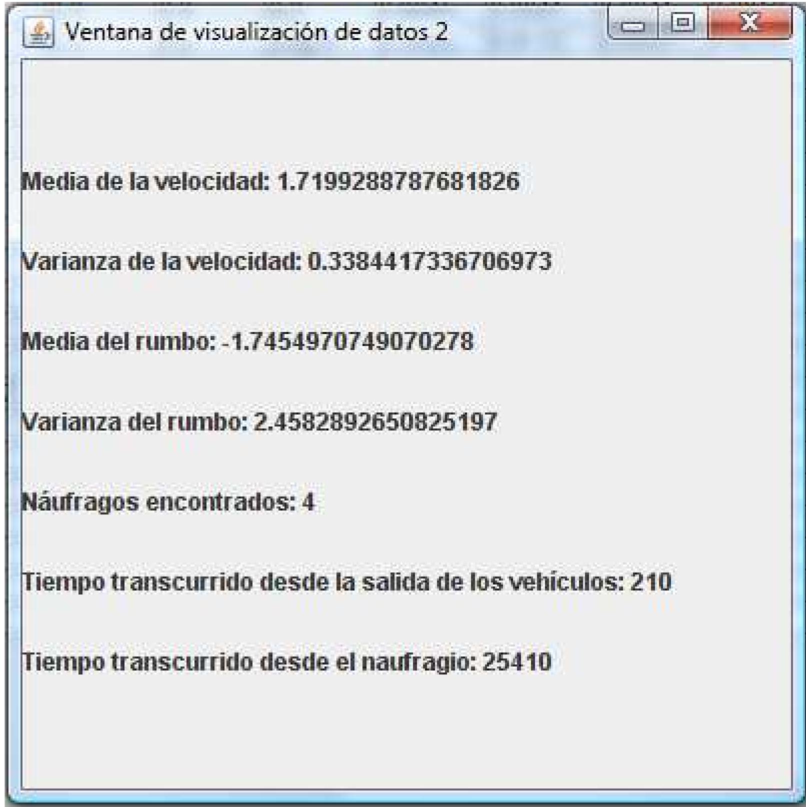
Veamos como van evolucionando los parámetros en una ejecución:

Al principio, hasta que no se visualiza ninguno de los náufragos, el valor de la media y la varianza de la velocidad y el rumbo de los náufragos permanece desconocido, por lo que sólo se visualiza el tiempo transcurrido desde el naufragio y la salida de los vehículos, que es 0.

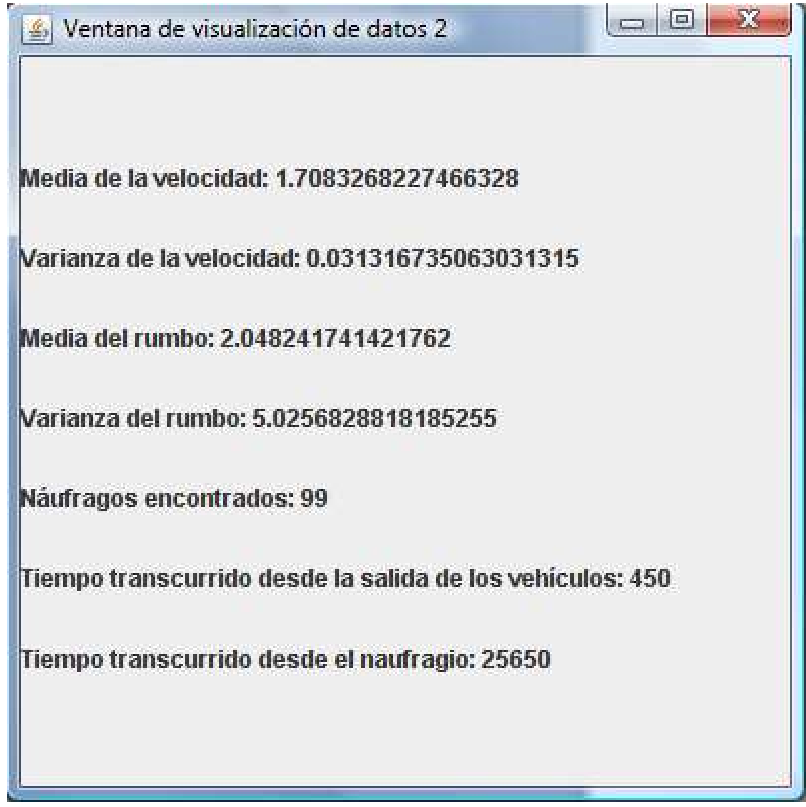
La ventana de parámetros tiene por tanto el siguiente aspecto:



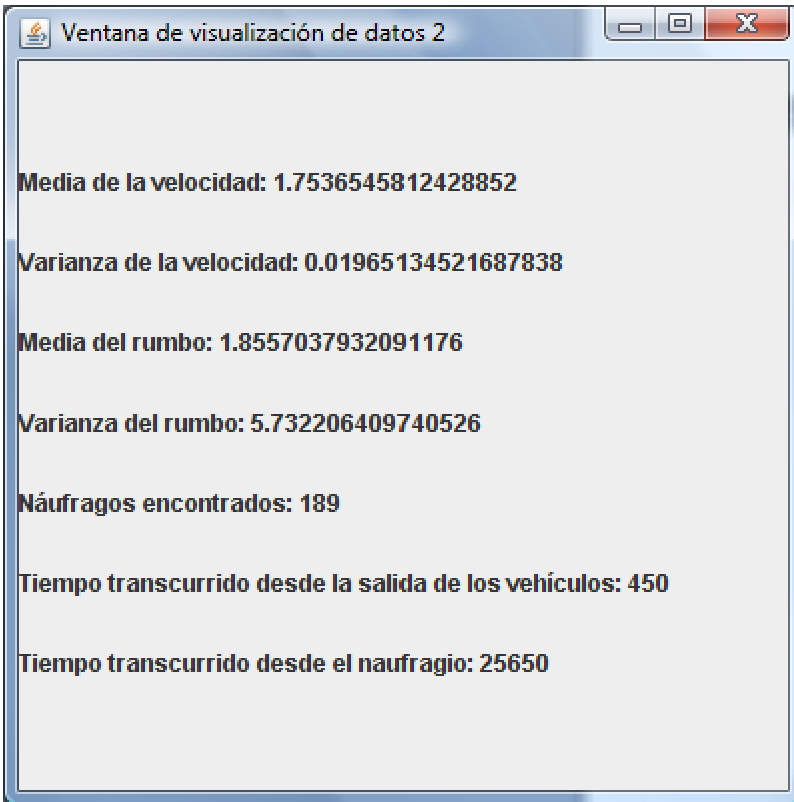
Tras el comienzo de la ejecución, transcurrido un tiempo, los vehículos de la simulación comienzan a encontrar a los náufragos, por lo que se calculan los valores de la media y la varianza de la velocidad y el rumbo correspondientes a esa posición. Un ejemplo de los parámetros transcurrido un tiempo, cuando se han avistado 4 náufragos, es el siguiente:



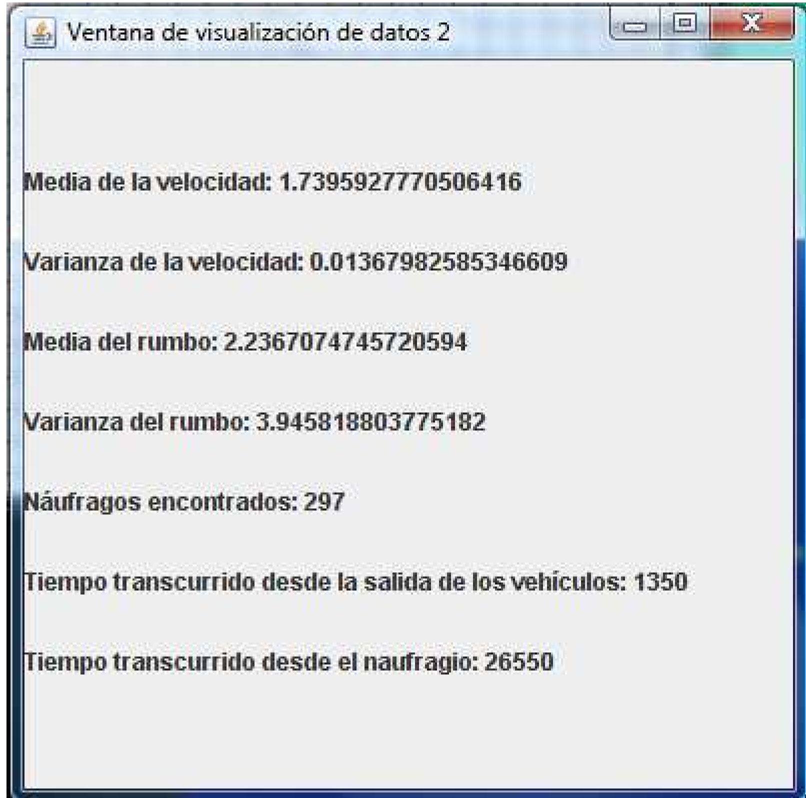
Transcurrido más tiempo, podemos observar como cuando el número de náufragos encontrados aumenta, se modifican los valores de las medias y las varianzas de la velocidad y el rumbo. Como ejemplo, podemos ver la siguiente captura de la simulación:



Podemos observar, como dentro del mismo tiempo, es posible que aumente el número de náufragos encontrados con lo que la media y la varianza calculada se vuelve a modificar. Podemos observar este hecho en la siguiente imagen de la simulación:



Según sigue avanzando el tiempo de la simulación, el número de náufragos encontrados por los vehículos de la misma es mayor, por lo que el algoritmo de aprendizaje puede tener más datos con los que calcular la media y la varianza de la velocidad y el rumbo de una forma más precisa. Por ejemplo, encontrados casi todos los náufragos de la simulación anterior, podemos observar los parámetros aprendidos por el sistema en la siguiente imagen:



Tema 10: Interfaz de simulación

10.1. Introducción

En este tema se presenta el diseño y la construcción de la interfaz de simulación y los entornos de coordinación y planificación de vehículos.

El objetivo de la interfaz de simulación es poder visualizar de forma clara, rápida y sencilla los resultados de las simulaciones, para poder valorar los datos obtenidos de estas. Se ha procurado dotar al simulador de un entorno tanto 2D y 3D creado exclusivamente para el proyecto, así como de un entorno de simulación real, para lo cuál se ha utilizado la herramienta Google Earth.

Para la creación de la interfaz de simulación se ha utilizado la herramienta JOGL (Java OpenGL), que se trata de una librería gráfica para Java.

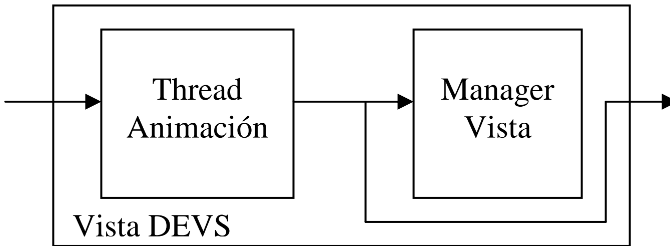
Durante el desarrollo de este tema, se presenta el diseño DEVS realizado para integrar la interfaz de simulación dentro del paradigma de desarrollo DEVS, el diseño de clases y la estructura creada para el desarrollo y funcionamiento de la interfaz y la jerarquía de objetos y de escena creada para la visualización 2D y 3D.

10.2. Diseño XDEVS

La gran dificultad de integrar un diseño XDEVS junto con diseño que utilice la librería gráfica JOGL radica en que debido a la especificación de la librería no se pueden realizar operaciones desde hilos (threads) o flujos de programa distintos.

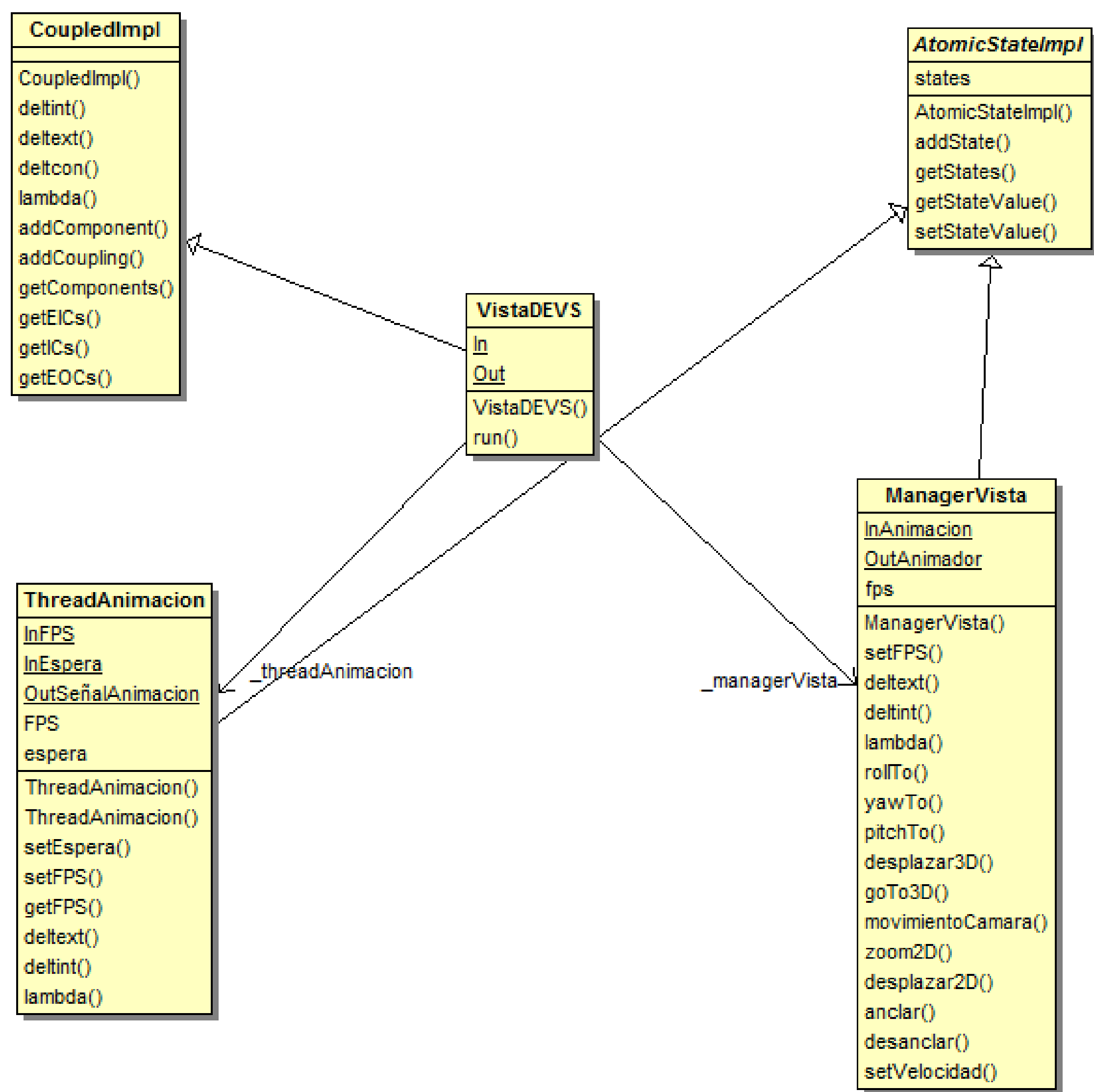
Dado que la implementación XDEVS utiliza hilos distintos para cada uno de los componentes de las simulaciones DEVS, existe una barrera clara a la hora de crear un interfaz de simulación puramente DEVS.

Por ello, se ha optado por la integración de todos los componentes que hacen uso de objetos de la librería gráfica JOGL dentro de un único elemento XDEVS, ManagerVista, y utilizar un esquema DEVS simplificado, que es el siguiente:



1. Diseño DEVS de la interfaz

Se utiliza un modelo compuesto (VistaDEVS) formado por dos modelos atómicos (ThreadAnimacion y ManagerVista). El modelo atómico ThreadAnimación tiene como sigma el tiempo de refresco de la aplicación, y cada vez que se cumple este tiempo se encarga de enviarle un mensaje al modelo ManagerVista, el cuál se encarga de la actualización de las vistas 2D y 3D cada vez que recibe esta señal. El diagrama UML que de la implementación es el siguiente:



2. Diseño UML de la vista DEVS

Podemos observar como las clases ManagerVista y ThreadAnimacion heredan de la clase XDEVS AtomicStateImpl y la clase VistaDEVS hereda a su vez de la clase CoupledImpl. VistaDEVS tiene como atributos un objeto de la clase ThreadAnimacion y otro de la clase ManagerVista, que son los que forman el modelo compuesto.

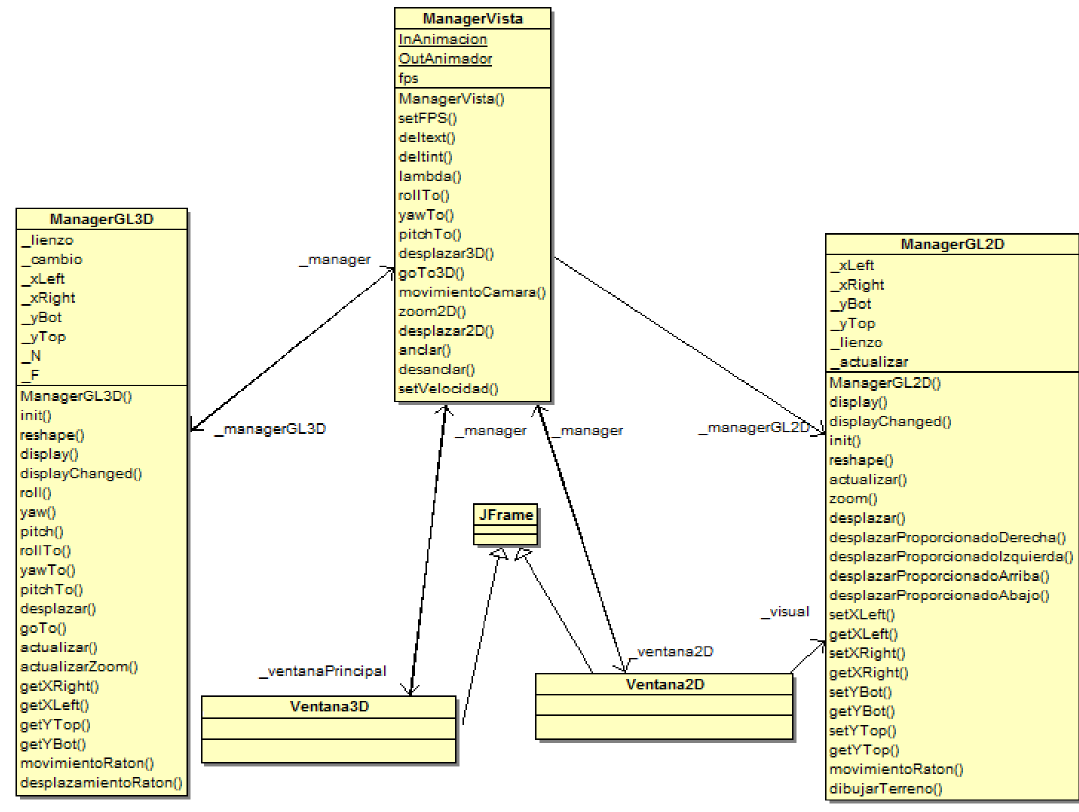
La estructura de la clase ManagerVista y la jerarquía de clases que utiliza se explican con profundidad en el apartado siguiente, puesto que ya no implementan un comportamiento DEVS.

10.3. Diseño y estructuración del manager de la vista

El componente ManagerVista es el cerebro de la interfaz de simulación. Integra todos los elementos que se encargan de la visualización de los resultados obtenidos por la simulación. Desde aquí, se realizan se direccionan todas las operaciones que se realizan en la interfaz gráfica hacia los correspondientes elementos que las implementan.

Como elemento DEVS, tiene como misión actualizar los componentes 2D y 3D cada vez que recibe un mensaje proveniente del ThreadAnimación.

El esquema UML que representa este elemento y las clases que utiliza es el siguiente:



3. Diseño UML del ManagerVista

Como podemos observar en el diagrama UML, el ManagerVista consta de los elementos necesarios para la representación de las vistas 2D y 3D. En concentro, consta de un manager por cada una de las vistas (ManagerGL2D y ManagerGL3D) y una ventana, que hereda de la clase JFrame, donde se dibujarán los elementos de la simulación (Ventana3D y Ventana2D).

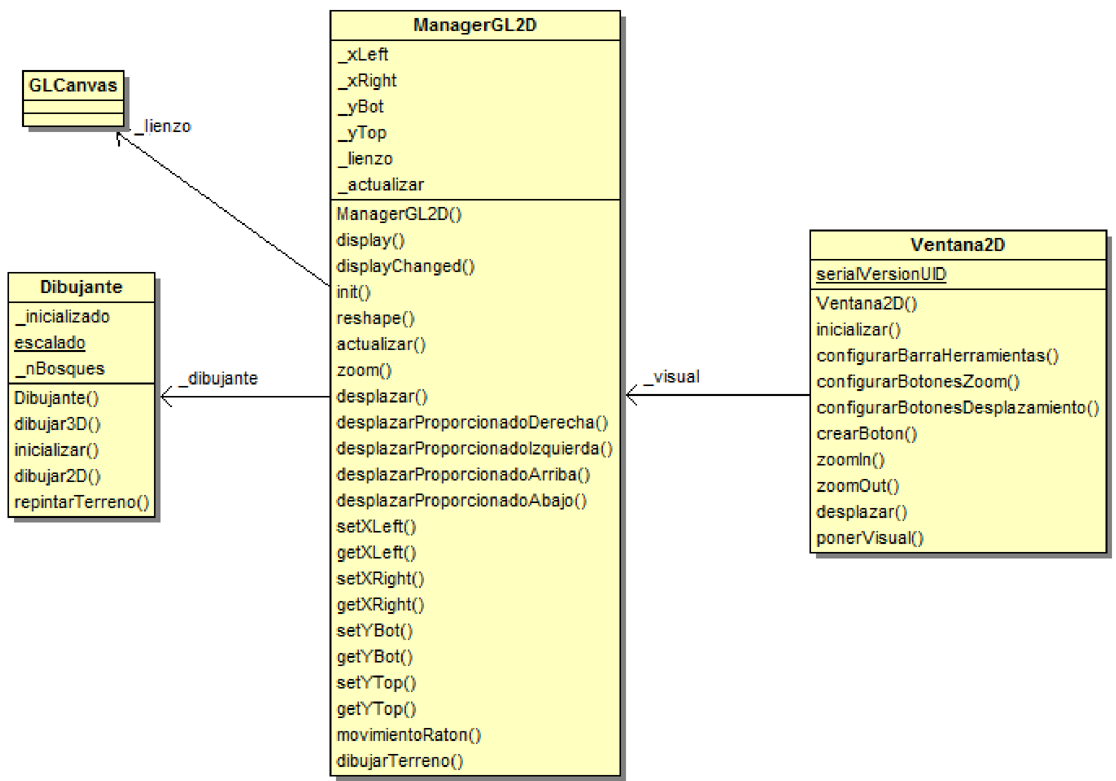
En el momento en que se produce una transición externa, el ManagerVista se encarga de actualizar tanto el ManagerGLD2D y el ManagerGL3D a través de los métodos actualizar(). Gracias a esto, conseguimos que se actualicen tanto la vista 3D como la vista 2D en el momento que se produce una transición interna en el ThreadAnimación, cuyo sigma se establecerá con el número de fotogramas por segundo (FPS) que queramos mostrar en nuestra vista.

El diseño detallado de las interfaces 2D y 3D se explica en los apartados sucesivos.

10.4. Diseño y estructuración de la interfaz 2D

Para la construcción de la interfaz de representación 2D se han utilizado, además de los elementos previamente comentados (ManagerGL2D y Ventana2D), dos elementos más. Un objeto de la clase Dibujable, cuyo objetivo es dibujar todos los elementos que se encuentren dentro de la escena 2D, y un objeto de la clase GLCanvas, que se trata de una superficie de dibujo (canvas) propia de la librería gráfica JOGL, sobre la que se va a dibujar los elementos de la interfaz. Este lienzo se visualiza a través de la Ventana2D.

El diagrama UML de la interfaz 2D es el siguiente:



4. Diseño UML de la interfaz 2D

El objetivo de la interfaz gráfica 2D radica en conocer de forma rápida y precisa la posición de todos los elementos del simulador dentro del entorno de simulación. Para ello, dentro de la interfaz 2D se representa el terreno como un mapa de alturas, donde cada diferencia de altura de 100 metros está representada con un nuevo color.

Para facilitar el uso de la interfaz 2D, se incluyen las siguientes operaciones:

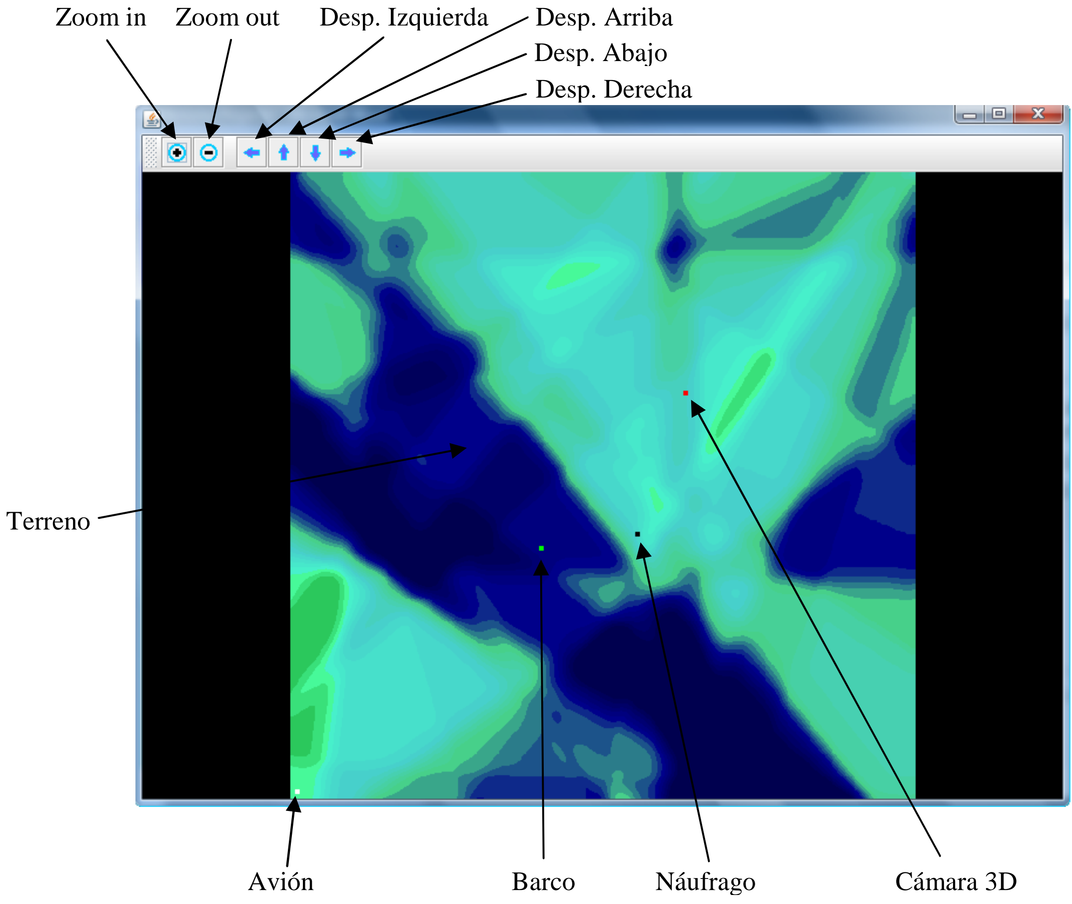
- Operaciones de zoom: Aumentan o disminuyen el área visible del entorno de simulación siempre manteniendo constantes las proporciones del área visible de la ventana. Cuenta con las operaciones:
  - Zoom in: Representada por el icono , disminuye el área visible dentro de la ventana 2D, por lo que aumenta la precisión con la que vemos la superficie representada dentro de esta ventana. Cada vez que se pulsa este botón, se realiza un zoom con un factor x0.5 (aumentado).
  - Zoom in: Representada por el icono , aumenta el área visible dentro de la ventana 2D, por lo que disminuye la precisión con la que vemos la superficie representada. Cada vez que se pulsa este botón se realiza un zoom con un factor x2 (disminución).
- Operaciones de desplazamiento: Nos permiten desplazarnos por el área visible de la simulación permitiendo visualizar partes que quedaban ocultas en un momento dado. Consta de las siguientes operaciones:
  - Desplazamiento a la izquierda: Representado por el icono , desplaza el área visible a la izquierda.
  - Desplazamiento hacia arriba: Representado por el icono , desplaza el área visible hacia arriba.
  - Desplazamiento hacia abajo: Representado por el icono , desplaza el área visible hacia abajo.
  - Desplazamiento a la derecha: Representado por el icono , desplaza el área visible a la derecha.

Todas estas operaciones, junto con el redimensionamiento de la ventana 2D mantienen la proporción del área visible, por lo que esta se redimensiona para ajustarse a las dimensiones de la ventana.

Los objetos que se visualizan en la ventana 2D son los siguientes:

- Aviones: Se representan como un punto blanco.
- Barcos: Se representan como un punto verde.
- Náufragos: Se representan como un punto negro.
- Cámara: La posición de la cámara en la vista 3D se representa como un punto rojo en la interfaz 2D.

Un ejemplo de la visualización de una simulación en la interfaz 2D es el siguiente:

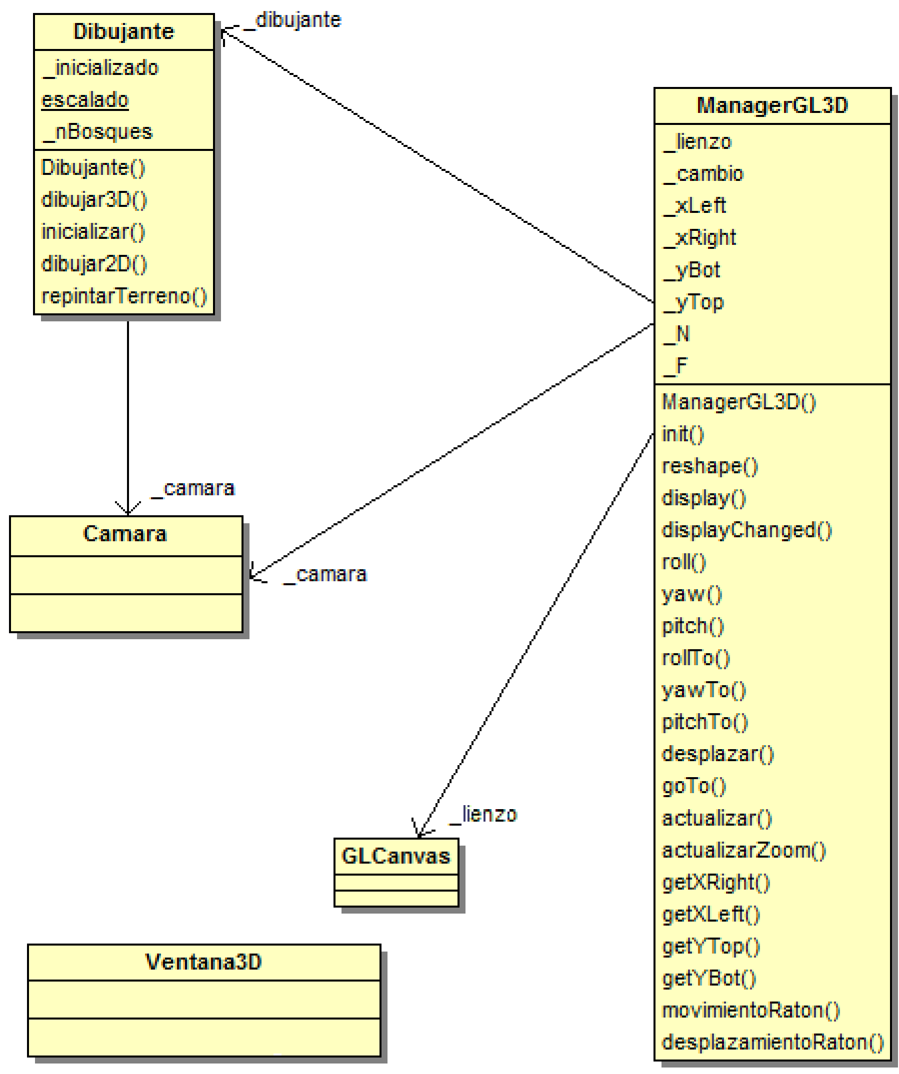


5. Ejemplo de visualización en la interfaz 2D

Además de utilizar los botones propios de las operaciones de desplazamiento, se pueden realizar estos desplazamientos sin más que hacer clic izquierdo con el ratón sobre el terreno de la interfaz 2D y desplazar sin soltar el botón. Con esto, desplazamos la imagen visible manteniendo el punto sobre el que hemos hecho clic.

10.5. Diseño y estructuración de la interfaz 3D

Al igual que en la interfaz 2D, utilizamos los componentes de la clase Dibujante (que se encargará de dibujar los objetos de la simulación) y un lienzo de la clase GLCanvas, que será el visualizado en la Ventana3D. Además se utiliza un objeto de la clase Camara, que representa a la cámara de la vista en 3D y sobre la que se pueden realizar todas las operaciones propias de una cámara.



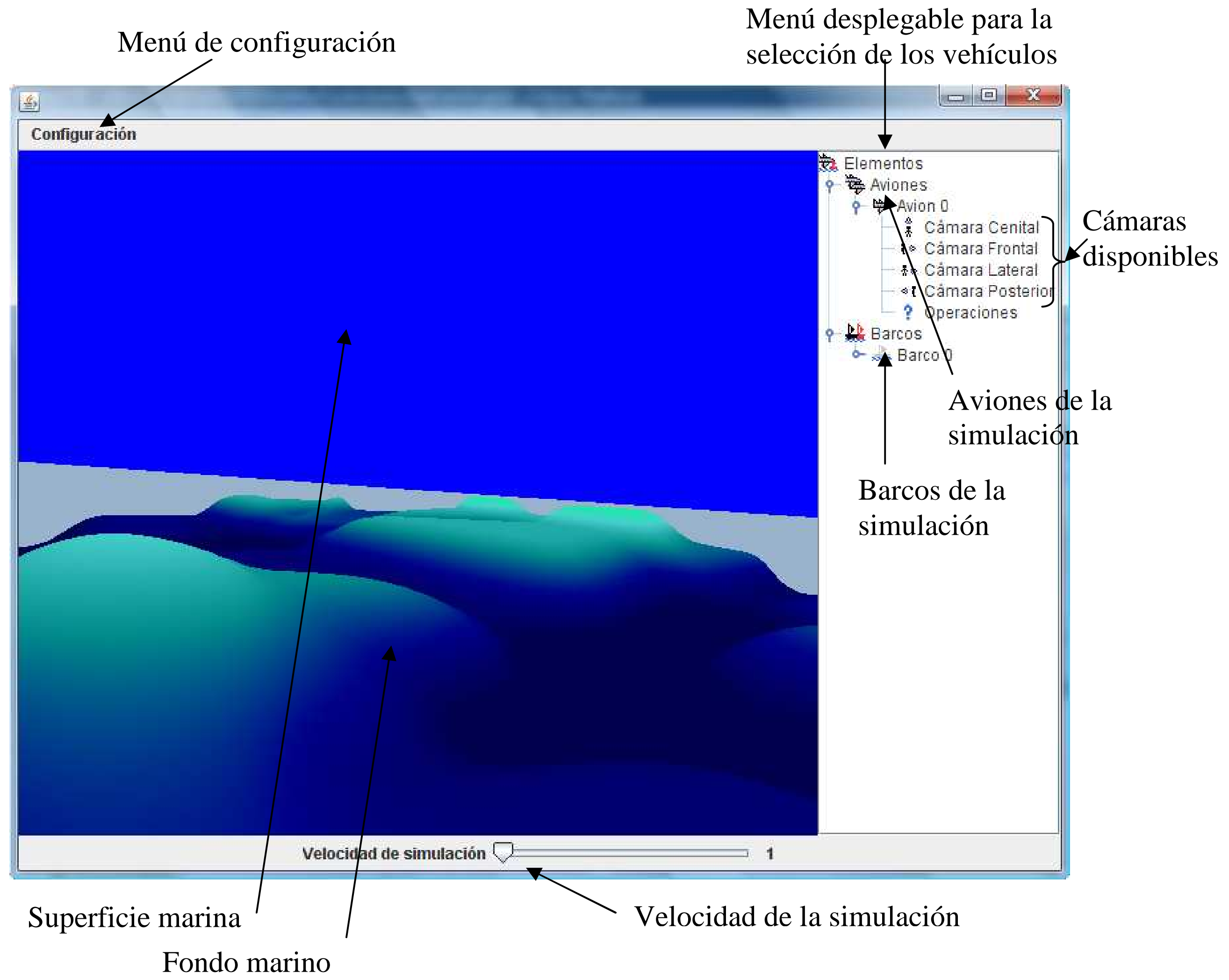
6. Diseño UML de la interfaz 3D

El funcionamiento de la interfaz 3D es similar a la de interfaz 2D. Cuando se produce una actualización, el ManagerGL3D se encarga de repintar la escena 3D, gracias al Dibujante, el cuál dibuja los objetos sobre la escena. Las operaciones sobre el punto de vista de la representación 3D son implementadas por la clase Cámara.

Las operaciones que pueden realizarse sobre la interfaz 3D son las siguientes:

- Navegación libre por el entorno de simulación: Se realiza desplazándose por la escena gracias al ratón y a los cursores del teclado. Basta con hacer clic izquierdo sobre la superficie de la simulación y arrastrar sin soltar el botón del ratón para orientar la cámara en la posición deseada. Para desplazar la cámara en las direcciones sobre la orientación de esta, utilizamos los cursores del teclado.
- Navegación anclada a un vehículo de la simulación: Para acompañar a unos de los vehículos de la simulación, seleccionamos una de las cámaras disponibles para cada uno de los vehículos existentes en la simulación. Esto lo hacemos a través del menú desplegable situado en la parte derecha de la interfaz 3D. Para elegir una de las cámaras, hacemos doble clic sobre ella. Para liberar a la cámara del movimiento anclado a uno de los vehículos, basta con hacer click sobre un elemento del árbol que no sea una cámara.
- Modificación de la velocidad de simulación: Gracias al slider situado en la parte inferior de la interfaz, se puede modificar la velocidad de la simulación. Esta se puede establecer a velocidad real (1) o hasta cien veces más rápido (100).
- Modificar los parámetros de la configuración de la interfaz de simulación: Lo conseguimos a través del elemento de menú Configuración de la interfaz 3D.

Un ejemplo de la visualización de una simulación en la interfaz 3D es el siguiente:



7. Ejemplo de visualización en la interfaz 3D

La imagen que podemos observar es resultado de una simulación en un entorno marino con un avión y un barco. La cámara se encuentra situada entre la superficie marina y el fondo, por lo cuál podemos observar las diferentes irregularidades del fondo oceánico, las cuales quedan representadas en la vista 2D a través de un mapa de alturas.

10.5.1. Diseño de la cámara 3D

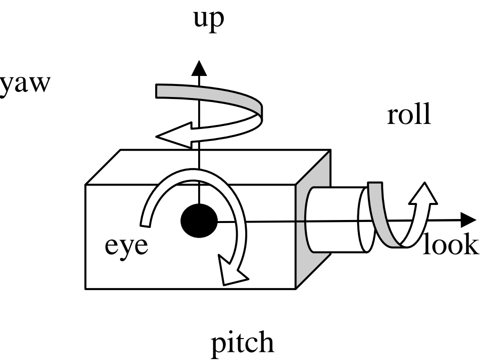
La cámara 3D cumple las funciones del ojo del observador dentro de la interfaz de simulación 3D. Esta se encuentra colocada en el punto desde el cuál observamos la escena, con su inclinación y rotación. Se encarga por tanto de realizar todas las operaciones necesarias sobre el punto de vista.

Para conseguirlo, la cámara realiza operaciones sobre las matrices de proyección de la tubería gráfica de JOGL, en concreto, sobre la matriz de modelado y vista.

La cámara está definida por los siguientes atributos:

- eye: Punto que indica la posición de la cámara
- look: Vector que apunta en la dirección hacia la que está enfocada la cámara
- up: Vector perpendicular a la dirección de look y que apunta hacia la parte superior de la cámara.
- Ángulo roll: Ángulo que gira la cámara tomando como eje el vector look.
- Ángulo yaw: Ángulo que gira la cámara tomando como eje el vector up.
- Ángulo pitch: Ángulo que gira la cámara tomando como eje un vector perpendicular a up y look.

Con esto, la posición y orientación de la cámara queda completamente definida.



8. Ejes y ángulos en la cámara 3D

Las operaciones permitidas por la cámara para modificar el punto de vista son las siguientes:

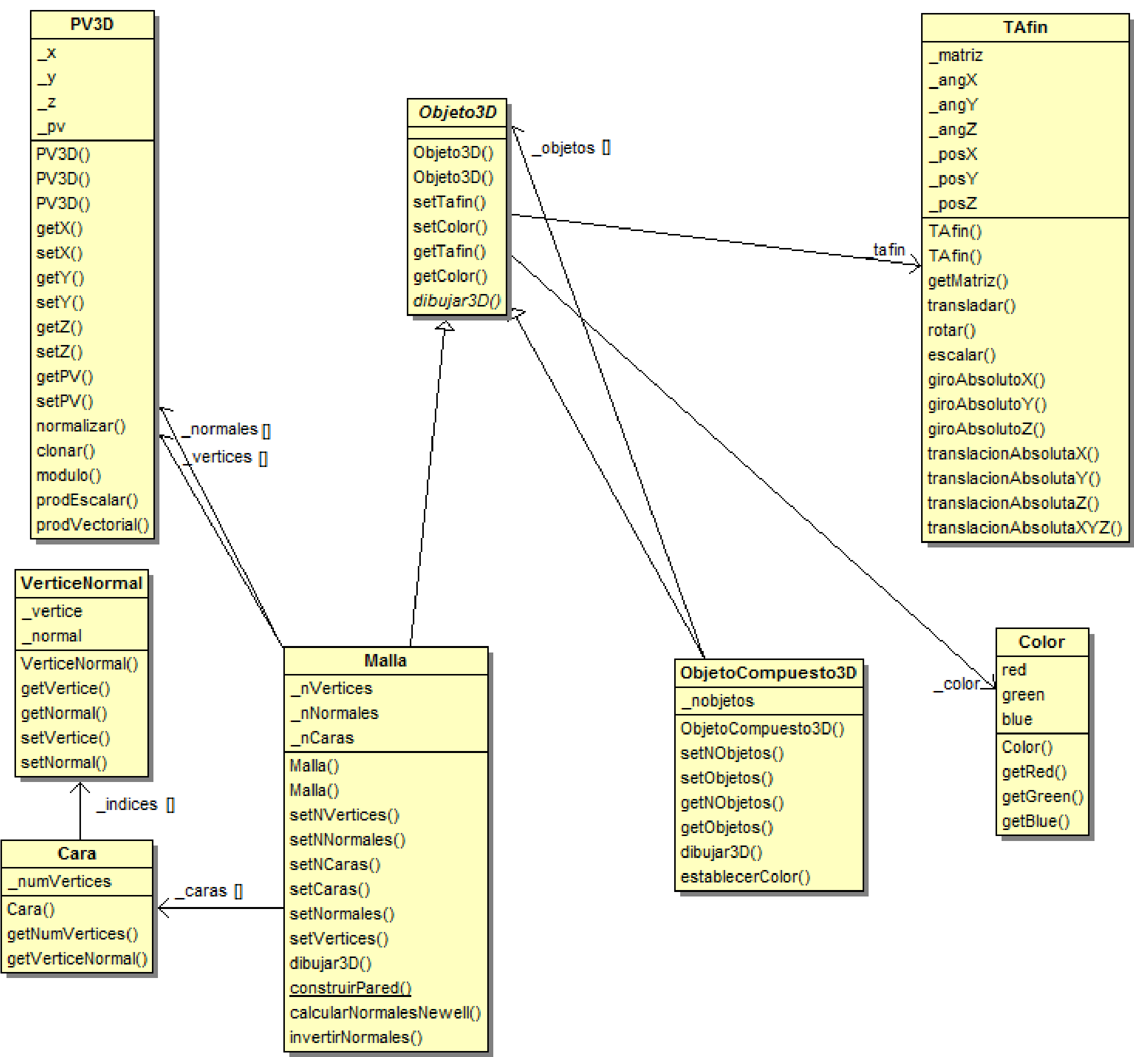
- roll(ángulo): Hace girar la cámara en el ángulo roll con un ángulo dado.
- Yaw(ángulo): Hace girar la cámara en el ángulo yaw con un ángulo dado.
- pitch(ángulo): Hace girar la cámara en el ángulo pitch con un ángulo dado.
- rollTo(ángulo): Posiciona la cámara con un ángulo de roll dado.
- yawTo(ángulo): Posiciona la cámara con un ángulo de yaw dado.
- pitchTo(ángulo): Posiciona la cámara con un ángulo de pitch dado.
- desplazar(x,y,z): Desplaza la posición de la cámara sumando las coordenadas x, y y z a la posición del eye de la cámara.
- goTo(x,y,z): Posiciona el eye de la cámara en la posición x, y y z.

Aunque durante las simulaciones sólo de utiliza un tipo de proyección para la visualización 3D, la cámara puede fijar tres tipos diferentes de visualización:

- Perspectiva: Es la utilizada para la visualización de la simulación. Está definida por la distancia de dos planos, near y far, que delimitan el área visible, el ángulo de apertura de la perspectiva y un valor de proporción.
- Oblicua: Está definida a través de cuatro puntos que delimitan un cubo de visualización, los planos near y far, y un punto para trazar los vectores que representan al los objetos.
- Ortogonal: Está definida por cuatro puntos que delimitan un cubo de visión y los planos near y far, sobre el que se proyectan los objetos.

10.6. Jerarquía de objetos 3D

Para la creación de los elementos 3D de la simulación (excepto el terreno) se ha creado una jerarquía de elementos que permite la construcción de elementos de una forma sencilla. La jerarquía construida se representa en el siguiente diagrama UML:



9. Diseño UML de la estructura de objetos 3D

A continuación de explican con detalle cada una de las clases que forman esta jerarquía.

10.6.1. TAfin

Se trata de una matriz de 16 componentes que sirve para representar las rotaciones, translaciones y giros que se pueden realizar sobre un objeto 3D. Cada objeto 3D va acompañado de su matriz 3D que lo posiciona de manera absoluta dentro de la escena.

Consta de los siguientes métodos:

- trasladar(): Realiza una traslación relativa del objeto desde la posición actual.
- rotar(): Realiza una rotación relativa a la posición actual del objeto según el vector y el ángulo pasados como parámetro.
- escalar(): Realiza una escalación del objeto según el vector dado como parámetro.
- giroAbsolutoX(): Realiza un giro absoluto del objeto según el eje X posicionando al objeto con el ángulo dado.
- giroAbsolutoY(): Realiza un giro absoluto del objeto según el eje Y posicionando al objeto con el ángulo dado.
- giroAbsolutoZ(): Realiza un giro absoluto del objeto según el eje Z posicionando al objeto con el ángulo dado.
- traslacionAbsolutaX(): Realiza una traslación absoluta del objeto en el eje X posicionándolo en la componente dada.
- traslacionAbsolutaY(): Realiza una traslación absoluta del objeto en el eje Y posicionándolo en la componente dada.
- traslacionAbsolutaZ(): Realiza una traslación absoluta del objeto en el eje Z posicionándolo en la componente dada.
- traslacionAbsolutaXYZ(): Realiza una traslación del objeto a la posición dada.

En el momento en el que se dibuje el objeto, se multiplicará la matriz de transformación afín del objeto por la matriz de modelado y vista de la tubería gráfica. Con esto, el objeto se dibujará en la posición deseada.

10.6.2. Color

Representa el color de un objeto en formato RGB. Tiene tres componentes que representan los valores r, g y b respectivamente del color al que representan.

10.6.3. Objeto3D

Es la superclase de todo objeto 3D de la simulación. Todo objeto que quiera ser representado en la interfaz 3D debe heredar de la clase 3D. Tiene como atributos:

- _tafin: Matriz de transformación afín que posiciona al objeto dentro de la escena. Esta matriz representará las operaciones de translación rotación y escalado que se realicen sobre el objeto 3D para situarlo dentro de la escena.
- _color: Objeto de la clase Color que como su nombre indica, representa el color de objeto.

La clase Objeto3D es una clase abstracta, puesto que no define el comportamiento de uno de sus métodos, el método dibujar3D(). Este método es un método común a todos los objetos 3D y será este método el encargado de dibujar propiamente al objeto a través de las funciones proporcionadas por la librería gráfica JOGL. Como veremos más adelante, cada una de las clase de heredan de la clase Objeto3D sobrescribe este método, y lo hace de una manera diferente, por eso este método es un método abstracto.

10.6.4. PV3D

Esta clase se utiliza para la representación tanto de puntos como de vectores en 3D. Tiene los atributos:

- _x: Coordenada x del elemento.
- _y: Coordenada y del elemento.
- _z: Coordenada z del elemento.
- _pv: Componente que nos indica si el elemento se trata de un punto o de un vector: 1 si se trata de un punto y 0 si se trata de un vector.

Gracias a la utilización de las coordenadas homogéneas y al uso de la componente _pv, podemos tratar de igual manera tanto a los puntos como a los vectores, por eso se representan con la misma clase.

Además de los modificadores y accesotes propios de los atributos de los que consta la clase PV3D, se utilizan las siguientes operaciones implementadas por la clase:

- normalizar(): Normaliza las coordenadas del elemento.
- clonar(): Devuelve un nuevo elemento igual al actual.
- modulo(): Devuelve el módulo del elemento.
- prodEscalar(): Devuelve el resultado del producto escalar entre el elemento actual y otro pasado como parámetro.
- prodVectorial(): Devuelve el resultado del producto vectorial entre el elemento actual y otro pasado como parámetro.

10.6.5. VerticeNormal

Se trata de una clase utilizada por la clase Cara y sirve para relacionar un vértice con su correspondiente normal. Posee dos atributos, los cuales indican los índices de un vértice y de la normal correspondiente.

10.6.6. Cara

Las caras se utilizan para la representación de superficies planas dentro de las mallas. Cada cara se representa como un conjunto de elementos VerticeNormal, que relacionan cada uno de los vértices que forman la cara con la normal de esta.

10.6.7. Malla

Se trata de una de las dos clases que heredan de la clase Objeto3D y sobrescriben el método dibujar3D(). Representa los objetos tridimensionales como un conjunto de vértices, caras y normales. Consta de los siguientes elementos:

- _vertices: Conjunto de vértices que constituyen la maya. Se trata de puntos es el espacio 3D representados por objetos de la clase PV3D.
- _caras: Conjunto de elementos de la clase Cara. Se trata de las caras de objeto formadas entre el conjunto de puntos y que constituirán los lados del objeto.
- _normales: Se trate de un conjunto de vectores perpendiculares al conjunto de caras.

Puesto que la clase Malla hereda de la clase Objeto3D y podemos crear objetos de la clase Malla, es necesario sobrescribir el método dibujar3D(). Para dibujar en tres dimensiones una malla, en primer lugar se procede a la multiplicación de la matriz de transformación afín del objeto por la matriz de modelado y vista de la tubería gráfica. Tras esto, se procede al recorrido de las caras de la malla, sobre cada una de las cuales se creará un polígono formado por cada uno de los puntos que forman la malla y la normal correspondiente a esa cara.

10.6.8. ObjetoCompuesto3D

Se trata de la clase restante que hereda de Objeto3D y que por lo tanto como la anterior sobrescribe el método dibujar3D(). Consta de una colección de objetos 3D por los que está formado.

En este caso, para dibujar el objeto compuesto, en primer lugar se aplica la matriz de transformación afín y después se recorre la lista de objetos 3D por los que está formado el objeto compuesto. Por lo tanto, la matriz de transformación afín del objeto compuesto afecta a todos los objetos 3D por los que está formado. De esta forma, cuando se pinta alguno de estos objetos, su matriz de transformación afín se multiplica por la matriz de modelado y vista sobre la que ya se ha multiplicado la matriz de transformación afín del objeto compuesto 3D. De esta forma, cada uno de los objetos que forma el objeto compuesto se posiciona con su matriz de transformación respecto de la posición establecida por la matriz de transformación afín del objeto compuesto.

10.7. Visualización de los datos de las simulaciones

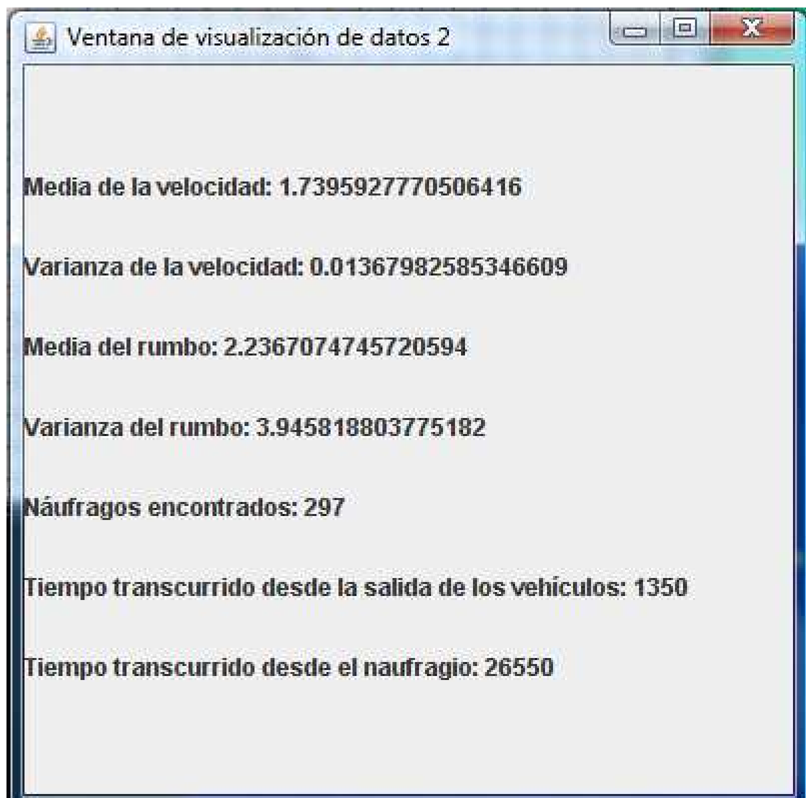
Para la visualización de los datos de las simulaciones se utilizan dos nuevas ventanas. Son las siguientes:

10.7.1. Ventana de visualización de probabilidades

Esta ventana tiene por objetivo poder visualizar los valores de las probabilidades que el algoritmo genético asigna a cada una de las casillas que son contempladas. En ella, cada una de las casillas mostradas de la tabla corresponde a una casilla utilizada por el algoritmo genético. El valor asociado a cada casilla se muestra en la casilla correspondiente de la tabla. El aspecto de esta ventana es el siguiente:

10.7.2. Ventana de visualización de parámetros

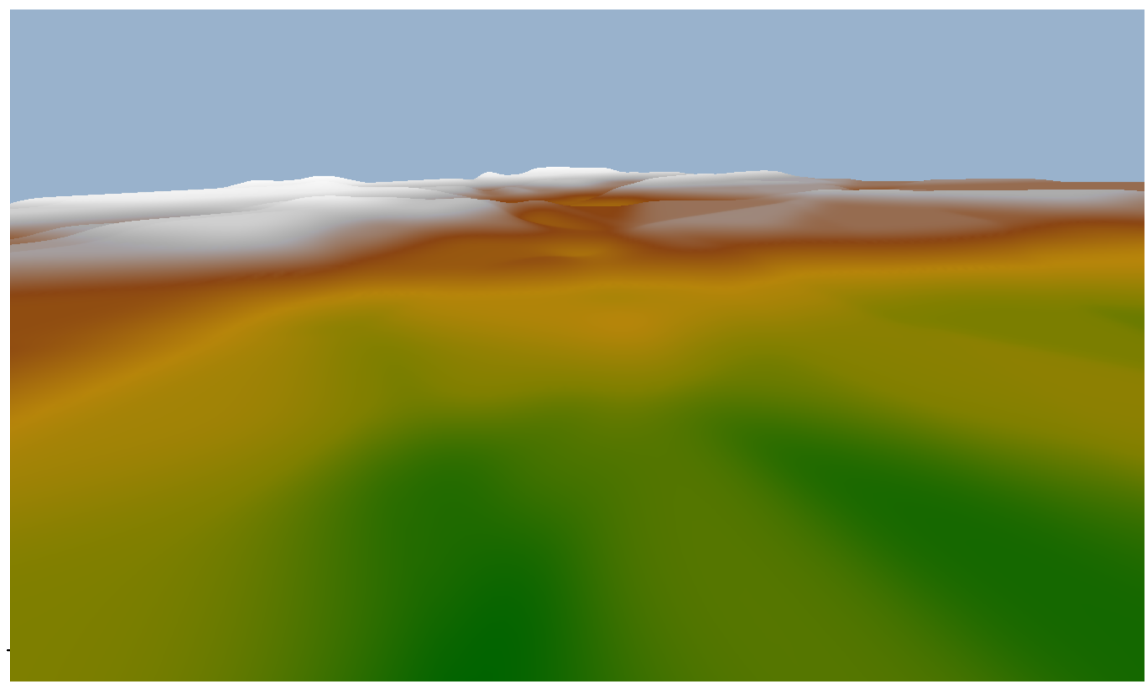
En esta se visualizan los restantes parámetros de la simulación que no han sido visualizados previamente. En concreto, podemos ver el tiempo transcurrido desde el naufragio, el tiempo transcurrido desde la partida de los vehículos de rescate, el número de náufragos rescate y la media y la varianza aprendidas del rumbo y la velocidad de los náufragos encontrados. El aspecto de esta ventana es el siguiente:



Tema 11: Creación de terrenos y escenarios

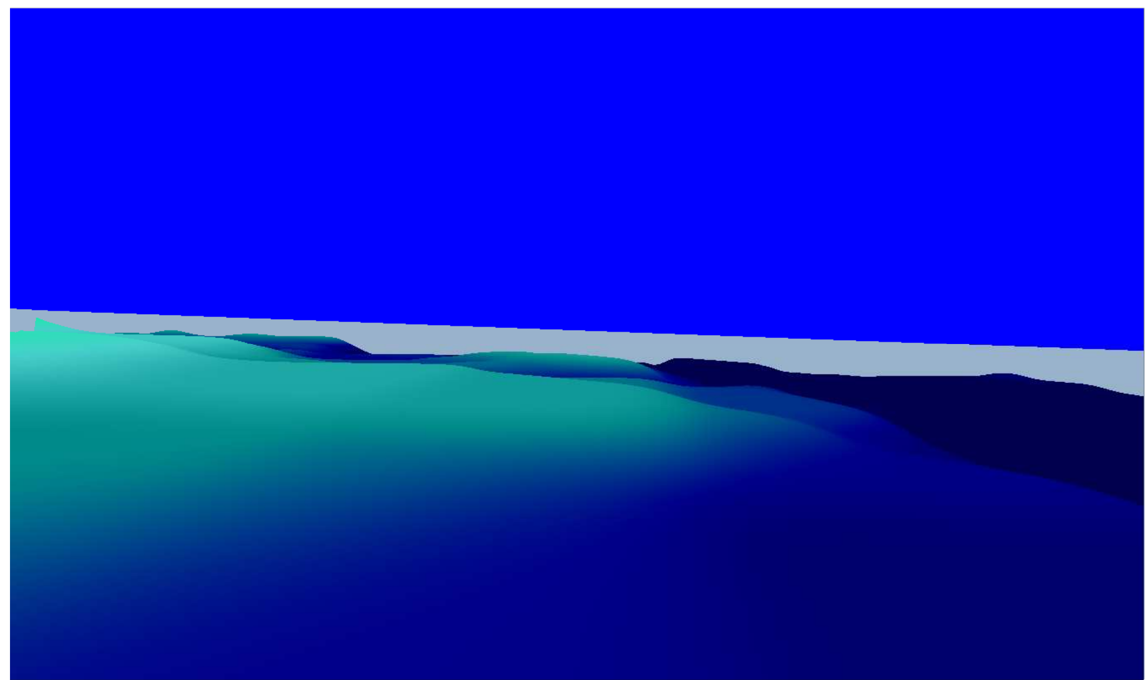
La generación de terrenos se realiza de forma algorítmica al comienzo de cada ejecución con los datos proporcionados por el usuario. Existen tres tipos de terreno diferente:

- Terreno de montaña: Tipo de terreno generado para la extensión del proyecto y la realización de tareas como la extinción de incendios. Para generarse necesita la información:
    - Longitud del terreno en km
    - Anchura del terreno en km
    - Numero de iteraciones que queremos que realice el algoritmo de generación del terreno. Cuantas más iteraciones realice el algoritmo de generación del terreno, obtendremos mejores resultados. Un valor con el que se obtienen unos resultados aceptables es 32.
    - Altura máxima del terreno en m.
    - Altura mínima del terreno en m.
    - Valor del filtro de erosión. Con este filtro se "erosionan" las posiciones más elevadas del terreno homogeneizándolas con las posiciones adyacentes. El valor máximo del filtro es 1 y el mínimo es 0. Obtenemos resultados aceptables con valores en torno a 0,3.



10. Ejemplo de terreno de montaña generado con el simulador

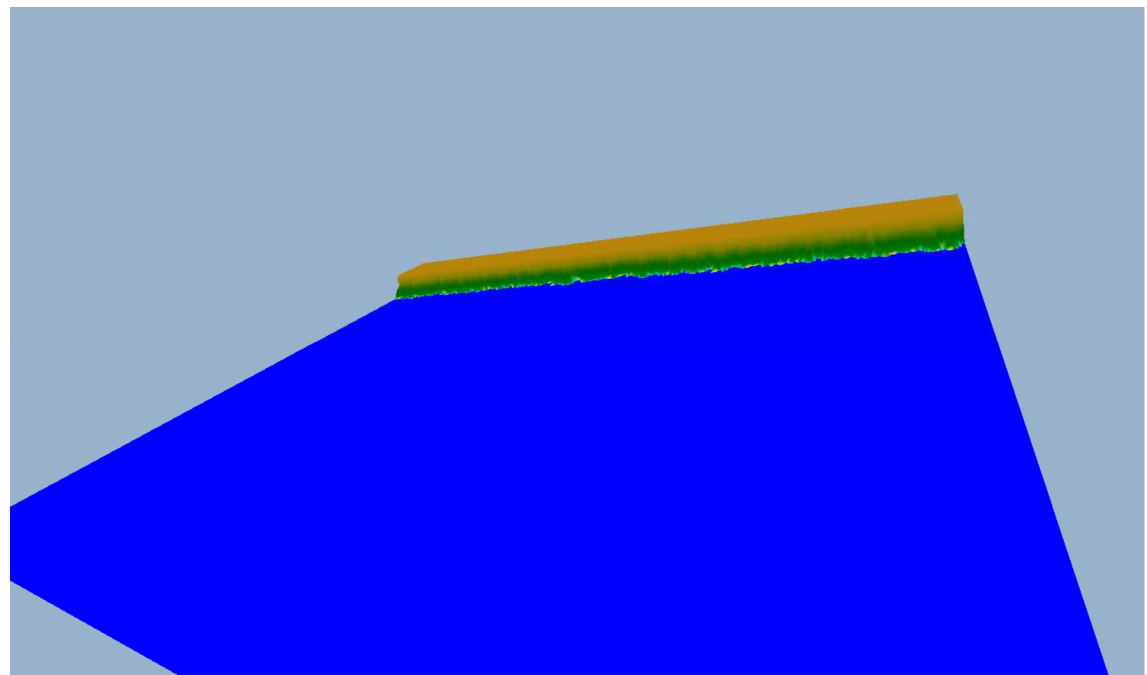
- Terreno de mar: Tipo de terreno utilizado para la simulación del rescate de náufragos. Necesita los siguientes parámetros:
    - Longitud del terreno en km.
    - Anchura del terreno en km.
    - Numero de iteraciones que queremos que realice el algoritmo de generación del terreno. Similar al terreno anterior.
    - Profundidad máxima del terreno en m.
    - Profundidad mínima del terreno en m.
    - Valor del filtro de erosión. Similar al terreno anterior.



11. Ejemplo de terreno de mar generado con el simulador

- Terreno de costa: Otra variante de terreno que podemos utilizar para utilizar como entorno de nuestras simulaciones. En este caso, se crea tanto un terreno de montaña y a partir de un margen delimitado por el usuario, se comienza la creación de un entorno marino, para proceder tras ello a la erosión del conjunto. Los parámetros necesarios para la creación de este tipo de terreno son los siguientes:

- Longitud del terreno en km.
- Anchura del terreno en km.
- Numero de iteraciones que queremos que realice el algoritmo de generación del terreno. Similar al terreno anterior.
- Altura máxima del terreno en m.
- Profundidad máxima del terreno en m.
- Valor del filtro de erosión. Similar al terreno anterior.
- Porcentaje del terreno que queremos que sea mar.



12. Ejemplo de terreno de costa generado por el simulador

La representación del terreno se hace a través de una matriz de dimensiones correspondientes con la longitud en km. del terreno. Por lo tanto, el algoritmo de generación de terreno dados los parámetros del mismo, devuelve una matriz de estas dimensiones, donde cada valor de la matriz representa la altura del terreno en ese punto.

11.1. Algoritmos de generación de terreno

11.1.1. Algoritmo de generación de terrenos de montaña y mar

Puesto que los terrenos de montaña y mar son prácticamente similares, se utiliza el mismo algoritmo para su generación. Para la creación de este tipo de terrenos se sigue el proceso:

1. Se inicializa la matriz de terreno a cero.
2. Por cada una de las iteraciones de generación de terreno:
   - 2.1. Se calcula una altura no superior a la altura máxima (o profundidad máxima en el caso del terreno de mar) en función de la iteración (la altura máxima para la primera iteración y 0 para la última).
   - 2.2. Se eligen dos puntos aleatorios del mapa, asegurándose de que no sean el mismo, y se calcula un vector que va desde el primero al segundo.
   - 2.3. Se recorre la matriz del terreno haciendo lo siguiente:
     - 2.3.1. Se calcula un vector desde el primer punto calculado aleatoriamente hasta el punto actual del mapa.
     - 2.3.2. Si el resultado del producto vectorial de los vectores calculados es mayor que 0, a la posición actual se le suma la altura calculada en el paso 2.1.
   - 2.4. Se filtra el terreno con el filtro establecido
3. Una vez generado el terreno se normaliza entre las alturas máxima y mínima (o entre las profundidades máxima y mínima).

11.1.2. Algoritmo de generación de terrenos de costa

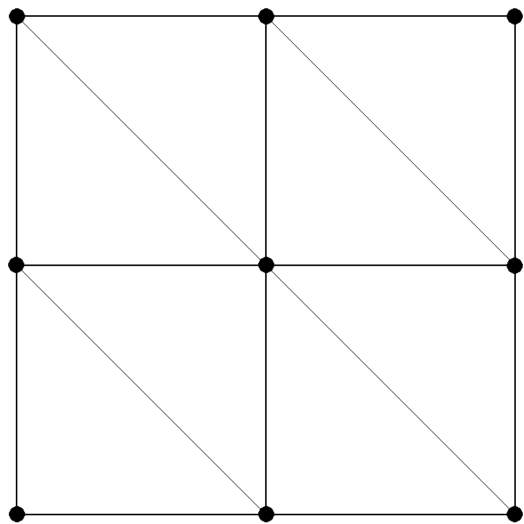
El algoritmo de generación de terreno de costa es bastante similar al de generación de terrenos de mar y de montaña y es el siguiente:

1. Se inicializa la matriz de terreno a cero.
2. Por cada una de las iteraciones de generación de terreno:
   2.1. Se calcula una altura no superior a la altura máxima en función de la iteración (la altura máxima para la primera iteración y 0 para la última).
   2.2. Se eligen dos puntos aleatorios del mapa, asegurándose de que no sean el mismo, y se calcula un vector que va desde el primero al segundo.
   2.3. Se recorre la matriz del terreno, recorriendo en anchura sólo hasta en punto en el entorno del porcentaje de mar pasado como parámetro. Para todos los puntos anteriores a ese punto:
      2.3.1. Se calcula un vector desde el primer punto calculado aleatoriamente hasta el punto actual del mapa.
      2.3.2. Si el resultado del producto vectorial de los vectores calculados es mayor que 0, a la posición actual se le suma la altura calculada en el paso 2.1.
   2.4. Para todos los puntos posteriores a este punto, se les resta la profundidad máxima.
   2.5. Se filtra el terreno con el filtro establecido
3. Una vez generado el terreno se normaliza entre la altura máxima y la profundidad máxima.

11.2. Algoritmo de dibujo del terreno.

El terreno no forma parte de la estructura de objetos 3D puesto que la representación de este como una malla requiere la generación de multitud de objetos que hacen que la memoria de la que dispone la máquina virtual de java para la ejecución de la simulación se agote.

Por lo tanto es necesaria la búsqueda de un sistema de dibujar el terreno de forma que no requiera la creación de tantos objetos. Para ello, de divide la cuadrícula formada por la matriz de terreno en triángulos que poder dibujar. Cada uno de los triángulos que se pueden ver en la siguiente imagen, corresponde a un plano dibujado en la interfaz 3D.



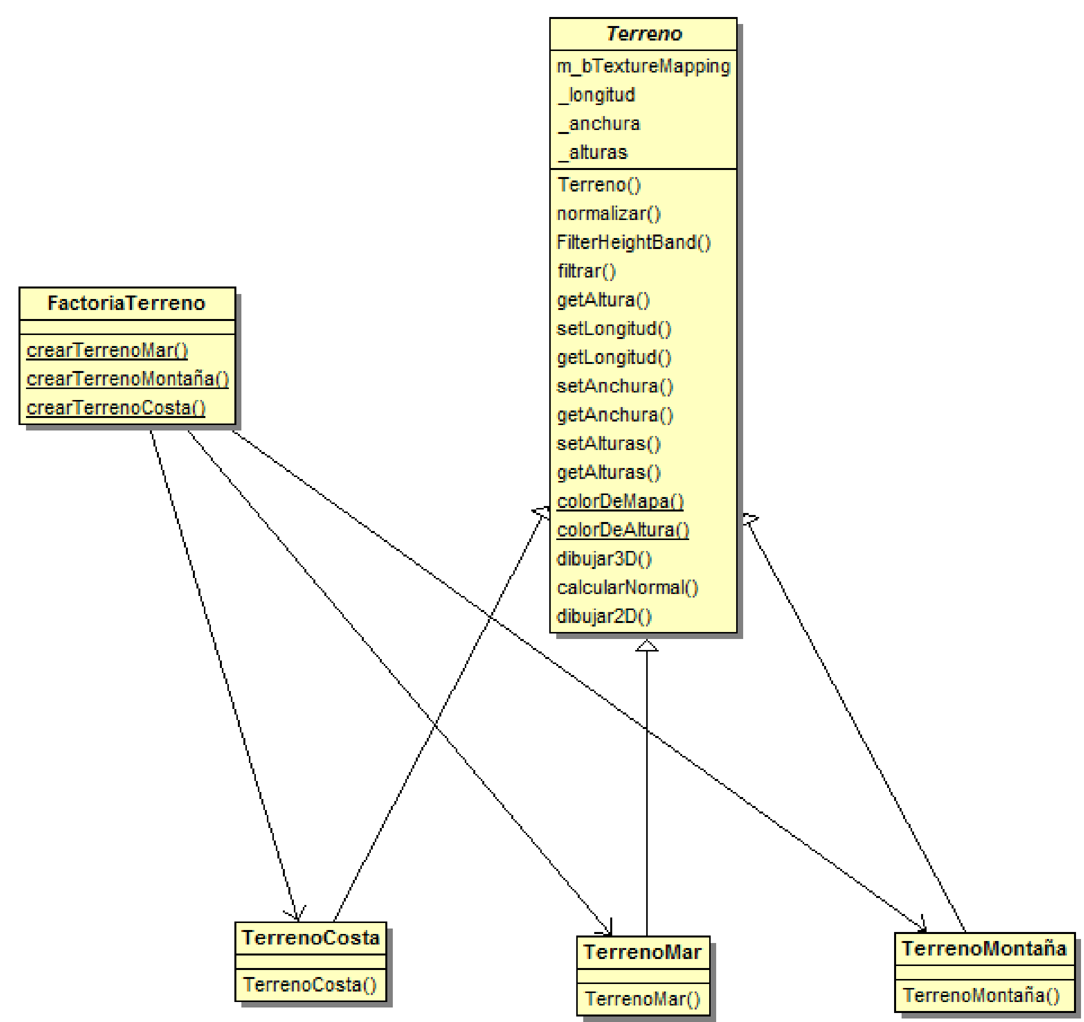
13. División del terreno en triángulos

Para conseguir un efecto más realista del terreno, en vez de dibujar cada triángulo de un color homogéneo, se asigna a cada vértice del triángulo un color en función de su altura, por lo que gracias a la tubería grafica de JOGL se consigue un efecto de degradado que da realismo a la imagen obtenida.

Por lo tanto, para el dibujo del terreno se recorre toda la matriz de alturas del terreno dibujando cada uno de los triángulos mostrados en la imagen superior. Se le asigna a cada uno de los vértices un color en función de su altura, por lo que la superficie resultante adquiere un color degradado. Por último, para dibujar la superficie del mar, se comprueba si todos los vértices que forman un triángulo están sumergidos, lo que indica que ese triángulo forma para del fondo marino. Si es así, además del triángulo propio del terreno, se dibujará otro triángulo en la misma posición, pero con altura 0 y color azul, para simular la superficie del mar.

11.3. Diseño de la construcción de terrenos

Para facilitar la creación de los terrenos, se ha implementado un patrón factoría. El diseño UML del mismo es el siguiente:



14. Diagrama UML de la construcción del terreno

La clase terreno implementa todo el comportamiento propio de los terrenos, como son los métodos dibujar3D() o dibujar2D(), puesto que son comunes para todo tipo de terrenos. Las clases TerrenoCosta, TerrenoMar y TerrenoMontaña, y su único método es un constructor, puesto que en lo único que difieren los unos de los otros es en su forma de crear la matriz de alturas que representa al terreno.

A la hora de crear un terreno, basta con utilizar el método apropiado de la clase FactoriaTerreno con los parámetros adecuados. La factoría devuelve un objeto de la clase Terreno instanciado con un objeto de tipo TerrenoCosta, TerrenoMar o TerrenoMontaña. Con esto conseguimos que sólo se instancien estas clases dentro de la factoría, por lo que si cambia la clase que implementa cualquiera de ellas, el código sólo deberá se modificado dentro de la factoría y el resto del código permanecerá igual.

Tema 12: Simulación en entornos reales: Google Earth

Una vez tenemos construida una interfaz de simulación 2D y 3D para poder visualizar los resultados de nuestras simulaciones, decidimos dar un paso más y llevar nuestro proyecto a un entorno de simulación real. Para ello decidimos utilizar la herramienta Google Earth, puesto que es una herramienta libre de gran potencia y gran auge actualmente para la planificación y visualización de simulaciones.

Lo que pretendemos conseguir con esto, es tener una herramienta de geovisualización completa que nos permita utilizar nuestro simulador en entornos reales, y visualizar los resultados de las simulaciones en tiempo real, igual que lo hacemos con las interfaces 2D y 3D creadas por nosotros, sobre territorios reales como son los que podemos encontrar en esta herramienta.

Los resultados de incluir una herramienta como esta en nuestro proyecto son claros: utilización de imágenes reales de los territorios sobre los que estamos trabajando, la posibilidad de configurar nuestras simulaciones en cualquier parte del mundo, la gran capacidad de trabajo que ofrece Google Earth y su interfaz de manejo y lo más importante: la capacidad de ver nuestras simulaciones en cualquier parte del mundo sin necesidad del simulador, sin más que compartir un fichero.

Para ello, en primer lugar hemos investigado la forma de trabajar con Google Earth. Google Earth obtiene los datos que visualiza de ficheros .kml (ficheros .xml con unas etiquetas especiales propias de Google Earth), o ficheros .kmz (ficheros .zip en los que están comprimidos los ficheros .kml anteriormente nombrados, junto que los ficheros que utiliza). La sintaxis de los ficheros .kmz en el apéndice acerca de los mismos al final de esta memoria.

A continuación se describe el proceso seguido así como los archivos generados para poder obtener los resultados de nuestras simulaciones en tiempo real sobre Google Earth.

Para poder realizar la visualización de la simulación, esta durante la ejecución, crea dos ficheros:

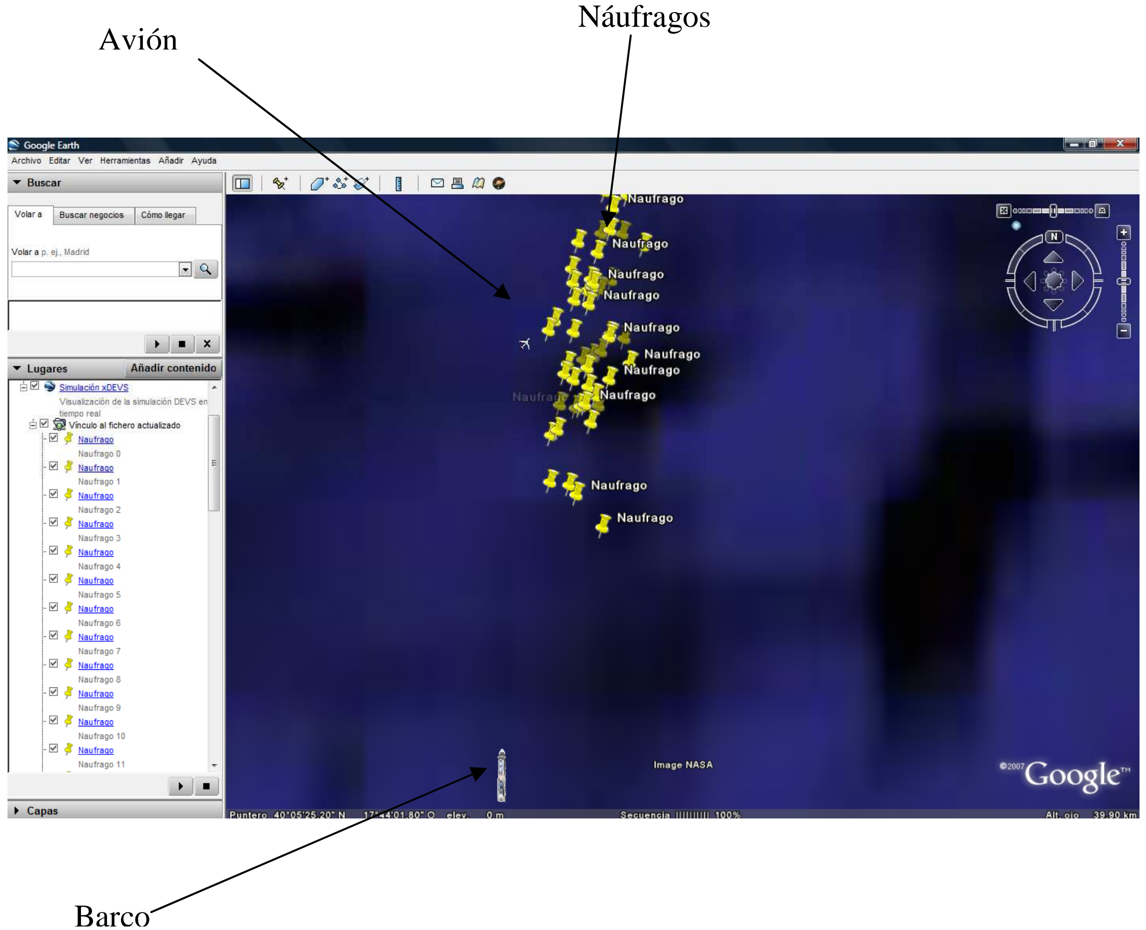
- principal.kml: Fichero que establece una referencia al fichero secundario.kml, y que establece el tiempo de refresco, transcurrido el cuál, Google Earth vuelve a cargar el fichero. Este tiempo de refresco vendrá definido como un parámetro establecido por el usuario al principio de la ejecución.
- Secundario.kml: Fichero que se encarga del posicionamiento en Google Earth de los elementos de la simulación.

Para visualizar los elementos de la simulación, se utilizan los siguientes modelos:

- Para la visualización de los aviones se utiliza un modelo de avión 3D que representa a un Boeing 747.
- Para la representación de los barcos se utiliza un modelo de un trasatlántico.
- Para la representación de los náufragos se utiliza un marcador de posición en forma de chincheta.

Para simulaciones grandes (más de 100 náufragos), se recomienda no utilizar un tiempo de refresco inferior a 5 segundos, para que el simulador tenga tiempo de sobra a escribir el fichero antes de que Google Earth lo lea. Para simulaciones pequeñas, se pueden utilizar tiempos de refresco en torno a los 2 segundos.

El resultado de la visualización en Google Earth de una simulación es el siguiente:



15. Ejemplo de la visualización de una simulación en Google Earth

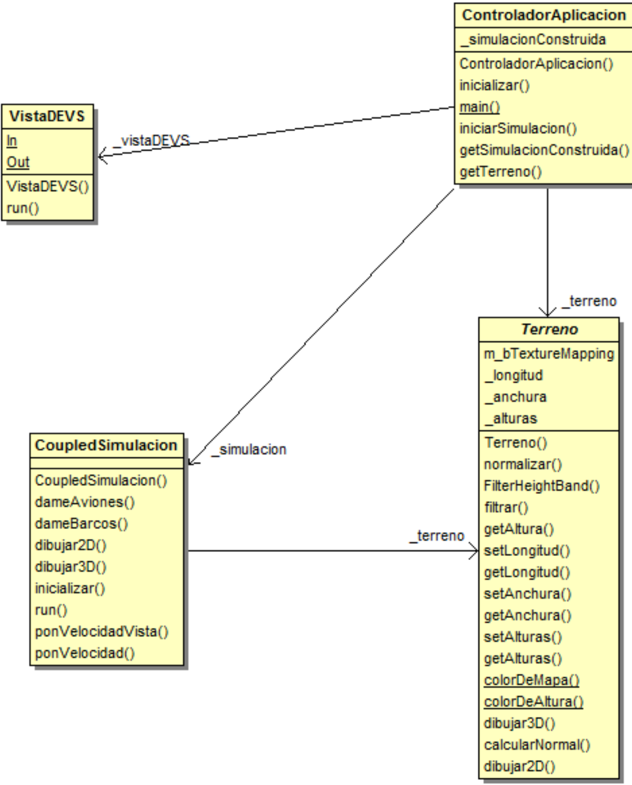
Tema 13: Integración de las partes

La integración de los diversos componentes que integran la aplicación se encuentra en la clase ControladorAplicacion. Esta clase es la encarga de lanzar las dos simulaciones DEVS que integran el sistema:

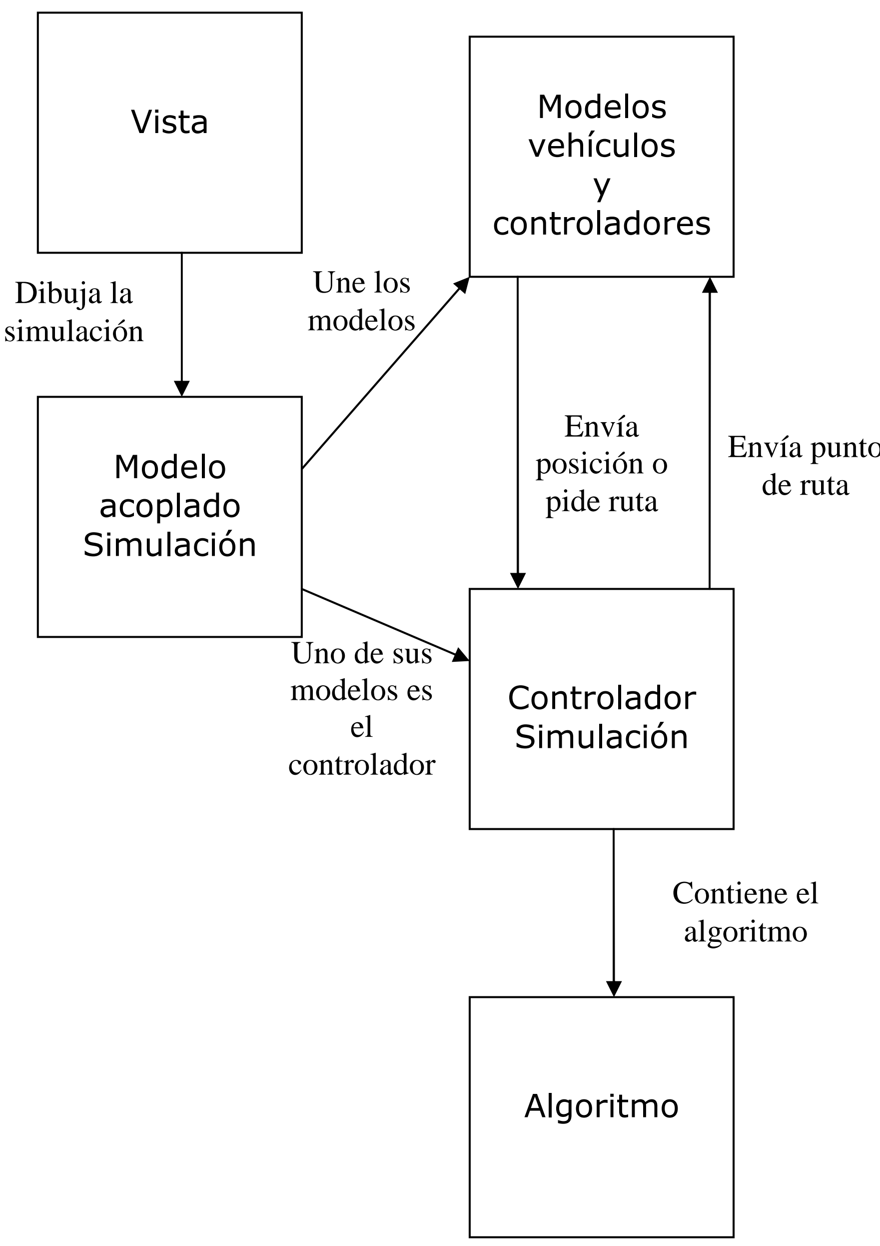
- Por una parte la simulación que se encarga de la simulación del entorno de rescate y simulación de vehículos.
- Por otra parte la simulación que se encarga de la visualización de los datos anteriores (interfaz gráfica).

Se ha optado por lanzar las simulaciones de forma independiente a través de dos hilos distintos para que la ejecución de la una no ralentice la ejecución de la otra y viceversa. De esta forma se pretende seguir con una visualización normal a pesar de que los cálculos necesarios por el algoritmo o por la simulación de los vehículos ralenticen el sistema.

El diagrama UML que representa el diseño de la integración de los componentes es el siguiente:



En forma de diagrama, podemos visualizar esta integración de la forma siguiente:



Apéndice A: Memoria XML

Introducción

Esta parte de la aplicación esta orientada a una comunicación entre distintos computadores de la misma o distintas redes. Se han diseñado dos funciones de utilidades opuestas, una pasa de un objeto java a un documento XML con toda la información de este objeto xDevs, y la otra dado un documento XML crea un objeto xDevs.

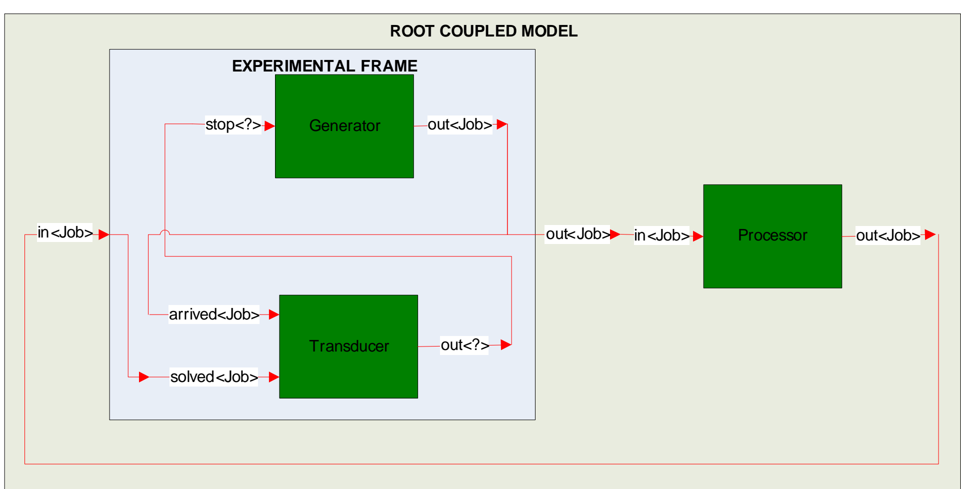
Formato del documento XML

El documento XML dado a la aplicación y el resultante de la función tienen que ser de esta forma:

```
      <Coupled Name=”Efp” File=”Efp.java” Host=”192.168.1.3”>
       <Coupled Name=”Ef” File=”Ef.java” Host=”192.168.1.9”>
        <Ports>
         <InPort Name=”in”/>
         <OutPort Name=“out”/>
        </Ports>
        <Atomic Name=”Generator” File=”Generator.java” Host=”…”>
         <Ports>
          <InPort Name=”stop”/>
          <OutPort Name=“out”/>
         </Ports>
         <GlobalState>
          <Sigma value=”2.5”/>
          <Phase value=”GeneratingJobs”/>
         </GlobalState>
         <States>
          <State Name=”period” Type=”Double” Value=”2.5”/>
          <State Name=”count” Type=”Integer” Value=”0”/>
         <States>
        </Atomic>
        <Atomic     Name=”Transducer”     File=”Transducer.java”
Host=”…”>
         <Ports>
          <InPort Name=”arrived”/>
          <InPort Name=”solved”/>
          <OutPort Name=“out”/>
         </Ports>
         <GlobalState>
```

```
          <Sigma value="100"/>
          <Phase value="Generating stats"/>
         </GlobalState>
         <States>
          <State       Name="numJobsArrived"       Type="Integer"
Value="0"/>
          <State Name="numJobsSolved" Type="Integer" Value="0"/>
          <State       Name="observationTime"       Type="Double"
Value="100"/>
          <State Name="totalTa" Type="Double" Value="0"/>
          <State Name="clock" Type="Double" Value="0"/>
         <States>
        </Atomic>
       <Connections>
        <Connection ComponentFrom="Ef" PortFrom="in"
                    ComponentTo="Transducer" PortTo="solved"/>
        <Connection ComponentFrom="Generator" PortFrom="out"
                    ComponentTo="Ef" PortTo="out"/>
        <Connection ComponentFrom="Generator" PortFrom="out"
                    ComponentTo="Transducer" PortTo="arrived"/>
        <Connection ComponentFrom="Transducer" PortFrom="out"
                    ComponentTo="Generator" PortTo="stop"/>
      </Connections>
       </Coupled>
       <!—Aquí más modelos atómicos y/o acoplados -->
      </Coupled>
```

Que se correspondería con este modelo DeVs:



El campo host generado es actualmente vacio, ya que no se ha hecho orientado a una red especifica, por lo que debe ser cambiado para utilizarlo.

Apéndice B: Formato de los ficheros .kml

Los ficheros .kml son básicamente ficheros xml con etiquetas especiales que pueden ser interpretadas por los Earth browsers de Google. Su formato se basa en el uso de bloques que definen cada uno de los elementos que pueden utilizarse. Los más importantes son los siguientes:

Encabezado de los ficheros .kml

Todo fichero .kml debe contener la siguiente cabecera y final de fichero:

```
<?xml version=''1.0'' encoding=''UTF-8''?>
<kml xmlns=''http://earth.google.com/kml/2.2''>
      --> Cuerpo del fichero .kml
</kml>
```

En el interior del cuerpo del fichero .kml pueden incluirse de manera consecutiva todos los elementos enumerados a continuación hasta formar el fichero .kml completo con la información que queremos visualizar.

Localización de posiciones (Placemarks)

Las posiciones en un fichero .kml vienen indicadas de la siguiente forma:

```
<Placemark>
      <name>Nombre del lugar</name>
      <description>Descripción del lugar</description>
      <Point>
            <coordinates>Longitud del punto (grados), latitud del
      punto (grados),
            altura del punto (m sobre el mar)</coordinates>
      </Point>
</Placemark>
```

Las posiciones se indican en el mapa con una chincheta amarilla. Podemos utilizar código HTML dentro de la descripción del lugar de la siguiente forma:

```
<description>
        <![CDATA[
          <h1>CDATA Tags are useful!</h1>
          <p><font color="red">Text is <i>more readable</i> and
          <b>easier to write</b> when you can avoid using entity
          references.</font></p>
        ]]>
</description>
```

Capas (GroundOverlays)

Nos permiten visualizar una imagen sobre el terreno. La forma de definirlas es la siguiente:

```
<GroundOverlay>
      <name>Nombre de la capa</name>
      <description>Descripción de la capa</description>
      <Icon>
               <href>Dir. Imagen a visualizar</href>
      </Icon>
      <LatLonBox>
        <north>Límite norte de la imagen</north>
        <south>Límite sur de la imagen</south>
        <east>Límite este de la imagen</east>
        <west>Límite oeste de la imagen</west>
        <rotation>Rotación de la imagen</rotation>
      </LatLonBox>
</GroundOverlay>
```

Caminos (Paths)

Los caminos, como su propio nombre indican nos permiten definir caminos a través de una sucesión de puntos unidos a través de una línea continua. Para generarlos se utiliza el siguiente código:

```
<LineString>
        <extrude>1</extrude>
        <tessellate>1</tessellate>
        <altitudeMode>absolute</altitudeMode>
        <coordinates> longitud, latitud, altura
                       (para todos los puntos del camino)
        </coordinates>
</LineString>
```

El campo <extrude> <\extrude> especifica si los puntos deben estar conectados mediante una línea al suelo (1 para conectarlos 0 para no conectarlos).

El campo <tessellate> <\tessellate> habilita al camino seguir la curvatura de la tierra (1 activado, 0 desactivado).

El campo <altitudeMode> <\altitudeMode> indica si las alturas de los puntos especificados a continuación son relativas a la altura del mar, a la altura del terreno o si deben ser ignoradas (clampToGround: las alturas serán ignoradas y el camino irá pegado al suelo, relativeToGround: las alturas son relativas a la altura del terreno, absolute: las alturas son respecto al nivel del mar)

Polígonos (Polygons)

En kml también podemos definir polígonos con la siguiente structura:

El campo <outerBoundaryIs> <\outerBoundaryIs> indica que se va a definir el exterior del polígono. Debe tener siempre un campo del tipo <LinearRing> que definirá este perímetro.

El campo <innerBoundaryIs> <\innerBoundaryIs> es opcional e indica que se va a definir el perímetro interno del polígono. Puede tener tantos campos del tipo <LinearRing> como sean necesarios.

```
<Polygon>
      <extrude>1</extrude>
      <altitudeMode>relativeToGround</altitudeMode>
      <outerBoundaryIs>
        <LinearRing>
        <coordinates> longitud, latitud, altura
                       (para todos los puntos del poligono)
        </coordinates>
        </LinearRing>
      </outerBoundaryIs>
      <innerBoundaryIs>
        <LinearRing>
        <coordinates> longitud, latitud, altura
                       (para todos los puntos del poligono)
        </coordinates>
        </LinearRing>
      </innerBoundaryIs>
    </Polygon>
```

Estilos (Style)

Se pueden modificar el estilo de todos los elementos descritos anteriormente mediante la etiqueta <Style> <\Style>. Cada elemento tiene un estilo asociado y por lo tanto una etiqueta <Style>

distinta. Para consultar como se define cada etiqueta y los elementos de cada una, por favor consultar el Reference Guide de kml: http://code.google.com/apis/kml/documentation/kml_tags_beta1.html

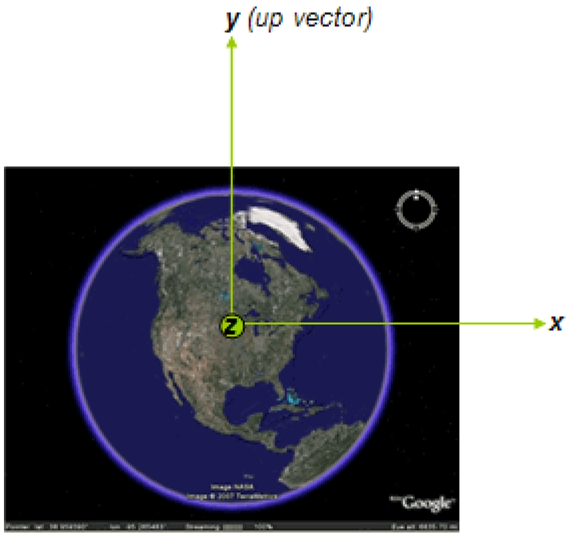
Operaciones con la camara

En cualquier punto del código kml podemos establecer la orientación de la cámara que más se adapte a nuestras necesidades. Esto lo podemos hacer gracias al siguiente código:

```
<Camera id="ID">
   <longitude>0</longitude>
   <latitude>0</latitude>
   <altitude>0</altitude>
   <heading>0</heading>
   <tilt>0</tilt>
   <roll>0</roll>
   <altitudeMode> </altitudeMode>
 </Camera>
```

Las etiquetas <longitude> <\longitude>, <latitude> <\latitude>, <altitude> <\altitude> definen respectivamente la longitud, la latitud y la altura en la que se encuentra la cámara. El campo <heading> <\heading> define la dirección (azimuth) de la cámara en grados (0 norte hasta 360). El campo <tilt> <\tilt> define el ángulo de rotación de la cámara respecto a su eje X (en grados 0-180). Por último el campo <roll> <\roll> define el ángulo de rotación de la cámara respecto al eje z (valores en grados entre -180 y +180).

La siguiente imagen muestra los ejes de la cámara en GoogleEarth:



Manejo del tiempo

La etiqueta <TimeSpan> describe un período de tiempo. Su sintaxis es la siguiente:

```
<TimeSpan id="ID">
  <begin>comienzo</begin>
  <end>final</end>
</TimeSpan>
```

La etiqueta <TimeStamp> define un instante en el tiempo. Su sintaxis es:

```
<TimeStamp id=ID>
  <when>...</when>
</TimeStamp>
```

Mediante la combinación de las operaciones de la cámara y las posibilidades del manejo de tiempos podemos conseguir que el usuario experimente en primera persona el recorrido de la ruta creada por el simulador.

Para conocer el resto de las etiquetas existentes en kml:

http://code.google.com/apis/kml/documentation/kml_tags_beta1.html

Palabras clave para su búsqueda bibliográfica

1. Simulación
2. Devs
3. UAV
4. Rescate
5. Genético
6. Aprendizaje
7. Control
8. OpenGL
9. Google Earth
10. XML

Bibliografía

- Intelligent Optimisation Techniques: Genetic Algorithms, Tabu Search, Simulated Annealing and Neural Networks by D.T. Pham
- Focus on 3D terrain programming by/ Trent Polack Publicac. Cincinnati (Ohio): Premier Press, cop.
- Java 2. Vol. 1, Fundamentos / Cay S. Horstmann, Gary Cornell ; traducción, KME Sistemas, S.L. Publicac. Madrid Prentice-Hall, D.L. 2003
- Computer graphics using OpenGL / F.S. Hill Publicac. Upper Saddle River (New Jersey) : Prentice Hall, cop. 2001
- Introduction to DEVS Modeling and Simulation with Java: Developing Component-Based Simulation Models by / Bernard P. Zeigler
- Programación en 3d con java 3d. by/ J.J. Pratdepadua Bufill
- Marine Systems Simulators http://www.marinecontrol.org

Autorización a la difusión

Autorizamos a la Universidad Complutense a difundir y utilizar con fines académicos, no comerciales y mencionando expresamente a sus autores, tanto la propia memoria, como el código, la documentación y/o el prototipo desarrollado.

Sergio González Sanz

David Rodilla Rodríguez

Carlos Sánchez García