UNIVERSITAT POLITÈCNICA DE VALÈNCIA

Escuela Técnica Superior de Ingeniería del Diseño

# Mejora en la automatización del perfil vertical de un simulador de vuelo

Trabajo Fin de Grado

Grado en Ingeniería Electrónica Industrial y Automática

AUTOR/A: López Monreal, Carlos

Tutor/a: Yuste Pérez, Pedro

Cotutor/a: Vila Carbó, Juan Antonio

CURSO ACADÉMICO: 2021/2022

Escuela Técnica Superior de Ingeniería del Diseño

Universidad Politécnica de Valencia

Grado en Ingeniería Electrónica Industrial y Automática

TRABAJO FIN DE GRADO

# MEJORA EN LA AUTOMATIZACIÓN DEL PERFIL VERTICAL DE UN SIMULADOR DE VUELO

Autor: Carlos López Monreal

Tutor: Pedro Yuste Pérez

Cotutor: Juan Antonio Vila Carbó

Valencia, Junio 2022

# Agradecimientos

Debo agradecer que a día de hoy este trabajo sea una realidad, tanto a aquellas personas que dudaban de mi dedicación, como a aquellas que siempre han estado ahí. Las primeras me han dado la fuerza para no rendirme y seguir adelante, las segundas me han hecho el camino más llevadero y me han hecho sentir querido y apoyado a lo largo de estos años.

Hago especial mención a toda mi familia, por su gran apoyo y amor incondicional en todo momento. A mi gran amigo Fran, cuyos consejos han sido fundamentales para mí en este trabajo. A Álvaro, buen amigo de la infancia y con el que he tenido el placer de sentirme respaldado en muchos momentos de flaqueza durante este proyecto. Y por supuesto, a mi tutor, por su comprensión hacia mí y su tiempo.

A todos aquellos que han compartido tanto los buenos como los no tan buenos momentos durante estos años, mi más sincero agradecimiento por brindarme la oportunidad de ser quien soy y de haber conseguido todo lo que me he propuesto.

# Resumen

El presente trabajo fin de grado, pretende servir como mejora dentro de un sistema ya creado en otro trabajo fin de grado. Es necesario una breve explicación de los aspectos más importantes del trabajo que se ha querido mejorar, de forma que sea más fácil contextualizar este proyecto.

El sistema del que se parte consiste en un programa que simula el control de tráficos aéreos. Para ello, se crea un simulador de vuelo basado en las ecuaciones de BADA (Base of Aircraft Data) Revisión 3.9. de EUROCONTROL, de donde se recogen todas las ecuaciones necesarias para la modelización de las dinámicas de cada aeronave. A parte, se hace uso de parámetros y coeficientes de los archivos de BADA, a los que el programa tiene acceso en forma de archivos de texto. El lenguaje de programación utilizado es Java, ya que debido a su portabilidad y facilidad de ejecución en cualquier sistema operativo resulta idóneo para un proyecto de este tipo. El entorno de desarrollo que se utiliza en ambos casos, tanto en este proyecto como en el proyecto del que se parte es NetBeans, más concretamente la versión 8.2.

En cuanto a las mejoras que se han realizado en este programa, lo que se ha conseguido es la representación a tiempo real del perfil vertical de un avión mientras realiza un vuelo, mejorando de forma conjunta el comportamiento de este sobre todo en los descensos, con el objetivo de que cumpla con todas las restricciones que tenga impuesta la ruta. Por otra parte, se ha querido simular la dinámica para el aviso del despliegue de spoilers del avión, además de automatizar su uso, es decir, es el programa el que decide en qué momento debe avisar al piloto de la recomendación de uso de spoilers. Cuando una aeronave debe forzar un descenso con más pendiente de lo habitual, utiliza los spoilers, superficies supersustentadoras situadas en las alas que ayudan a un descenso con más pendiente. En el cuadro de mandos de cada aeronave existe un indicador que se enciende en caso de considerar que dicho despliegue es necesario. En este programa podemos forzar esta situación en la que el avión avisa al piloto en caso de necesitar los spoilers en el descenso.

Todo esto, se va a explicar más adelante en detalle, viendo las clases creadas, las modificaciones realizadas y analizando de forma gráfica los cambios que se han ido haciendo.

# Índice General

Índice 9

Índice de figuras 11

Índice de tablas 15

Índice de ecuaciones 15

Glosario 16

Siglas 18

Notación 19

# Índice

1. Introducción……21
   1.1. Motivación……21
   1.2. Objetivos……22
   1.3. Estado del arte……23
   1.4. Materiales y Métodos……26
2. Pliego de condiciones……28
   2.1. Pliego de condiciones generales……28
      2.1.1. Condiciones de compatibilidad……28
      2.1.2. Condiciones de la interfaz gráfica……29
      2.1.3. Condiciones específicas del guiado del perfil vertical……30
         2.1.3.1. Modelo de prestaciones BADA……30
         2.1.3.2. Restricciones SID, STAR e IAC……31
   2.2. Pliego de especificaciones técnicas……31
      2.2.1. Especificaciones de materiales y equipos……31
         2.2.1.1. Herramientas hardware……32
         2.2.1.2. Herramientas software……33
      2.2.2. Especificaciones de ejecución……33
         2.2.2.1. Código……33
            2.2.2.1.1. Introducción a la escritura del código……33
            2.2.2.1.2. Clase VerticalRoute……35
            2.2.2.1.3. Otras clases……49
         2.2.2.2. Interfaz gráfica……57
            2.2.2.2.1. Características de la interfaz……58
            2.2.2.2.2. Evolución de la interfaz……68
3. Planos……80
   3.1. Introducción al lenguaje UML……80
   3.2. Introducción al diagrama de clases……81
      3.2.1. Clases……81
      3.2.2. Relaciones……82
   3.3. Diagrama de clases……83
      3.3.1. Módulos del diagrama……83
      3.3.2. Diagrama de clases simplificado……102
4. Proceso de desarrollo……103
   4.1. Fase preliminar y de aprendizaje……103
      4.1.1. Sub-fase de aprendizaje sobre terminología aeroespacial……103
      4.1.2. Sub-fase de aprendizaje sobre Java y POO……104
   4.2. Fase de lectura de código……104
   4.3. Fase de planificación y análisis……105
   4.4. Fase de diseño……105
   4.5. Fase de implementación y testeo……105
   4.6. Fase de puesta a punto……107

5. Estudio económico……………………………………………………………108
   5.1. Diagrama de Gantt…………………………………………………...…108
   5.2. Presupuesto…………………………………………………………..108
6. Conclusión…………………………………………………………………110
7. Ampliaciones y desarrollo futuro……………………………………………..112
8. Bibliografía…………………………………………………………………114

# Índice de figuras

Figura 1: Display de aviones Boeing con sistemas VSD………………………………23

Figura 2: Interfaz FlightRadar24 para la monitorización de un vuelo………………....24

Figura 3: gráfica del perfil vertical del avión. Velocidad y altitud respecto del tiempo.24

Figura 4: Interfaz FlightAware para la monitorización de un avión a tiempo real….…25

Figura 5: Perfil vertical de un avión en FlightAware…………………..…….………...26

Figura 6: Indicador SPEED BRAKE ARMED de un Boeing 737 activado en el simulador X-Plane……………………………………………………………………...29

Figura 7: Obtención del ángulo de un nivel de vuelo a otro, dadas las velocidades vertical y horizontal……………………………………………………………………….39

Figura 8: Supuesto de incumplimiento de restricciones………………………………..40

Figura 9: Rectificación ruta teórica……………………………………………………41

Figura 10: Panel con datos dinámicos del avión. Fragmento de la interfaz de usuario original…………………………………………………………………………………42

Figura 11: Situación de viraje, visto desde arriba……………………………………...44

Figura 12: Modo de actuación en los virajes…………………………………………..44

Figura 13: Simbología para la representación de restricciones en la gráfica…………47

Figura 14: Rectificación en caso de restricción de altitud determinada……………….48

Figura 15: Rectificación en caso de restricción de altitud determinada……………….50

Figura 16: Cálculo dato distancia en ascenso………………………………………….51

Figura 17: JTextField dedicado a introducir el valor de altitud………………………..52

Figura 18: Posicionamiento incorrecto y correcto del TOD…………………………..53

Figura 19: Caso de descenso escalonado, con dos TODs………………………………55

Figura 20: Numero de direcciones e índice de cada waypoint…………………………56

Figura 21: Diseño final de la interfaz final de usuario…………………………………57

Figura 22: Indicador SPEED BRAKE ARMED desactivado………………………….58

Figura 23: Indicador SPEED BRAKE ARMED activado……………………………..58

Figura 24: Dimensiones del indicador SPEED BRAKE ARMED en pixels…………..58

Figura 25: Localización del indicador SPEED BRAKE ARMED……………………..59

Figura 26: Diseño de la gráfica para el control del perfil vertical del avión…………...60

Figura 27: Grafica del perfil vertical del avión en movimiento………………………..60

Figura 28: Ejemplo sistema VSD 1……………………………………………………61

Figura 29: Ejemplo sistema VSD 2……………………………………………………62

Figura 30: Ejemplo sistema VSD 3……………………………………………………62

Figura 31: Dimensiones de la gráfica JFreeChart en pixels……………………………62

Figura 32: Localización gráfica perfil vertical…………………………………………63

Figura 33: Localización del TOC y visualización de las restricciones…………………65

Figura 34: Localización del TOD y visualización de las restricciones…………………65

Figura 35: Localización TODs descenso escalonado y visualización de las restricciones 1……………………………………………………………………………………...66

Figura 36: Localización TODs descenso escalonado y visualización de las restricciones 2……………………………………………………………………………………...67

Figura 37: Mensaje de aviso de valor FL incorrecto…………………………………...68

Figura 38: Diseño original de la interfaz………………………………………………69

Figura 39: Alineación de los elementos de la tabla de datos dinámicos………………..69

Figura 40: Primer diseño de la interfaz en nuestro proyecto…………………………...70

Figura 41: Evolución del perfil vertical 1………………………………………………71

Figura 42: Evolución del perfil vertical 2………………………………………………72

Figura 43: Evolución del perfil vertical 3……………………………………………73

Figura 44: Evolución del perfil vertical 4……………………………………………73

Figura 45: Evolución del perfil vertical 5……………………………………………74

Figura 46: Evolución del perfil vertical 6……………………………………………74

Figura 47: Evolución del perfil vertical 7 – estado final………………………………75

Figura 48: Evolución del perfil vertical 8 – estado final………………………………75

Figura 49: Evolución del perfil vertical 9 – estado final………………………………76

Figura 50: Evolución del perfil vertical 10 – estado final……………………………...76

Figura 51: Evolución del perfil vertical 11 – estado final……………………………...77

Figura 52: Evolución del perfil vertical 12 – estado final……………………………...77

Figura 53: Evolución del perfil vertical 13 – estado final……………………………...78

Figura 54: Evolución del perfil vertical 14 – caso de colisión y paro de la simulación..78

Figura 55: Evolución del perfil vertical 15 – aspecto de una ruta completa…………...79

Figura 56: Evolución del perfil vertical 16 – uso automático del indicador de aviso SPEED BRAKE ARMED……………………………………………………………...79

Figura 57: Representación genérica de una clase en UML……………………………81

Figura 58: Llamada de la clase 2 desde la clase 1……………………………………...82

Figura 59: Llamada de la clase 2 desde la clase 1 y viceversa…………………………82

Figura 60: Clase TrafficRadar…………………………………….……………………83

Figura 61: Clase Fijo……………………………………….…………………………..84

Figura 62: Clase Aeropuerto…………………………………….……………………..84

Figura 63: Clase VerticalRoute………………………………………………………...85

Figura 64: Clase PlanDeVuelo……………………………………….………………...85

Figura 65: Clase FligthLayer…………………………………………………….........86

Figura 66: Clase RouteLayer…………………………………………………….........86

Figura 67: Clase DatosBADA……………………………………………………........87

Figura 68: Clase Aproximación………………………………………………………...87
Figura 69: Clase DaotsENAIRE……………………………………………………..88
Figura 70: Clase DatosES………………………………………………….............88
Figura 71: Clase MensajeConexionES………………………………………………89
Figura 72: Clase MensajeDesconexiónTrafico………………………………………89
Figura 73: Clase MensajePosicionLibre……………………………………………..90
Figura 74: Clase TFG_Davinia………………………………………………………90
Figura 75: Clase MensajeElegirSim…………………………………………………91
Figura 76: Clase RecibeXPlane……………………………………………………...91
Figura 77: Clase Calculos……………………………………………………………92
Figura 78: Clase Trafico……………………………………………………………..93
Figura 79: Clase Restriccion…………………………………………..…………..93
Figura 80: Clase SIDSTAR…………………………………………………………94
Figura 81: Clase Punto………………………………………………………………94
Figura 82: Clase enMovimiento……………………………………………………..95
Figura 83: Clase Pista………………………………………………………………95
Figura 84: Clase MensajeUsuarioES………………………………………………..96
Figura 85: Clase MensajeSelecSID…………………………………………………97
Figura 86: Clase MensajeSelecSTAR………………………………………………97
Figura 87: Clase Visor……………………………………………………………....98
Figura 88: Clase SIMU……………………………………………………………...99
Figura 89: Clase MensajeConexiónXP……………………………………………100
Figura 90: Clase MensajeSelecPista………………………………………………..101
Figura 91: Clase MensajeRecibeES………………………………………………..101
Figura 92: Esquema UML del proyecto completo…………………………………102
Figura 93: Ejemplo de mejora del programa………………………………………112

# Índice de tablas

Tabla 1: Fragmento del archivo B738_.PTF……………………………………………...…38

Tabla 2: Tabla con los valores calculados de ángulos de ascenso/descenso………...…39

Tabla 3: Diagrama de Gantt del proyecto……………………………………………...108

Tabla 4: Presupuesto del proyecto…………………………………………………......109

# Índice de ecuaciones

Ecuación 1: Ecuación para la obtención del ángulo de ascenso/descenso……………..39

Ecuación 2: Ecuación para la obtención de la distancia de los puntos de la ruta teórica51

# Glosario

- **Array:** un array en Java, es una colección de variables de un mismo tipo. Pueden ser multidimensionales, no obstante, una vez creado, su tamaño es inmodificable.

- **ArrayList:** en lenguaje Java, un ArrayList es una colección de variables, que no tienen por que ser de un mismo tipo, son multidimensionales, pero además su tamaño sí que puede variar a lo largo de un programa, no como en el caso de los arrays.

- **BufferedImage:** es una clase que se utiliza para mantener una representación de una imagen en memoria en una aplicación Java, de forma que se pueda modificar o guardar en cualquiera de los formatos estándar propios de imágenes.

- **Display:** del inglés monitor.

- **Double:** cuando una variable es de tipo double, significa que puede almacenar valores de punto flotante, es decir, prácticamente cualquier número real.

- **JLabel:** una de las clases de Java Swing, la cuál permite mostrar texto no seleccionable e imágenes en nuestra interfaz.

- **JTextField:** una de las clases de Java Swing, la cuál permite crear un input que se utiliza para la captura de datos.

- **Método setter:** del inglés Set, que significa "establecer". Sirve para asignar un valor iniciar a un atributo.

- **Offset:** hacemos referencia al offset en una medida, cuando desplazamos el valor de referencia un cierto valor por arriba o por abajo.

- **Pause:** significa pausar, detener, parar.

- **Render:** el verbo render significa "reproducir" o "representar". Se utiliza para definir un proceso de cálculo desarrollado por un ordenador, destinado a producir una imagen o secuencia de imágenes.

- **Scrollbar:** en español barra de desplazamiento, es un objeto de la interfaz gráfica de usuario que permite desplazarnos, a lo largo de una gráfica, una imagen o una página web.

- **Shape:** es español "forma". En el proyecto se hace referencia a un "shape" cuando hablamos de los puntos que crea java por defecto en cada punto de la gráfica que estemos diseñando.

- **Swing:** Es una biblioteca gráfica para Java. Permite diseñar y crear interfaces gráficas de usuario.

- **Take off:** en español "despegar".

- **Thread:** en español **"**hilo". En Java nos referimos a los hilos cuando estamos hablando de las diferentes tareas que debe llevar el programa. Dichas tareas se organizan en hilos, y dichos hilos se pueden ejecutar de forma independiente uno de otro, sin la necesidad de ejecutarse de forma simultánea.

- **Waypoint:** concepto comúnmente utilizado en aeronavegación, para hacer referencia a los puntos que definen una ruta aérea. Dichos puntos están definidos por coordenadas

- **Zoom in/zoom out:** en español acercarse/alejarse.

# Siglas

- **API:** Interfaz de programación de aplicaciones, del inglés Application Programming Interface.
- **BADA:** Base of Aircraft Data.
- **CAS:** Velocidad calibrada, del inglés Calibrated Airspeed.
- **EOD:** End of Descent.
- **FAP:** Punto de aproximación final, del inglés Final Approach Point.
- **FL:** Nivel de vuelo, del inglés Flight Level.
- **GUI:** Interfaz gráfica de usuario, del inglés Grafic User Interface.
- **IAC:** Instrument Approach Chart.
- **IDE:** Entorno de Desarrollo integrado, del inglés Integrated Development Environment.
- **OACI:** Organización de Aviación Civil Internacional.
- **POO:** Programación Orientada a Objetos.
- **ROCD:** Rate of Climb/Descent.
- **SACTA:** Sistema Automatizado de Control del Transito Aéreo, del inglés Automatic Air Traffic Control System.
- **SID:** Salida instrumental normalizada, del inglés Standard Instrumental Departure.
- **STAR:** Ruta estándar de llegada terminal, del inglés Standard Terminal Arrival Route.
- **TAS:** Velocidad verdadera, del inglés True Airspeed.
- **TOC:** Top of Climb.
- **TOD:** Top of Descent.

- **UML:** Lenguaje unificado de modelado, del inglés Unified Modeling Language.
- **VNAV:** Sistema de navegación vertical, del inglés Vertical NAVigation system.
- **VS:** Velocidad vertical, del inglés Vertical Speed.
- **VSD:** Monitor de Situación Vertical, del inglés Vertical Situation Display.

# Notación:

Se ha escrito en cursiva todas aquellas palabras en inglés, además de toda referencia al código del proyecto, ya sean nombres de métodos, clases o variables. No obstante, no se aplicará a nombres propios, siglas o acrónimos de organismos públicos o privados.

# 1. Introducción

Después de estar estudiando conceptos relacionados con la ingeniería aeroespacial, en concreto dentro de la especialidad de aeronavegación, nos pudimos dar cuenta de que el control de tráfico aéreo tiene una gran importancia hoy en día, en un mundo donde se realizan de media casi 100.000 vuelos diarios. Esta enorme cantidad de aeronaves sobrevolando nuestras cabezas requiere de una buena organización y sincronización para evitar la colisión entre aviones, y es ahí donde juega un papel crucial este gran sector. Actualmente, existen herramientas que permiten monitorizar un vuelo desde que despega hasta que finaliza su trayecto, para ello se están intentando crear herramientas cada vez más precisas o mejorar las ya existentes, de forma que el uso del espacio aéreo esté mucho más optimizado y, además, su control sea mucho más sencillo. Uno de los principales objetivos para conseguir esto, es automatizar lo máximo posible el control del tráfico aéreo. Dicha automatización permite rebajar considerablemente la carga de trabajo de los controladores, lo que conlleva una disminución de posibles despistes o errores y una mayor capacidad de atención en otras posibles tareas que también requieran de su atención.

Este objetivo es el que da sentido al presente trabajo. Un trabajo basado en el lenguaje de programación Java, que busca la mejora de una herramienta de simulación de un controlador aéreo ya existente. Creemos que es necesario mencionar que la herramienta de la que se parte para la realización de este proyecto, pertenece al trabajo fin de grado de una antigua estudiante del Grado en Ingeniería Aeroespacial de la Universidad Politécnica de Valencia, Davinia González Morello. El título del trabajo es: Herramienta para la simulación del sistema de control de tráfico aéreo, y con él obtuvo en 2017 un tercer premio a mejor trabajo en la Conferencia de Estudiantes de PEGASUS.

## 1.1. Motivación

Son varios los motivos que me han llevado a realizar un trabajo de este tipo. Desde un primer momento, quería realizar algo que estuviera relacionado con el ámbito aeroespacial ya que siempre me ha apasionado, por lo que pensé que sería buena idea empezar a orientar mi carrera hacia ese sector con un trabajo fin de grado así. El siguiente paso fue contactar con un profesor de la rama de interés, Pedro Yuste, mi tutor. Pedro fue quien me presentó este trabajo, el cuál yo acepté casi al instante, ya que me pareció muy interesante; vi rápidamente una forma de mejorar considerablemente mi nivel en el lenguaje de programación Java, y a la vez todo un mundo nuevo de conocimiento aeroespacial.

## 1.2. Objetivos

El objetivo general de este trabajo es:

- La mejora en la automatización del perfil vertical de un simulador de vuelo

Dicho objetivo puede dividirse en dos más concretos:

- Creación de una gráfica que represente el perfil vertical de la aeronave, mostrando a tiempo real su movimiento.
- Creación de un indicador que avisa de cuándo es recomendable el despliegue de los spoilers, en caso de necesitarlos en descenso. La activación y desactivación de este avisador es automática.

Por último, como tareas necesarias para poder realizar los principales objetivos están:

- Mejorar mi conocimiento del lenguaje de programación Java.
- Aprender a usar la librería *JFreeChart* y algunas de sus clases y métodos.
- Familiarizarse y llegar a entender gran cantidad de conceptos relacionados con las aeronaves y las torres de control, además de conocer el modelo de BADA utilizado.
- Comprender el código del cual partimos, saber interpretar todas las clases y qué hace cada una de ellas.

## 1.3. Estado del arte

Algo muy similar a lo que se ha diseñado en este trabajo es el llamado VSD, que lo incorporan algunas aeronaves Boeing y Airbus. En algunos modelos son los pilotos los que deciden habilitar o no esta función. A pesar de no haber encontrado información de valor acerca de dichos softwares, podemos ver a continuación un par de ejemplos gráficos de sistemas VSD.

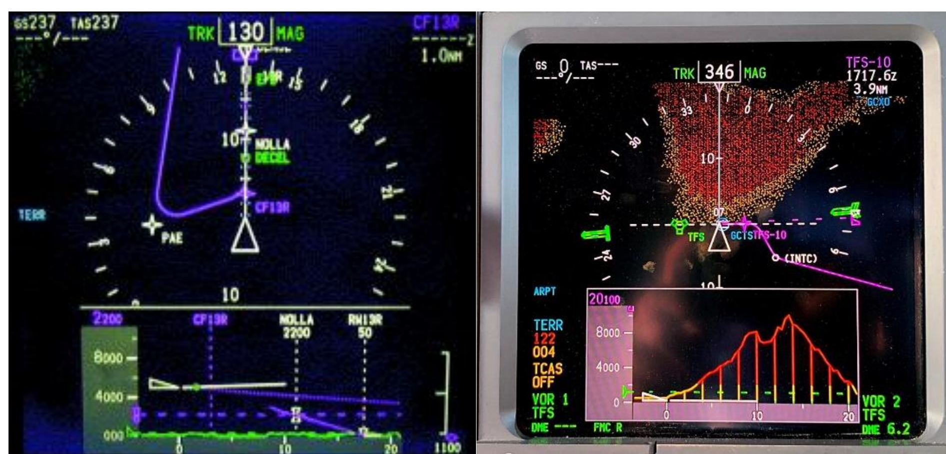


*Figura 1: Display de aviones Boeing con sistemas VSD*

Nuestro sistema de visualización del perfil vertical se ha intentado asemejar lo máximo posible al sistema VSD de la ilustración izquierda, que corresponde con el VSD de un Boeing 737. El B737 fue el primer avión donde se incorporó esta función, posteriormente se incorporó en los B600, B700, B800 y B900. Por otra parte, Airbus también incorpora esta opción, pero bajo otro nombre, en los aviones A350 y A380 (serie que ya no se fabrica).

Para hacer que nuestra gráfica se asemejara lo máximo posible a la del B737, tuvimos que tener en cuenta las proporciones de los ejes, el tamaño de la propia gráfica, la disposición de los *waypoints*, y otros aspectos que posteriormente se verán en detalle.

A parte de lo comentado sobre los sistemas VSD, con el fin de obtener una visión general sobre lo que existe actualmente en cuanto a la monitorización de vuelos, vamos a exponer un par de ejemplos de programas ya existentes. Cabe destacar que existen herramientas profesionales, como el SACTA perteneciente a Indra, usada por los propios controladores aéreos. No obstante, también hay softwares para uso público general, algunos de ellos de pago.

A continuación, vemos un par de ejemplos de *softwares* ya existentes en el mercado, para ayudar a poner en contexto la aplicación que nosotros hemos creado respecto a las que ya hay:

- **FlightRadar24:** una de las páginas sobre monitorización de vuelos más conocidas en todo el mundo. Es una página que ofrece mucha información de interés sobre los vuelos que hay a tiempo real; no obstante, mucha de la información sólo está disponible para clientes de pago.

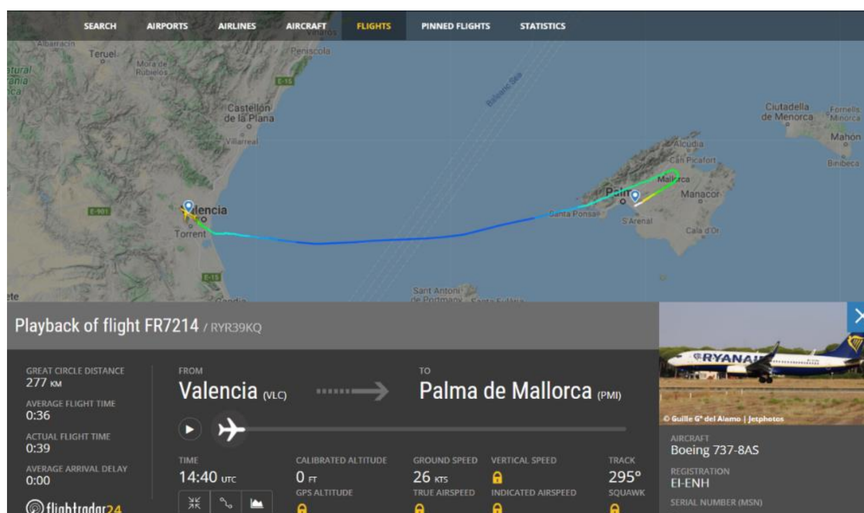

*Figura 2: Interfaz FlightRadar24 para la monitorización de un vuelo*

Como podemos observar hay información que puede ser de nuestro interés, pero también mucha de ella sólo disponible en la versión de pago, por lo que, a pesar de ser una herramienta potente, se ve bastante limitada si no se paga. Otra de las opciones que ofrece, es poder ver el perfil vertical de un avión durante su trayecto, mostrando a cada instante de tiempo los valores de altitud y velocidad.

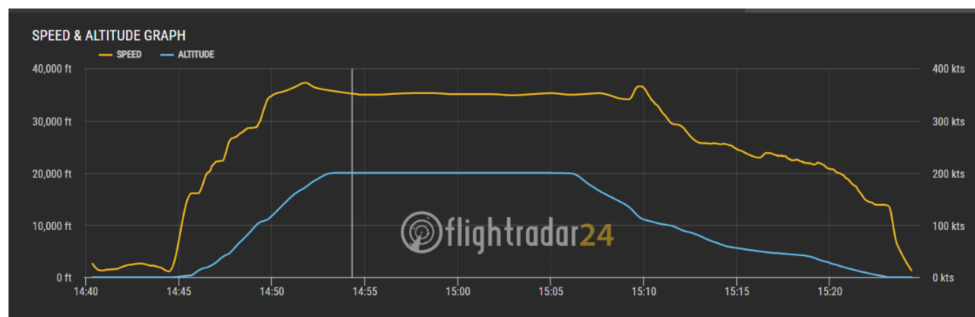

*Figura 3: gráfica del perfil vertical del avión. Velocidad y altitud respecto del tiempo.*

Como vemos es una gráfica de doble eje; uno de ellos representa la velocidad del avión respecto del tiempo (amarillo), y el otro representa la altitud, también respecto del tiempo (azul).

- **FlightAware:** tal y como se expone en la propia página de FlightAware, es la empresa líder en ofrecer información y datos avanzados y de gran precisión para tomar cualquier decisión que tenga que ver con la aviación de manera informada.

  Tanto este *software* como FlightRadar, se basan en la lectura de datos ADS-B de importantes proveedores, tales como ARINC, Garmin, Honeywell GDC, Satcom Direct, entre otros, además de más de 150 estaciones propias.

  A continuación, se comparten dos imágenes: la primera sobre la interfaz de FlightAware en el seguimiento de un avión; la siguiente, es la representación del perfil vertical de la aeronave, muy similar a la que ofrecen en FlightRadar. Puede observarse que la gráfica también es de eje doble, y que en uno de ellos se representa altitud en función del tiempo (verde) y en otro velocidad (amarillo).

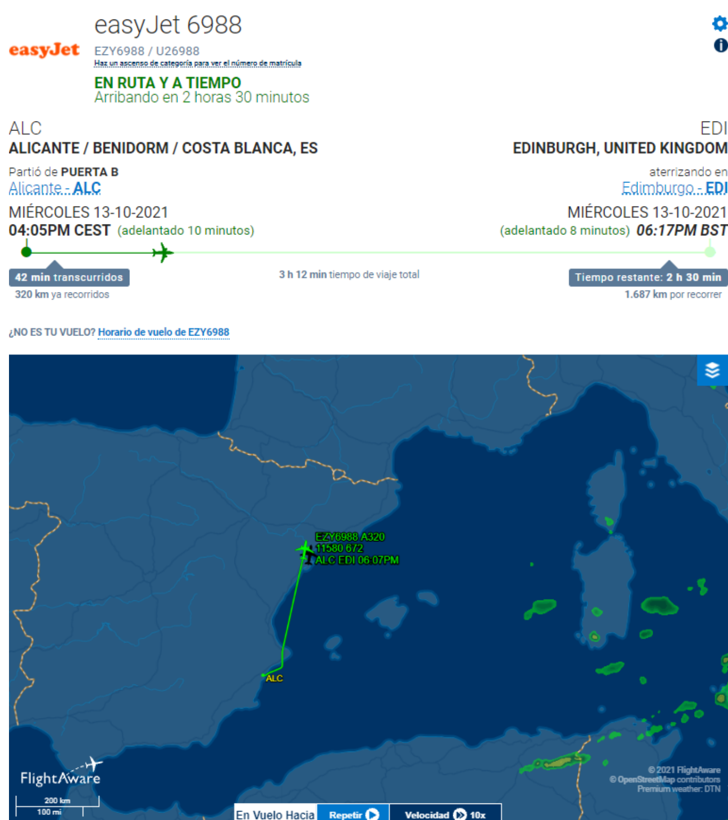


*Figura 4: Interfaz FlightAware para la monitorización de un avión a tiempo real*

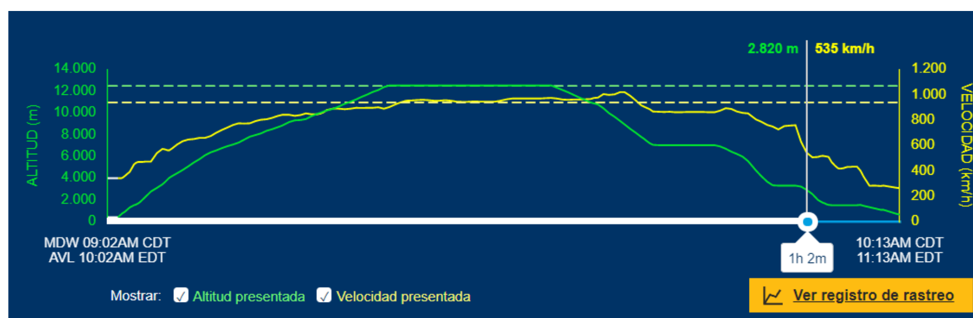


*Figura 5: Perfil vertical de un avión en FlightAware*

Tal y como hemos visto, hay herramientas muy potentes en cuanto a la monitorización de un vuelo. En cuanto al servicio que ofrecen sobre la representación del perfil vertical de cualquier aeronave, vemos que es bastante similar en ambas, las dos tienen un eje doble en función del tiempo y representan dos funciones: altitud y velocidad.

En nuestro caso, a pesar de querer representar las mismas variables, se ha optado por una gráfica de un solo eje, en el que se representa directamente la altitud en función de la distancia recorrida, asemejándose lo máximo posible a los sistemas VSD comentados al principio de este apartado. Los detalles del diseño se irán explicando con detalle a lo largo de los siguientes puntos.

## 1.4. Materiales y métodos

Para el correcto desarrollo de este proyecto, era necesario estudiar a fondo y conocer los materiales y métodos que en este apartado se detallan.

En primer lugar, la creación del programa no hubiera sido posible sin un lenguaje de programación, en este caso **Java**. Y a su vez, necesitamos también un entorno de desarrollo, que en nuestro caso ha sido **NetBeans**. Por otra parte, ha sido de vital importancia estudiar a fondo librerías importantes como es el caso de ***JFreeChart***.

**Java:** es uno de los lenguajes más utilizados a nivel mundial y está basado en la programación orientada a objetos. Este lenguaje de programación, al estar orientado a objetos se basa en 4 pilares fundamentales: la abstracción, el encapsulamiento, el polimorfismo y la herencia. Java nos permite una fácil manipulación del código y además puede usarse en cualquier sistema operativo, es por ello que en el sector aeroespacial su uso es habitual.

**NetBeans:** es un IDE (Entorno de Desarrollo Integrado) que ofrece servicios integrales para una mayor facilidad a la hora del desarrollo de *software*. Da soporte a lenguajes tales como Java, C/C++, HTML5, PHP, entre otros. También puede ser instalado en varios sistemas operativos, como Windows, Mac OS o Linux. Es una buena opción si lo que se busca es potencia y funcionalidad.

***JFreeChart*****:** está desarrollada en Java y es de código abierto. Esta librería permite crear todo tipo de gráficas de forma sencilla, desde diagramas de Gantt hasta gráficos 3D. Tiene la ventaja de tener una API bien documentada y de fácil acceso, lo que permite la búsqueda de herramientas con relativa rapidez. En nuestro caso, hacemos uso de la gráfica X-Y, ya que la representación que vamos a realizar contiene dos variables (distancia y altitud).

Nota: en cuanto a las herramientas utilizadas en el proceso de aprendizaje de Java, se han usado gran cantidad de tutoriales, por lo que en el apartado de la bibliografía hemos indicado únicamente la página de la cual todos ellos provienen.

# 2. Pliego de condiciones

En este apartado vamos a describir en detalle de qué se compone nuestro proyecto, explicaremos cuáles son los requisitos mínimos y cómo se ha procedido en su realización.

El pliego de condiciones de un proyecto, generalmente se divide en dos partes: El **pliego de condiciones generales**, donde se describe de forma general el contenido del proyecto y las características mínimas que debe tener para cumplir con los objetivos generales; y el **pliego de especificaciones técnicas**, que se divide a su vez en especificaciones de materiales y equipos, donde se definen los materiales y herramientas que se utilizan en el proyecto, y especificaciones de ejecución, en este apartado se describe cómo se ha realizado el proyecto, describiendo todas sus características y funcionalidades.

## 2.1. Pliego de condiciones generales

### 2.1.1. Condiciones de compatibilidad

Lo que desde un primer momento debíamos asegurar, es que el *software* desarrollado siguiera siendo compatible en el mismo entorno en el que se desarrolló la primera versión del mismo. Y que, en definitiva, todas las herramientas que funcionaban en la primera versión, también funcionaran en la nueva.

Tanto el proyecto original como esta nueva versión desarrollada, utilizan lenguaje Java en su totalidad; la plataforma utilizada es la *Java Standard Edition* (Java SE) versión 8, que ofrece herramientas de desarrollo *software* a través del *Java Development Kit* (JDK 8). Como entorno de desarrollo se utilizó en la primera versión NetBeans 8.1., en nuestro caso hemos utilizado NetBeans 8.2., pero las dos se distribuyen bajo la licencia *Common Development and Distribution License* (CDDL) v1.0 y la GNU General *Public License* (GPL) v2, la cual permite tanto su uso comercial como no comercial.

Para la implementación del simulador de vuelo, el primer proyecto se basó en el manual de usuario de BADA Revisión 3.9 de EUROCONTROL. Actualmente existen versiones más actualizadas, pero se ha decidido seguir con la 3.9. por los temas de compatibilidad entre versiones. El proyecto original también incorporaba para la simulación del perfil horizontal del avión, la librería Openmap versión 5.0.3., la cual hemos seguido manteniendo en este proyecto.

### 2.1.2. Condiciones de la interfaz gráfica

Este punto detalla los requisitos principales que debía tener nuestro proyecto a nivel de interfaz.

Principalmente nuestro proyecto se debía basar en la creación de una gráfica que pudiera simular el perfil vertical de un avión a tiempo real, perfeccionando además las mejoras pertinentes para un correcto comportamiento de la aeronave a la hora de cumplir con cualquier restricción. Además, debía incorporar un indicador a modo de aviso cuando el avión detectara que se debía forzar el descenso con más pendiente.

No se especificó ninguna forma de crear la gráfica. En Java, existen varias librerías que permiten crear componentes gráficos y pudimos tener la máxima libertad a la hora de elegir qué librerías usar. En cuanto al indicador *SPEED BRAKE ARMED,* tampoco se especificó qué tipo de elemento gráfico debía ser, pero si se indicó que debía asemejarse lo máximo posible a los indicadores que se dedican a ello en la vida real. En concreto se tomó como referencia el indicador disponible en los Boeing 737-6Q8/-7Q8/-8Q8.

*Figura 6: Indicador SPEED BRAKE ARMED de un Boeing 737 activado en el simulador X-Plane*

En cuanto a la posición de la gráfica y del indicador en la interfaz final de usuario, tampoco se especificó una localización en concreto, únicamente se pedía una buena integración en la interfaz por parte de los dos elementos.

### 2.1.3. Condiciones específicas del guiado del perfil vertical

En relación a lo especificado respecto a la funcionalidad de la gráfica para la monitorización del perfil vertical del avión, podemos diferenciar principalmente dos condiciones.

#### 2.1.3.1. Modelo de prestaciones BADA

El perfil vertical, al igual que el perfil horizontal ya creado, se debía basar para la simulación del movimiento y las dinámicas del avión, en el manual de usuario BADA ya mencionado anteriormente. De esta forma tanto el perfil horizontal como el vertical responderían de forma simultánea a la misma dinámica, minimizando al máximo errores a la hora de la simulación. Del manual BADA, se extraen todas las ecuaciones y todos los coeficientes necesarios para implementar el comportamiento de cada tipo de aeronave.

Para la obtención de todos los parámetros y coeficientes, hacemos uso de los archivos que componen la base de datos BADA. Estos archivos son:

- Archivo .OPF (*Operations Perfomance File*)
- Archivo .APF (*Airlines Procedures File*)
- Archivo .PTF (*Perfomance Table File*)
- ~~Archivo .PTD (*Perfomance Table Data*)~~ → *no se utiliza*
- Archivo BADA.GPF

Los cuatro primeros archivos son diferentes según el modelo de avión, mientras que el último es un archivo con parámetros globales.

La primera versión del programa no hace uso de los archivos .PTD ni .PTF. Por el contrario nosotros en nuestro programa hemos hecho uso del archivo .PTF para la modelización de las fases de ascenso y de descenso, a parte del resto de archivos que en la primera versión se utilizan. Debido a la estructura de este archivo, leerlo desde Java resulta una tarea con gran nivel de dificultad, pero a su vez, los valores de dicho archivo no difieren mucho de un modelo de avión a otro. Por este motivo se decidió obtener los datos únicamente de un archivo .PTF, en concreto del modelo de avión B738 (archivo B738_.PTF) ya que es uno de los modelos con los valores más normalizados.

#### 2.1.3.2. Restricciones SID, STAR e IAC

En la simulación, aparte de obedecer a las dinámicas descritas por BADA, el avión también debe respetar las restricciones de altitud que haya en la ruta que se le establezca al tráfico y esas restricciones ya las recogía el proyecto anterior en archivos de texto dedicados a ello.

Las restricciones se pueden encontrar en las denominadas cartas aeronáuticas, según recoge el Anexo 4 de la OACI. Dichas cartas pueden estar dedicadas a salidas instrumentales y se denominan SID, a las llegadas instrumentales, STAR y a las fases de aproximación en caso de ser instrumentales, IAC Todas estas cartas son procedimientos estandarizados por la autoridad aeronáutica competente, con el objetivo de intentar minimizar el tráfico potencialmente conflictivo a través del uso de rutas, restricciones de altitud y velocidad y puntos de control específicos.

En el programa, tanto las SIDs, como las STARs, como también los procedimientos de aproximación IAC, se recogen en archivos de texto planos dedicados a la obtención de datos de los *waypoints*. Entre ellos, se encuentran los datos de restricción de altitud, en caso de haberlos, ya que puede haber *waypoints* que no tengan restricción. Las restricciones pueden ser de altitud máxima, de altitud mínima, o de altitud determinada, y esos son los datos que en este proyecto nos han interesado.

El programa original obedecía principalmente en la fase de descenso, a una dinámica simplificada de lo que debe realizar el avión en caso de haber restricciones. Por ello, otra de las condiciones en este proyecto era remodelar la algoritmia del programa original de forma que, a parte de seguir obedeciendo al modelo de BADA, el avión modificara su dinámica para realizar el descenso cumpliendo con todas las restricciones presentes en la ruta.

## 2.2. Pliego de especificaciones técnicas

En este apartado nos centramos más en explicar qué es lo que se ha diseñado concretamente en el proyecto. Una vez claros los objetivos y requisitos mínimos, vamos a explicar con detalle todo lo que se ha modelizado y con qué herramientas.

### 2.2.1. Especificaciones de materiales y equipos

En cuanto a materiales y equipos que se han utilizado, vamos a hacer una primera diferenciación entre material o equipo *hardware* y herramientas *software*.

#### 8.1.1.1. Herramientas hardware

Para la realización de este proyecto, se ha utilizado siempre el mismo equipo *hardware*; desde un primer momento se buscó que fuera de la mejor calidad dentro de lo que nos pudiésemos permitir para tener un buen rendimiento y a la vez el menor número de complicaciones posibles en un futuro.

El elemento principal, el ordenador, es un portátil HP, modelo HP Pavilion Gaming Laptop 16-a0xxx, con procesador Intel Core i7-10750H CPU @ 2.60GHz, con 16 GB de memoria RAM. Cuenta con un sistema operativo Windows 10. Es un ordenador potente a la vez que compacto, responde con fluidez y transmite fiabilidad.

Para poder trabajar más cómodamente y de forma más eficiente, incorporamos al espacio de trabajo un segundo monitor a modo de complemento para poder visualizar varias ventanas a la vez, de tal forma que agilizara en gran medida todas las tareas. El monitor es un LG, que incorpora una pantalla de 27'', con buena calidad y sin brillo en la pantalla, para evitar los reflejos excesivos de luz.

En el espacio de trabajo también nos hemos centrado en obtener un ambiente que propicie la concentración, y de ello depende en gran medida el ruido que haya a la hora de trabajar. Por ese motivo se le da también importancia al *mouse*, un Trust compacto e inalámbrico; este *mouse* es silencioso, permite hacer "*click*" sin apenas escuchar nada. Tiene un tiempo de reacción muy bajo y es amable con la ergonomía.

Por último, el teclado es una herramienta con la que sobre todo en programación, se trabaja a todas horas, por ello le damos mucha importancia al tipo de teclado que se esté usando a la hora de trabajar. En nuestro caso, escogimos un teclado DREVO modelo Calibur V2 TE 60%. Un teclado mecánico, muy compacto y de buena calidad, que incorpora switches rojos, los más sensibles que hay actualmente. Además, es un teclado con distribución americana; nos interesaba un teclado con estas características para ser más eficientes a la hora de teclear. Un *switch* rojo no tiene dos recorridos antes de ser pulsado completamente, como sí que los tienen los *switches* marrones, además este tipo de *switches* reducen bastante el ruido. Por otra parte, con la distribución americana es mucho más cómoda la escritura de código, ya que los símbolos y elementos que normalmente se usan programando, están más al alcance, permitiendo así más rapidez a la hora de programar.

#### 8.1.1.2. Herramientas software

En cuanto a las herramientas que hemos utilizado para el diseño de nuestro programa, principalmente destacar la ya mencionada librería *JFreeChart* para la creación de la gráfica. Antes de decidirnos a empezar el diseño con esta librería, se estudió la viabilidad de usar otras herramientas, como *OpenMap*, charts4j u Openchart2.

*OpenMap* es una herramienta muy potente, pero está más orientada a graficar puntos en planos cartográficos, ya que sitúa puntos mediante latitudes y longitudes, lo que dificultaba la tarea a la hora de situar un punto dada su altura, por ejemplo. En cuanto a charts4j y Openchart2 son herramientas capaces de crear gráficos sencillos, pero no están a la altura de lo que nuestra gráfica iba a necesitar.

*JFreeChart* parecía y ha acabado siendo la opción más viable. Con esta herramienta hemos podido crear un gráfico dinámico con cantidad de elementos visuales y con un enorme nivel de personalización. Todo gracias a la excelente API, consistente y bien documentada.

La API de Java también nos ha permitido el diseño del indicador *SPEED BRAKE ARMED* gracias a la GUI, que es en lo que se basa todo el diseño gráfico del programa. La GUI es la parte del programa que permite al usuario interactuar con él. Principalmente se hace uso de componentes básicos predefinidos en el paquete Swing, en concreto para nuestro indicador se ha utilizado la clase *JLabel*.

Por último, se ha utilizado también el paquete AWT, principalmente para poder crear iconos, cambiar colores, tamaños de letras y, en definitiva, la parte que tiene que ver con la personalización de la interfaz.

## 2.2.2. Especificaciones de ejecución

En este apartado se explica de forma detallada el contenido del programa, tanto la parte del código como la parte gráfica con la que el usuario debe interaccionar.

### 2.2.2.1. Código

#### 2.2.2.1.1. Introducción a la escritura del código

Lo que hicimos desde un principio cuando llegó la hora de programar, fue escribir a partir del código del que partimos, por lo que, para diferenciar el código ya existente del nuevo, aparece al principio y al final de nuestro código lo siguiente:

```
//-----------------------------------------------------------------------
// CÓDIGO CARLOS
//-----------------------------------------------------------------------

    {contenido}

//-----------------------------------------------------------------------
// FIN CÓDIGO CARLOS
//-----------------------------------------------------------------------
```

De esta forma con escribir en el buscador del IDE alguna coincidencia con esas líneas, encontraríamos fácilmente nuestro código.

En caso de que nuestro código sean unas pocas líneas, situamos al final de cada una de ellas lo siguiente:

```
// codigo carlos
```

Si en alguna clase hemos tenido que añadir código y para ello hemos tenido que definir variables globales dentro de la misma, dichas variables no tienen esta línea de identificación, ya que se usan dentro del código que ya incorpora dichas líneas, por lo que podría resultar redundante.

Una vez aclarado cómo buscar lo que hemos programado, procedemos a describir la filosofía de diseño de nuestro código.

En cuanto a la estructura del código, pensamos que tal y como ya estaba estructurado el programa, debíamos dedicar una clase exclusivamente para la creación de la gráfica del perfil vertical. En el paquete "*application*" se sitúa la clase "*TrafficRadar*", encargada principalmente de crear el mapa del perfil horizontal con OpenMap. Nos pareció buena idea dedicar ese paquete únicamente al diseño de las gráficas, por lo que nuestra clase dedicada a ello, llamada *VerticalRoute*, se sitúa también en el paquete "*application*".

Principalmente casi todo nuestro programa está escrito en la clase *VerticalRoute*, no obstante, para la creación del indicador, para la simulación de la dinámica del avión o incluso para poder crear varias instancias de nuestra clase, se ha tenido que programar dentro de clases como *enMovimiento*, *Trafico*, *SIMU* y *Calculos*.

#### 2.2.2.1.2. Clase *VerticalRoute*

Esta clase, es la encargada de todo en cuanto a la gráfica del perfil vertical se refiere. Dentro de *VerticalRoute* tenemos los siguientes métodos:

***VerticalRoute()*:**

Método constructor de la clase *VerticalRoute*; como prácticamente cualquier constructor, se llama automáticamente cada vez que se crea un objeto de la clase que lo contiene, de forma que crea y a la vez inicializa el objeto.

En esta clase se inicializan las características generales de la apariencia de la gráfica. Se inicializan todas las gráficas XY que se van a incorporar, se crean todos los iconos que va a llevar, se establece la escala de los ejes y sus límites numéricos, se crea un título y un subtítulo, se les pone nombre a los ejes y a la leyenda y también se llama a los métodos de *VerticalRoute* que deben inicializarse desde un primer momento. Estos son *updatePlaneMovement(), loadTheoreticalRoute(), initWaypoints() y speedBrakeArmed().*

***updateChartPanel()*:**

En este método se configuran prácticamente todas las gráficas del panel de datos XY a nivel estético. Concretamente hemos creado seis gráficas, y dentro de *updateChartPanel()* se modifica el aspecto de cada una de ellas de forma independiente. Realmente se crearon siete gráficas, pero la primera fue una prueba para situar los *waypoints* en el panel y se ha mantenido en el programa por no desestructurar las otras seis restantes, pero se encuentra totalmente inhabilitada.

Cada una de las gráficas proviene de la clase *XYSeries*, perteneciente a la librería *jfree*. A cada gráfica se le ha llamado oSeries*n*, siendo *n* el número que se le asignó por orden de creación, (la clase en desuso es oSeries0).

- <u>oSeries1: Real Time Route</u>

Es la gráfica encargada de ir creando una línea por donde va pasando el avión. Todos los datos dinámicos del avión se van recalculando cada segundo. Esta gráfica crea cada segundo un nuevo punto que es justo la posición que toma el avión en cada recalculo.

- oSeries2 y oSeries3: Theoretical Route y Cruise & Descent Route

Estas dos gráficas, son las encargadas de representar la ruta que teóricamente el avión debe llevar. Se decidió dividir en dos por sencillez e independencia a la hora de realizar cálculos o modificaciones en una de las dos fases (ascenso/descenso): oSeries2 representa la ruta teórica en el ascenso, y oSeries3 representa la ruta de descenso y crucero. A pesar de ser gráficas diferentes, tienen las mismas características visuales puesto que el objetivo es simular continuidad en la ruta.

Primeramente, se creó oSeries2, para que luego desde oSeries3 fuera fácil crear la línea de crucero. Solo habría que situar un punto justo en el último punto del ascenso (oSeries2) desde oSeries3, de forma que el último punto del descenso (el punto donde empieza el descenso) se uniera al ascenso. No obstante, la leyenda de oSeries3 decidimos ocultarla, puesto que sólo queríamos que apareciese en la leyenda la ruta en tiempo real y la ruta teórica. Por ese motivo oSeries2 recibe el nombre que incluye a las dos gráficas: oSeries2 y oSeries3.

- oSeries4: Ground

Esta gráfica únicamente simula una línea a nivel FL0, haciendo de línea divisoria entre el suelo y el espacio aéreo, pero dentro de los límites en los que se mueve el avión, es decir, es una línea que únicamente tiene la longitud de la ruta que va a llevar el avión; desde el aeropuerto de salida hasta el de llegada.

- oSeries5: Indicate Altitude

Esta gráfica resulta muy útil a la hora de mejorar la claridad visual de la simulación. Mientras el avión está en vuelo, existe una función dentro del simulador que permite introducir la altura a la que queremos que se sitúe el avión. Esta gráfica es únicamente una línea horizontal que se sitúa a la altura que hemos introducido, a modo de indicador de altitud.

La línea en cuestión, aparece en los dispositivos VSD reales, y hemos procurado que tenga un aspecto similar a estos.  Suelen ser de color magenta y con líneas punteadas.

- oSeries6: Infinite Ground

Esta última gráfica, es similar a oSeries4, pero simula el suelo en toda su extensión, es decir, es una línea divisoria entre suelo y espacio aéreo, pero de longitud más allá de los propios límites de la ruta del avión.

Lo que buscábamos entre oSeries4 y oSeries6 es dividir tierra y aire, pero además diferenciar entre el suelo que forma parte de la ruta y el que no. Las dos gráficas tienen las mismas características, pero oSeries4 se ha creado con una línea más gruesa para diferenciar bien el suelo dentro de la ruta, del suelo fuera de ella.

El aspecto de cada una de estas gráficas se ha modificado gracias a la clase *XYLineAndShapeRenderer*, que proviene de la librería *jfree*. Esta clase hace de *render* y permite modificar al gusto el aspecto de cualquier gráfica XY dentro de *JFreeChart*.

En nuestro caso, la clase *XYLineAndShapeRenderer* la hemos usado para modificar el grosor de las líneas, el color y hacerlas discontinuas; para ocultar los valores de “Y” y los *shapes* que por defecto se generan en cada uno de los puntos que situamos. Además, hemos sido capaces también de ocultar la leyenda de las gráficas que no nos interesaban.

Por último, también hemos situado un par de iconos en la gráfica que simulan la forma de una torre de control, uno situado en la salida y otro al final de la ruta.

***loadTheoreticalRoute()*:**

En este método se le da forma a la mayoría de gráficas, es decir, se calculan los puntos que contiene cada una de ellas. En este método nos centramos, como su propio nombre indica, en las gráficas dedicadas a la representación de la ruta teórica: ascenso, descenso y crucero.

Los datos que se utilizan para el cálculo de los puntos de ascenso y descenso, son los que se encuentran en el archivo .PTF, como ya se ha explicado anteriormente, leer desde Java un archivo .txt con la estructura del archivo .PTF no era tarea fácil. Se intentó de diversas maneras, pero creímos conveniente optar por otra opción ya que este problema nos estaba robando demasiado tiempo del proyecto.

La opción fue coger como referencia un archivo .PTF de un modelo de avión concreto, y utilizarlo para todas las simulaciones. El archivo escogido es del modelo B738 ya que los valores de este avión encajan muy bien dentro del estándar, en la siguiente figura mostramos un fragmento del archivo.

```
BADA PERFORMANCE FILE                                        Apr 01 2010

AC/Type: B738__
                              Source OPF File:               Dec 19 2008
                              Source APF file:               Mar 05 2009

 Speeds:   CAS(LO/HI)  Mach   Mass Levels [kg]         Temperature:  ISA
 climb   - 250/300     0.78   low     -  49380
 cruise  - 250/280     0.78   nominal -  65300         Max Alt. [ft]:  41000
 descent - 250/290     0.78   high    -  78300
==============================================================================
 FL |          CRUISE           |            CLIMB                |      DESCENT
    |  TAS          fuel        |  TAS          ROCD         fuel  |  TAS  ROCD    fuel
    | [kts]       [kg/min]      | [kts]         [fpm]      [kg/min] | [kts] [fpm] [kg/min]
    |          lo    nom    hi  |          lo    nom    hi    nom   |        nom    nom
==============================================================================
  0 |                           |  157     2843  2384  2093   117.8 |  144    754   35.7
    |                           |                                   |
  5 |                           |  158     2829  2368  2076   116.8 |  145    769   35.4
    |                           |                                   |
 10 |                           |  159     2814  2353  2060   115.8 |  151    813   35.2
    |                           |                                   |
 15 |                           |  166     2917  2429  2123   115.3 |  163    782   22.4
    |                           |                                   |
 20 |                           |  167     2901  2413  2106   114.3 |  195    943   22.7
    |                           |                                   |
 30 |  230     26.5  31.6  36.8 |  190     3294  2708  2350   114.3 |  230   1001   13.5
    |                           |                                   |
 40 |  233     26.5  31.6  36.9 |  225     3769  3051  2624   115.1 |  233   1020   13.3
    |                           |                                   |
 60 |  272     31.5  35.6  39.8 |  272     4219  3235  2679   114.7 |  272   1327   12.9
    |                           |                                   |
 80 |  280     31.7  35.8  40.0 |  280     4070  3110  2564   110.7 |  280   1371   12.5
```

*Tabla 1: Fragmento del archivo B738_.PTF*

Los archivos .PTF contienen para unas determinadas alturas, en concreto de FL0 (tierra) hasta FL410 (41.000 ft), los valores de velocidad verdadera (TAS) y velocidad vertical (ROCD) tanto para la fase de ascenso como la de descenso, en todo momento para nuestros cálculos hemos cogido los datos nominales (columna “nom”). Lo que tratamos de hacer con estos datos es hallar el ángulo que se debe formar entre una altura y la siguiente, sabiendo sus velocidades.

Para ello se importaron los datos de altura (FL), velocidad (TAS) en nudos y de la velocidad vertical (ROCD en pies por minuto) para ascenso y descenso a un documento Excel; se organizaron los datos en tablas y se hicieron los cálculos necesarios para hallar los ángulos en cada FL. El cálculo es sencillo y basta con hacer trigonometría con los datos de partida teniendo en cuenta el cambio de unidades:

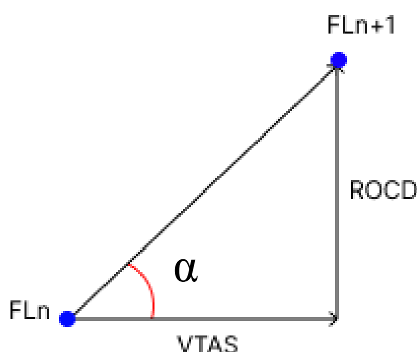


*Figura 7: Obtención del ángulo de un nivel de vuelo a otro, dadas las velocidades vertical y horizontal*

$$\alpha = arctg\left(\frac{ROCD}{VTAS}\right)$$

*Ecuación 1: Ecuación para la obtención del ángulo de ascenso/descenso*

Si hacemos este cálculo para cada una de las alturas, obtenemos la siguiente tabla de valores de ángulos en grados (columnas rojas):

| FL | CLIMB | | DESCENT | | CLIMB | | DESCENT | | CLIMB | DESCENT |
|---|---|---|---|---|---|---|---|---|---|---|
| | TAS [kts] | ROCD [fpm] | TAS [kts] | ROCD [fpm] | TAS [nmh] | ROCD [nmh] | TAS [nmh] | ROCD [nmh] | α [°] | α [°] |
| 0 | 157 | 2384 | 144 | 754 | 180.6725 | 27.09082 | 165.7123 | 8.568154 | 8.53 | 2.96 |
| 5 | 158 | 2368 | 145 | 769 | 181.8232 | 26.909 | 166.8631 | 8.738608 | 8.42 | 3.00 |
| 10 | 159 | 2353 | 151 | 813 | 182.974 | 26.73855 | 173.7678 | 9.238607 | 8.31 | 3.04 |
| 15 | 166 | 2429 | 163 | 782 | 191.0295 | 27.60218 | 187.5771 | 8.886335 | 8.22 | 2.71 |
| 20 | 167 | 2413 | 195 | 943 | 192.1803 | 27.42037 | 224.4021 | 10.71587 | 8.12 | 2.73 |
| 30 | 190 | 2708 | 230 | 1001 | 218.6482 | 30.77263 | 264.6794 | 11.37496 | 8.01 | 2.46 |
| 40 | 225 | 3051 | 233 | 1020 | 258.9255 | 34.67034 | 268.1317 | 11.59087 | 7.63 | 2.48 |
| 60 | 272 | 3235 | 272 | 1327 | 313.0122 | 36.76125 | 313.0122 | 15.0795 | 6.70 | 2.76 |
| 80 | 280 | 3110 | 280 | 1371 | 322.2184 | 35.3408 | 322.2184 | 15.5795 | 6.26 | 2.77 |
| 100 | 345 | 2977 | 334 | 1882 | 397.0191 | 33.82944 | 384.3605 | 21.3863 | 4.87 | 3.18 |
| 120 | 356 | 2805 | 344 | 1935 | 409.6777 | 31.8749 | 395.8683 | 21.98857 | 4.45 | 3.18 |
| 140 | 366 | 2628 | 354 | 1987 | 421.1855 | 29.86354 | 407.3761 | 22.57947 | 4.06 | 3.17 |
| 160 | 377 | 2443 | 365 | 2040 | 433.8441 | 27.76127 | 420.0347 | 23.18174 | 3.66 | 3.16 |
| 180 | 388 | 2252 | 376 | 2091 | 446.5026 | 25.59083 | 432.6933 | 23.76129 | 3.28 | 3.14 |
| 200 | 400 | 2055 | 387 | 2143 | 460.312 | 23.3522 | 445.3519 | 24.35219 | 2.90 | 3.13 |
| 220 | 412 | 1852 | 399 | 2193 | 474.1214 | 21.04539 | 459.1612 | 24.92037 | 2.54 | 3.11 |
| 240 | 425 | 1643 | 412 | 2243 | 489.0815 | 18.67039 | 474.1214 | 25.48855 | 2.19 | 3.08 |
| 260 | 438 | 1428 | 425 | 2291 | 504.0416 | 16.22722 | 489.0815 | 26.03401 | 1.84 | 3.05 |
| 280 | 452 | 1208 | 438 | 2338 | 520.1526 | 13.72723 | 504.0416 | 26.5681 | 1.51 | 3.02 |
| 290 | 459 | 1096 | 445 | 2360 | 528.208 | 12.45451 | 512.0971 | 26.8181 | 1.35 | 3.00 |
| 310 | 458 | 1423 | 458 | 3359 | 527.0572 | 16.1704 | 527.0572 | 38.17033 | 1.76 | 4.14 |
| 330 | 454 | 1225 | 454 | 3167 | 522.4541 | 13.92041 | 522.4541 | 35.98852 | 1.53 | 3.94 |
| 350 | 450 | 1006 | 450 | 3006 | 517.851 | 11.43178 | 517.851 | 34.15898 | 1.26 | 3.77 |
| 370 | 447 | 706 | 447 | 2653 | 514.3987 | 8.022702 | 514.3987 | 30.14763 | 0.89 | 3.35 |
| 390 | 447 | 468 | 447 | 2583 | 514.3987 | 5.318165 | 514.3987 | 29.35218 | 0.59 | 3.27 |
| 410 | 447 | 205 | 447 | 2541 | 514.3987 | 2.329538 | 514.3987 | 28.87491 | 0.26 | 3.21 |

*Tabla 2: Tabla con los valores calculados de ángulos de ascenso/descenso*

En el programa lo que hacemos es guardar los datos de las columnas FL, α *climb* y α *descent* y los guardamos en *arrays* diferentes, para poder hacer uso de esos datos cuando queramos. Las columnas de velocidades sólo nos hacían falta para el cálculo de los ángulos. No obstante, a modo aclarativo, las columnas en verde, son los datos de velocidad que hay en las columnas naranjas pero expresadas en las mismas unidades, millas náuticas por hora (nmh).

A partir de aquí, y como ya hemos mencionado con anterioridad, lo primero que hacemos es el cálculo de los datos que irán en la gráfica oSeries2, creando así la gráfica de ascenso. Posteriormente lo que hacemos es el cálculo del tramo de descenso, oSeries3. En el tramo de descenso los cálculos se van realizando desde el final hasta el principio, es decir, desde el punto de aterrizaje hasta el TOD, punto donde el avión termina la fase de crucero y empieza a descender. Una vez obtenido el TOD, creamos un último punto con coordenadas las mismas que el último punto del ascenso o TOC, de forma que con una línea se unan ascenso y descenso.

Hay ciertos casos, en los que la ruta teórica de forma natural no obedece a las restricciones, mayormente en los descensos; lo que debemos hacer en estos casos, es recalcular dicha ruta teórica para que respete dichas restricciones. La forma de proceder es la siguiente.

Hay ocasiones en las que en el descenso se produce una situación similar a la siguiente:

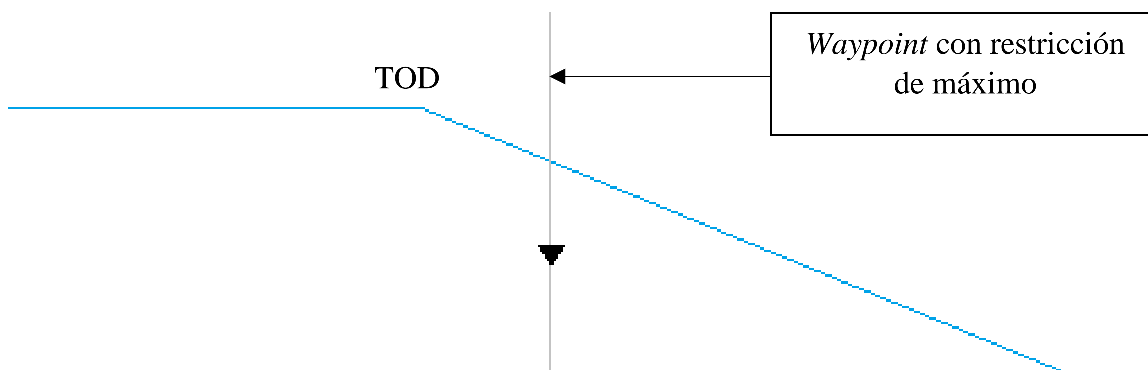


*Figura 8: Supuesto de incumplimiento de restricciones*

La ruta teórica (azul), según los datos proporcionados por BADA tiene una pendiente determinada, no obstante, puede darse el caso de que la pista del aeropuerto que hayamos elegido como pista de llegada, tenga una restricción de altitud máxima que la ruta teórica no es capaz de cumplir. En estos casos, la forma de proceder se ilustra en la siguiente figura:

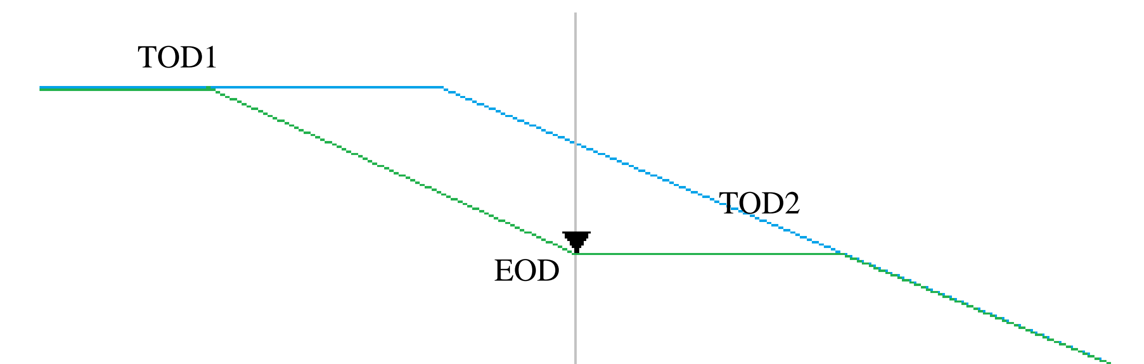


*Figura 9: Rectificación ruta teórica*

Ahora la ruta verde es la ruta teórica rectificada, que es la que el avión debe llevar. Si explicamos la situación desde el final del descenso hacia el principio, lo que hacemos es crear la ruta teórica según propone BADA hasta la altura de la restricción; ahora ese punto será el TOD2 ya que el avión debe descender dos veces. A esa altura trazamos una línea recta que hace de segunda fase de crucero hasta el punto de la restricción, el llamado EOD. Ahora el TOD1, se debe desplazar lo suficiente como para seguir respetando en el descenso los ángulos de BADA, con esto aseguramos que la ruta va a seguir con la pendiente asignada por BADA en cada tramo.

En este método se crean los algoritmos para poder realizar esta rectificación de forma genérica. Además, también se crean las gráficas oSeries4 y oSeries6, encargadas de crear el suelo de la ruta y el suelo más allá de ella.

Por último, este método también es el encargado de recoger los datos de los ángulos de descenso y guardarlos en un *ArrayList* que se usará para la configuración del sistema de aviso *SPEED BRAKE ARMED* en un método dedicado a ello.

***obtainDistance(), makeTurns()* y *updatePlaneMovement()*:**

Estos tres métodos se van a explicar en conjunto puesto que los tres están estrechamente relacionados. Tanto *obtainDistance()* como *makeTurns()* se crean para utilizarse en *updatePlaneMovement(),* que como su propio nombre indica, se encarga concretamente del movimiento del avión por la gráfica.

El método *obtainDistance()* es el responsable de almacenar en un *ArrayList*, la distancia en el eje X de forma acumulativa que va llevando el avión desde el inicio de la ruta, hasta su posición en cada segundo.

El programa original, incorpora un panel con los datos dinámicos que va tomando el avión en cada recálculo. Nosotros desde un principio sabíamos que, para el movimiento de nuestro avión, debíamos hacer uso de dichos datos de una forma o de otra, puesto que, de esa forma, conseguiríamos un buen nivel de dependencia del perfil horizontal, lo que conlleva reducir drásticamente problemas en el comportamiento del avión del perfil vertical. El panel del cuál recogemos los datos es el siguiente:

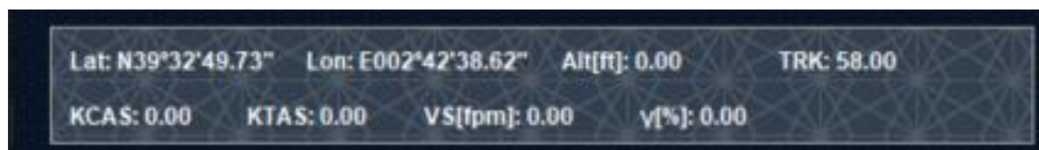


*Figura 10: Panel con datos dinámicos del avión. Fragmento de la interfaz de usuario original.*

En la primera línea de este panel vemos que tenemos la posición del avión en coordenadas geográficas (latitud y longitud), la altitud del avión en pies y el rumbo del avión en grados. Por otra parte, en la línea inferior nos encontramos con tres tipos de velocidades: la velocidad calculada KCAS, la velocidad verdadera KTAS y la velocidad vertical VS, por último, tenemos también el ángulo de cabeceo del avión, es decir, la inclinación que va llevando.

El movimiento del avión al final, no es más que el posicionamiento del mismo mediante coordenadas cartesianas. El movimiento en el eje Y se consigue mediante el dato de altitud en pies cada segundo. Por el contrario, el verdadero problema fue el movimiento en el eje X, y el encargado de calcular dicho desplazamiento es el método *obtainDistance().*

En el panel, los datos más útiles para poder describir desplazamiento horizontal son la velocidad (KTAS) y las coordenadas. En un primer momento pensamos que la velocidad sería una buena opción para hallar desplazamiento, pero a la hora de hacer el cálculo y testear su viabilidad, se observó que acumulaba cierto error, ya que no tenía en cuenta aceleraciones y se perdían muchos decimales en los cálculos. Finalmente optamos por describir el movimiento horizontal partiendo de los datos de latitud y longitud.

Lo que hicimos fue acumular en dos *ArrayLists*, los datos de latitud y longitud y posteriormente utilizarlos en un método del cual hablaremos más adelante llamado *greatCircleDistance(),* situado en la clase *Calculos*, que calcula la distancia ortodrómica entre dos puntos dadas sus coordenadas geográficas. Los datos obtenidos se acumulan en un *ArrayList* que luego el método *updatePlaneMovement()* utilizará para posicionar el avión.

El avión desde un primer momento parecía responder bien a la dinámica, pero conforme iba pasando los *waypoints* de la ruta, se retrasaba cada vez más respecto a la simulación del perfil horizontal, que es la que usábamos en todo momento de referencia. Finalmente analizando bien la problemática, nos dimos cuenta de que los *waypoints* del programa, se establecieron de forma que fuesen todos de tipo *fly-by* o puntos de recorrido de paso, es decir, no es necesario sobrevolarlos. Eso quiere decir que la distancia de un *waypoint* a otro no se corresponde con la distancia real que recorre el avión, por lo que nuestro cálculo cada vez que se llegaba a un *waypoint*, era cada vez más inexacto.

Lo que ocurre en la simulación es que el avión empieza a virar a cierta distancia antes de llegar al *waypoint* objetivo. Por tanto, en el movimiento del avión a lo largo del eje X, había que tener en cuenta esos virajes, y la dinámica que los describe se formula a través del método *makeTurns(),* que recoge los datos necesarios para realizar dichos virajes.

Una vez obtenidos todos los datos necesarios para el movimiento del avión*, updatePlaneMovement()* es el método encargado de darles funcionalidad. Utiliza todos los datos para situar el icono del avión donde corresponde cada segundo. El método detecta cuándo estamos realizando un viraje y cuándo estamos fuera de dicha acción, según en qué caso estemos, el cálculo de la posición es diferente. En caso de producirse un viraje, la dinámica diseñada es la que se muestra a continuación:

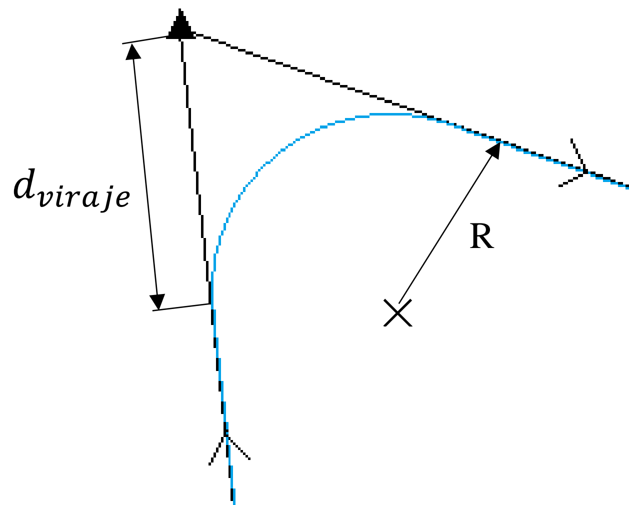


*Figura 11: Situación de viraje, visto desde arriba*

Tal y como se ilustra en la figura 11, existe una distancia a la que el avión detecta que se debe comenzar el viraje, $d_{viraje}$. La ruta azul es la que realmente toma el avión, la ruta negra es la distancia en línea recta de *waypoint* a *waypoint*. El error que comentábamos anteriormente es debido a este viraje, y para intentar reducirlo lo máximo posible, diseñamos un algoritmo que detecta en qué momento el avión sobrepasa o llega a $d_{viraje}$, y en ese instante, sitúa directamente el avión en el centro del *waypoint*. En cuanto se termina el viraje, el avión se separa del *waypoint* a una distancia $d_{viraje}$ y empieza otra vez el movimiento a partir de la distancia de viraje, pero teniendo en cuenta la diferencia entre 2*$d_{viraje}$ - $l_{círculo}$, siendo $l_{círculo}$ la longitud de la circunferencia que recorre el avión mientras realiza el viraje. De esta forma, el avión deja de acumular retardos. En la siguiente figura se expone de forma gráfica:

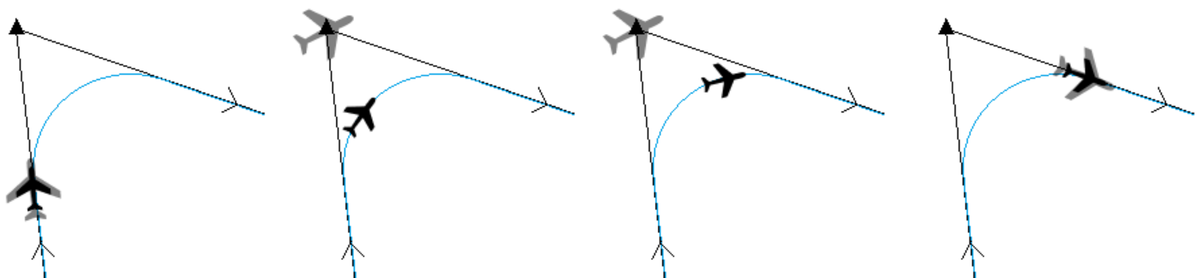

*Figura 12: Modo de actuación en los virajes*

La dinámica general es la siguiente: el avión negro y de menor tamaño simula el avión del perfil horizontal, el avión gris es el del perfil vertical. Cuando llegamos a un viraje y el sistema detecta que estamos a una distancia del *waypoint* $d_{viraje}$, el avión gris se sitúa en el waypoint, mientras que el otro va realizando el viraje y recalculando su posición cada segundo. Cuando el avión negro ha acabado de virar, el avión gris que se ha mantenido en el *waypoint* durante todo ese tiempo, se sitúa en la misma posición que el otro.

En un primer momento, se puede llegar a pensar que, una vez acabado el viraje, con sumar dos veces la distancia $d_{viraje}$, el avión se situará donde toca, pero lo que ocurre realmente es que se adelanta al avión del perfil horizontal, ya que $d_{viraje}$ no está teniendo en cuenta la distancia que ha estado recorriendo el avión del perfil horizontal. Es por ello que a la distancia 2*$d_{viraje}$ se le debe restar $l_{círculo}$.

El cálculo del posicionamiento en el eje Y cuando estamos realizando un viraje también había que tratarlo de forma diferente, ya que cuando el avión se situara en un *waypoint* al alcanzar $d_{viraje}$, también debíamos situar correctamente la altura del avión en dicho instante. El algoritmo que implementamos para ello lo que hace es predecir a qué altura estará el avión en dicho punto, teniendo en cuenta el ángulo que lleva el avión en el momento del viraje. En caso de estar en crucero y realizar un viraje, el avión no cambia su altitud. El cálculo de la altura en el viraje lo hace el método *makeTurns()*.

Si el avión no está virando, *updatePlaneMovement()* únicamente va posicionando en el eje X el avión según el cálculo de distancia que realiza el ya mencionado método *greatCircleDistance(),* y en el eje Y va tomando los datos de altitud que hay en el panel de datos. Cabe destacar, que en este método a la vez que se va simulando el movimiento del avión, también se va creando la gráfica en cargada de representar el camino del avión en tiempo real, que como ya hemos dicho anteriormente, es representada por oSeries1.

Otra de las capacidades que tiene el método *updatePlaneMovement(),* es detectar cuándo el avión está demasiado cerca del suelo con demasiada velocidad. De darse el caso, el sistema simula una colisión, deteniendo la simulación y situando una X en el lugar donde se ha producido el infortunio, dando a entender que el avión está totalmente inoperativo, además la consola también lanza un mensaje de alerta en rojo. A nivel de programación, que no de usuario, se han implementado dos opciones en cuanto al paro de la simulación, dependiendo de la prioridad e importancia que se le quiera dar a este tipo de situaciones. En caso de colisión, el programador tiene la capacidad de habilitar la opción de detener por completo todo el sistema, es decir, en caso de haber más tráficos creados, también se pararían. Esta es la opción más extrema, pero puede ser útil si se están comparando varios vuelos y uno de ellos falla. Ya que, si uno de ellos colisiona, una correcta comparación ya no sería posible. La otra

opción no tan radical, es la detención de la simulación, pero únicamente del tráfico en el que ha producido dicho accidente. Este es el caso habilitado por defecto.

***speedBrakeArmed()*:**

Este método es el responsable de la dinámica que toma el indicador creado llamado SPEED BRAKE ARMED. El algoritmo implementado en este método, se utiliza únicamente en los descensos que se hagan a lo largo de toda la ruta. En el momento que se produce un descenso, el método compara cada segundo el ángulo que lleva el avión y el que corresponde según BADA para la altura a la que se encuentra éste en dicho instante. La lógica programada considera los ángulos de BADA como los ángulos máximos que el avión debe tomar en un descenso. Es decir, en el momento en el que se sobrepase dicho ángulo, el sistema considera que es necesario forzar un descenso con más pendiente, por lo que en ese mismo instante, activamos el indicador.

Este aviso se produce de dos formas. La primera es activar el indicador cambiándolo de color a un verde similar al que en la vida real se utiliza. La segunda forma es avisando por consola, cada segundo que el indicador está activado, el sistema muestra un mensaje de aviso en rojo comunicando dicha activación; una vez desactivado, también se muestra un mensaje avisando de ello.

**rotate():**

Este método únicamente se encarga de realizar los algoritmos necesarios para rotar una imagen, en nuestro caso los iconos de tipo BufferedImage. A esta función se le debe pasar como parámetros la imagen que queremos rotar y el ángulo que queremos que se gire dicha imagen.

Desde un primer momento queríamos que el icono del avión, a la vez que se desplazara a través de los ejes X e Y, también hiciera caso al ángulo que debe de llevar en los ascensos y descensos, de forma que simulara también dicha inclinación. Ese movimiento se consigue gracias a este método. Que va tomando como ángulo de rotación el que se va recalculando cada segundo con las ecuaciones de BADA.

Por otra parte, tanto las restricciones de mínimo, máximo y restricción determinada, decidimos en la fase de diseño que podrían estar representadas por una pequeña indicación tal y como ocurre en los VSD reales. El símbolo establecido para ello es un triángulo de señalización:

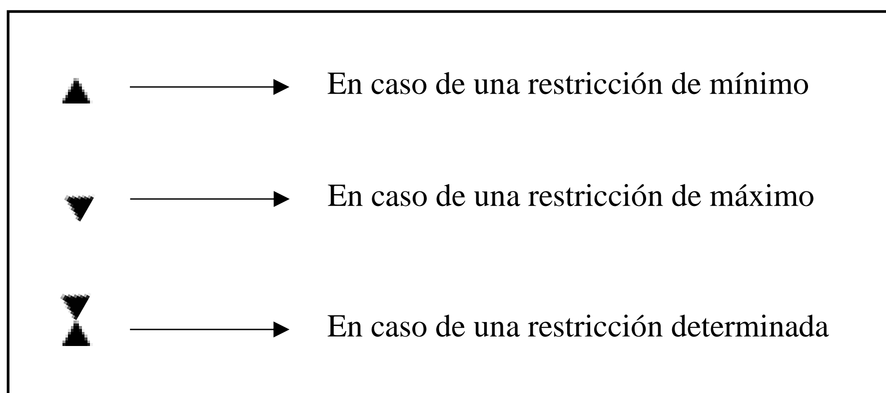


*Figura 13: Simbología para la representación de restricciones en la gráfica*

La rotación del icono del triángulo de restricción también se realiza mediante este método, así nos evitábamos tener que crear iconos de más.

***initWaypoints()*:**

Este método tiene tres funcionalidades principales muy importantes en cuanto a visualización de datos de la gráfica se refiere.

Su principal función, es representar los *waypoints* en la ruta, de forma que sepamos dónde se sitúan a lo largo de todo el trayecto. Desde un principio la idea era representar la situación de cada *waypoint* con una línea vertical, pero además este método está programado para mostrar en cada una de las líneas el nombre del *waypoint* que está representando y su restricción en caso de haber.

Por otra parte, este método también es capaz de rectificar la ruta en caso de haber alguna restricción de altitud determinada que no se cumpla. Por defecto, la ruta teórica se crea según los ángulos de BADA, y el método encargado de ellos como ya hemos descrito anteriormente, es *loadTheoreticalRoute()*. El método *initWaypoints()* detecta el tipo de restricción, y en caso de tratarse de una restricción de altitud determinada, analiza si en dicha restricción, la ruta teórica creada por defecto cumple con la restricción. En caso de no cumplirla, la ruta teórica se modifica haciendo que, en dicho punto, la ruta teórica pase por la altitud que determina la restricción. Si se modifica la altura en ese punto, los puntos siguientes hasta el siguiente *waypoint* deben situarse a la altura correcta para tener el mismo ángulo de descenso en todo ese tramo. Vemos un ejemplo gráfico a continuación:

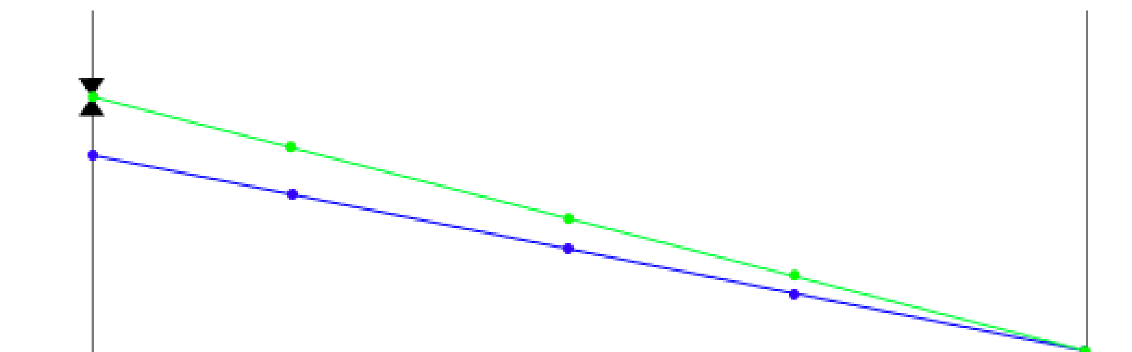

*Figura 14: Rectificación en caso de restricción de altitud determinada*

En la figura 14 la línea azul es la ruta teórica creada desde un primer momento por *loadTheoreticalRoute()*; podemos observar que no obedece a la restricción de altitud determinada en un *waypoint* concreto, por lo que el método *initWaypoints()*, es el encargado de rectificar dicha situación tal y como se ha explicado anteriormente. El resultado es el que representa la línea verde.

Por último, este método también es el encargado de situar triángulos de restricción en caso de haber restricciones. Lo que hace es analizar las restricciones que tenemos en cada *waypoint* y hace una diferenciación entre las que son máximas, mínimas y determinadas. Según el caso, como ya se ha adelantado en la figura… el programa sitúa un icono en forma de triángulo en la restricción que toque y a la altitud que marca la restricción.

Una vez diseñada toda la clase *VerticalRoute*, debíamos centrarnos en un problema que no quisimos abordar hasta que terminásemos de diseñar la clase entera, para no tener en un futro nuevos problemas conforme fuéramos añadiendo funcionalidades. El problema tenía que ver con la creación de objetos de la clase *VerticalRoute.* Hasta el momento, el programa funcionaba correctamente si sólo creábamos un tráfico, pero en cuanto creábamos más de uno, la clase *VerticalRoute* no se instanciaba correctamente. Nuestro objetivo era poder crear tantos tráficos como quisiéramos, y que en cada uno de ellos apareciese la gráfica creada en *VerticalRoute*, y que además funcionase totalmente independiente del resto de tráficos.

El principal problema era que teníamos muchas variables de tipo estático creadas. Esto quiere decir, que al crear varias instancias, se modifica el valor de la variable, pero para todas las instancias creadas. Dicho de otra manera: si una variable inicializada a 0, en la primera instancia vale 5, en el momento en el que creamos otra nueva instancia y modificamos el valor de esa variable, el valor se modifica también en la primera instancia y no desde el valor de inicialización, si no tomando como referencia el valor que ha tomado en esa primera instanciación. Por tanto, cambiamos todas las variables a no estáticas.

El otro problema tenía mucho que ver con la propia estructura del programa original, ya que hay muchas dependencias cruzadas, es decir, hay una gran cantidad de clases que dependen de otras y *VerticalRoute* estaba en medio de todo el entresijo de relaciones entre clases. Nuestro plan de acción fue en primer lugar clarificar la estructura interna del programa, localizando las clases que iban a influir en *VerticalRoute* y saber su orden de ejecución. El funcionamiento del programa se basa en la utilización de *Threads* (hilos), y cada *thread* se ejecuta en un momento diferente del programa.

Analizando en profundidad el programa, vimos que las clases que iban a influir en *VerticalRoute*, iban a ser *SIMU*, *Trafico*, y *enMovimiento*. De estas tres clases, estudiando el orden de los t*hreads*, supimos que *enMovimiento* y *SIMU* eran las que dependían de *Trafico*. Por lo que nuestra forma de proceder fue crear un método *setter* de la clase *VerticalRoute* dentro de *Trafico*, de forma que en el momento que desde *SIMU* o *enMovimiento* se requiriese de *Trafico*, se iba a poder utilizar la clase *VerticalRoute* de ese *Trafico* en concreto en el momento en el que llamáramos al *setter* creado.

#### 2.2.2.1.3. Otras clases

En algunas ocasiones, para poder realizar el diseño o los cálculos necesarios para nuestro proyecto, hemos tenido que modificar ciertas partes del código que ya estaba en otras clases, o añadir nuevo a estas.

**Clase *Calculos*:**

*Calculos* es la clase dedicada a hacer la mayoría de cálculos que el sistema debe hacer para el correcto funcionamiento del programa en general. Nosotros hemos intentado respetar lo máximo la filosofía del proyecto original, por lo que la mayoría de nuestros cálculos, también están incluidos en esta clase.

Primeramente, añadimos un par de métodos con el objetivo de cambiar unidades; uno de ellos cambia el valor de velocidad de nudos a km/s. El otro método transforma distancia de km a NM (millas náuticas). Nos dimos cuenta que este tipo de conversiones eran muy comunes a lo largo de la realización de los cálculos dentro de *VerticalRoute*, por lo que se decidió crear dos funciones que lo hicieran directamente.

El siguiente método que creamos fue el llamado *greatCircleDistance(),* dicho método sustituye a otro que ya estaba en el programa original, llamado *distanciaOrto(),* situado también en *Calculos*. Ambos tienen el objetivo de calcular la distancia ortodrómica entre dos puntos dadas sus coordenadas geográficas, pero después de analizar los resultados de *distanciaOrto()* para poder diseñar el método *initWaypoints(),* nos dimos cuenta de que los resultados eran incorrectos y las distancias no eran las reales. Lo comprobamos usando un

programa online que hace el mismo cálculo que nosotros buscábamos. Al realizar las comprobaciones pertinentes nos dimos cuenta de que los resultados no coincidían, por lo que rediseñamos el cálculo correcto con el método *greatCircleDistance().* Este método nos es de extremada utilidad en diversas ocasiones. Nos ha servido entre otras cosas, tanto para calcular la distancia entre *waypoints*, como para calcular la distancia entre el avión y los *waypoints*, por ejemplo.

El siguiente método es el llamado *intermediateGreatCirclePoint(),* este método nos devuelve la latitud y la longitud (latTOD y lonTOD) de un punto intermedio comprendido entre otros dos puntos, dadas latitud y longitud de los dos puntos iniciales y sus distancias al punto objetivo. Este método nos ha servido para el cálculo del TOD, ya que el cálculo original era erróneo y no situaba bien dicho punto, como se explicará en posteriores líneas. La situación puede verse en la siguiente figura:

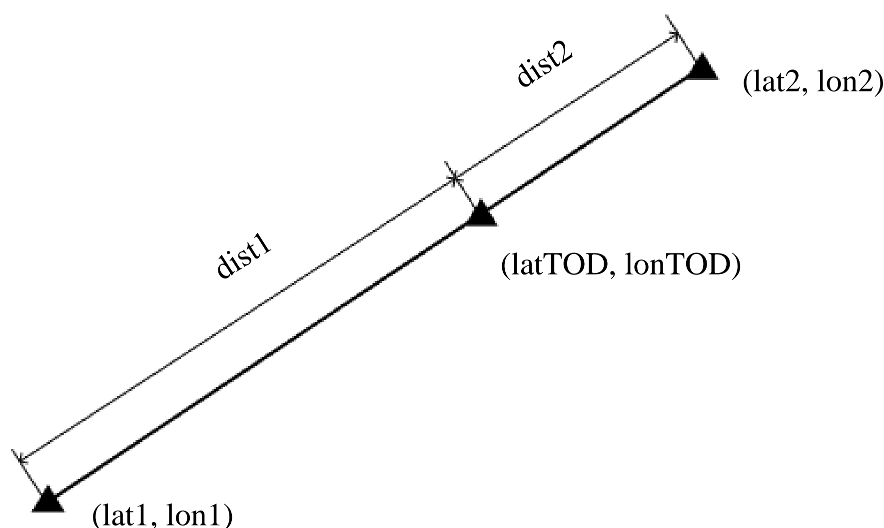


*Figura 15: Rectificación en caso de restricción de altitud determinada*

Por último, creamos tres métodos más en la clase *Calculos*, dedicados a recoger los datos de los *arrays* que obtuvimos al analizar el archivo .PTF y utilizarlos para obtener las distancias y alturas de los puntos de la ruta teórica, tanto en ascenso como en descenso. Estos datos de distancias y alturas, son utilizados posteriormente, como ya se ha explicado, por el método *loadTheoreticalRoute()* que se encarga de representarlos en la gráfica. oSeries2 representará los datos de ascenso y oSeries3 los datos de descenso.

El cálculo de estos tres métodos es el siguiente: el primero de ellos, *calculoPuntosRutaVerticalAscenso(),* lo que hace es calcular la coordenada X según los datos de ángulo y altura FL que nos proporciona BADA. Sabiendo ángulo y altura, usando trigonometría es sencillo averiguar el desplazamiento, como podemos ver en la figura 16. Los datos obtenidos se almacenan en un *array*. Lo mismo ocurre con el descenso, pero de este tramo se ocupa el segundo método, *calculoPuntosRutaVerticalDescenso().* Por último, está el método *restarFL(),* usado por los otros dos métodos. Lo que hace es recorrer el *array* con los datos de FL de BADA para restar la altura actual menos la anterior, de forma que al realizar el cálculo en los otros dos métodos, la distancia que se coge como altura es únicamente la longitud del cateto del triángulo que nos interesa, no la altura FL completa.

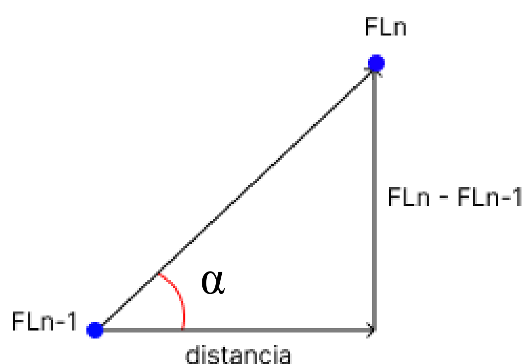


*Figura 16: Cálculo dato distancia en ascenso*

$$\text{distancia} = \frac{FL_{n+1} - FL_n}{VTAS} arctg(\frac{ROCD}{VTAS})$$

*Ecuación 2: Ecuación para la obtención de la distancia de los puntos de la ruta teórica*

El mismo proceso es el que se sigue para la obtención de los puntos de la ruta teórica en descenso.

**Clase *SIMU*:**

En primer lugar, en esta clase diseñamos el indicador, principalmente lo creamos y le asignamos una estética inicial.

Por otra parte, es en esta clase donde determinamos las dimensiones de nuestro panel y además, también habilitamos la opción de hacer *zoom in* y *zoom out* dentro de la gráfica. Por defecto las gráficas *JFreeChart*, tienen la opción de

hacer zoom creando un rectángulo con el *mouse* en el área que queremos aumentar, además, haciendo "*click"* derecho también se puede autoescalar la imagen, pero en nuestro caso, los ejes tienen una escala concreta y fija, por lo que esa opción de autoescalado deformaba por completo nuestro gráfico. Por tanto, se optó por deshabilitar todas esas funcionalidades e implantar el *zoom* mediante la rueda del *mouse*. Esta opción, permite al usuario recorrer de forma mucho más rápida y cómoda todo el gráfico, además no perdemos en ningún momento la escala de nuestros ejes.

Asimismo, en esta clase es donde creamos la recta de indicador de altitud de oSeries5. En la interfaz final de usuario existe un campo de texto dentro de los modos verticales, de tipo *JTextField*, que permite modificar la altitud a la que queremos que se sitúe el avión, este campo podemos verlo en la figura 17. Inicialmente, en dicho campo de texto hay un valor de FL determinado, siempre menor que el estipulado en el plan de vuelo. Lo que hacemos primeramente, es situar la recta que creamos en oSeries5, a la altitud que está indicando el *JTextField* en formato FL. No obstante, el sistema está programado para que en cualquier momento en el que dicho parámetro de altitud se cambie a otro valor, la recta de oSeries5 también cambie a ese mismo valor. De esa forma siempre vamos a estar indicando, tanto numérica como gráficamente, la altura a la que queremos que se sitúe el avión.

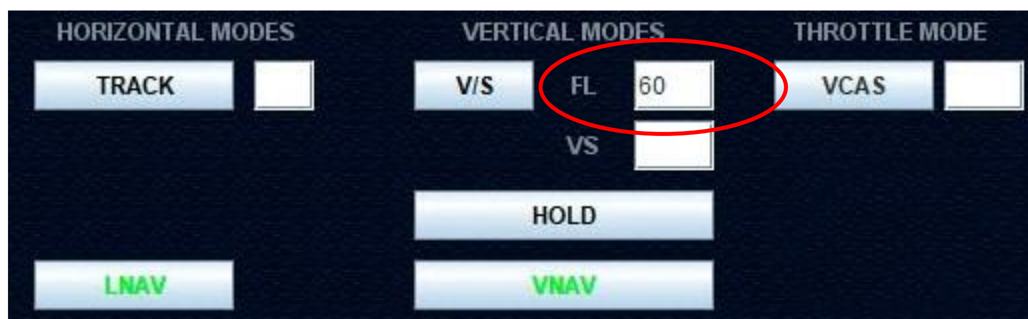


*Figura 17: JTextField dedicado a introducir el valor de altitud*

Por último, otra funcionalidad añadida que tiene *SIMU*, relacionada con los valores de altitud, es la capacidad de restringir el valor mínimo y máximo de alturas de la simulación. Es decir, si en el plan de vuelo hemos estipulado un FL200, el avión no va a poder volar a FL300, pero además tampoco va a poder volar a menos de FL0. El programa original no tenia en cuenta este tipo de límites; si se le introducía un valor de FL negativo, el avión volaría hasta dicha altitud sin tener en cuenta que es un caso imposible. Ahora el sistema tiene en cuenta este tipo de casos; cuando se le quiere indicar al avión que vuele a un valor de altitud negativo o superior al indicado en el nivel de vuelo, salta una ventana de aviso, recordando que sólo es posible introducir un valor de FL entre 0 y el valor introducido en el plan de vuelo. Cuando salta esta ventana, el valor

de FL que aparece en el *JTextField*, es el último dato correcto de altitud, por lo que la recta de oSeries5 también tomará ese valor, al igual que el propio avión. En definitiva, ni el campo de texto, ni la recta indicadora, ni el avión, cambiarán su valor de altitud a no ser que el valor que se introduzca sea correcto.

**Clase *Trafico*:**

Las modificaciones que se han realizado en *Trafico*, requerían de un alto nivel de comprensión del código original, por lo que no ha sido nada fácil modificar esta clase para un correcto funcionamiento. En *Trafico*, se han creado varios métodos, pero todos con la finalidad de poder calcular correctamente el TOD. Nosotros al crear una ruta teórica debíamos hacer que el TOD coincidiera con dicha ruta, porque desde un principio no lo hacía. Para poder comprender bien nuestra algoritmia, necesitamos primeramente contextualizar al lector de los métodos iniciales mediante los que se hacían los cálculos y cuál era la problemática de los mismos.

Inicialmente, el método encargado de obtener el *waypoint* que consideramos como TOD, es el llamado ***calculo_punto_descenso(),*** dentro de él, se utilizan varios métodos para obtener los datos que se necesitan. En concreto, se usan tres métodos: *selec_punto_inicio(), distancia_inicio_desc() y selec_punto_final()*; estos dos últimos complementan a *selec_punto_inicio()*.

- ***selec_punto_inicio()*:**

  Este método lo que hace es crear el fijo propiamente dicho. Se encarga de asignarle el nombre y también la posición geográfica (latitud y longitud). Al *waypoint* se le ha llamado “InicioDesc”. Además de ello, detecta entre qué dos puntos de la ruta debe situarse el fijo que creamos, ya que si no se le indica correctamente, las coordenadas del fijo pueden indicar una posición pero el orden en que está situado respecto al resto de *waypoints* puede no corresponder con la posición que indican las coordenadas. La problemática se representa en la siguiente figura:

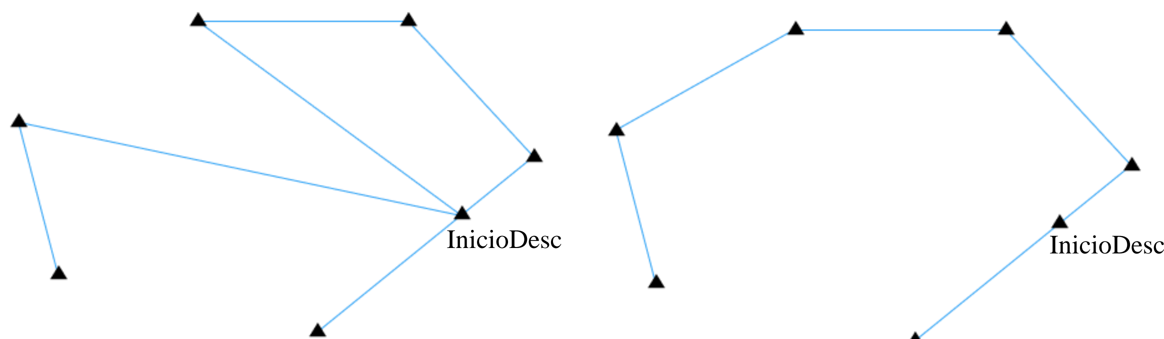


*Figura 18: Posicionamiento incorrecto y correcto del TOD*

Imaginemos que un fragmento de la ruta es el de la figura 18. En el caso de la ilustración izquierda, el *waypoint* llamado InicioDesc (TOD) tiene las coordenadas correctas, pero el orden en que se ha situado dentro del listado de *waypoints*, no es el debido. El orden que se debería llevar es el de la ilustración de la derecha. Este método también se encarga de situar el TOD entre los *waypoints* correctos, y los otros dos métodos ya mencionados complementan dicho cálculo.

- ***distancia_inicio_desc()* y *selec_punto_final()*:**

*distancia_inicio_desc()* calcula la distancia que hay entre un FAP calculado por el método *selec_punto_final(),* y la altura de crucero. Esta distancia es fija y se le pasa al método *selec_punto_inicio()* como parámetro. A partir de ahora nos referimos a dicha distancia como distanciaInicio.

Una vez obtenida distanciaInicio, *selec_punto_inicio()* la compara con la distancia que va calculando y acumulando entre *waypoint* y *waypoint*, empezando desde el FAP real de la ruta hasta el último waypoint de la lista; dicha distancia la vamos a llamar dist. En el momento en el que dist, es mayor que distanciaInicio, *selec_punto_inicio()* deja de calcular distancia entre *waypoints*, y coge los últimos dos *waypoints* que ha utilizado para hacer el cálculo, tomándolos como los puntos entre los cuales se debe situar el TOD.

Habían dos problemas principales en toda esta dinámica, ya que el TOD no se situaba correctamente en según qué casos, por tanto, procedemos a describir las soluciones que hemos ido llevando a cabo.

En primer lugar, en el método *selec_punto_inicio(),* se eliminaba dentro del listado de *waypoints* que se utiliza para el cálculo de la variable dist, el *waypoint* que coincidía con el último punto del descenso, por lo que a la hora de hacer la comparación entre dist y distanciaInicio, el resultado era erróneo, porque había un desajuste con los *waypoints* que estaba tomando para hacer el cálculo. Lo que hicimos fue eliminar la parte del código que borra el *waypoint* en cuestión, de forma que tuviésemos todos los *waypoints* disponibles para el cálculo.

Por otra parte, el cálculo en el método *selec_punto_final()* también era incorrecto, ya que no cogía el FAP real como último punto, por tanto, a la hora de hacer la comparación en el método *selec_punto_inicio(),* el cálculo nunca cuadraba, ya que para los cálculos de dist y distanciaInicio, no se tenía en cuenta el mismo punto de partida, es decir, el mismo FAP. Lo que hicimos fue modificar el método *selec_punto_final(),* para que siempre cogiera como punto final el FAP real de la ruta, de forma que el cálculo tanto de dist como de distanciaInicio, se hiciera a partir del mismo FAP.

Siguiendo con el método *selec_punto_inicio(),* el cálculo de las coordenadas era algo que también debíamos modificar, ya que inicialmente el TOD no se situaba donde marcaba nuestra ruta teórica, lo que quería decir que, a parte de los problemas ya descritos, las coordenadas tampoco eran las que tocaban, por lo que debíamos forzar cierta dependencia entre la ruta teórica y el TOD. Para ello hicieron falta dos métodos: *calculoCoordenadasTOD()* y el ya mencionado en la clase *Calculos intermediateGreatCirclerPoint().* El primero de ellos, calcula la misma ruta teórica en descenso que crea *loadTheoreticalRoute*() en la clase *VerticalRoute*. Una vez calculada detecta la posición del primer punto del descenso, el TOD, y determina la longitud y la latitud de los puntos anterior y posterior, y la distancia de estos dos *waypoints* al TOD. Introduciendo esos datos en el segundo método mencionado, *intermediateGreatCirclePoint(),* hallamos las coordenadas del TOD, que luego pasamos al método *selec_punto_inicio().*

Todos estos métodos descritos, son llamados desde el primer método mencionado, ***calculo_punto_descenso()***, que devuelve un fijo que no es más que el TOD calculado correctamente, coincidiendo ahora ya con el primer punto de la ruta teórica en descenso.

Esta dinámica, es válida en el caso de haber sólo un TOD. Pero como ya se ha comentado con anterioridad, se puede dar el caso de tener un descenso escalonado si las restricciones no se cumplen. En cuyo caso tenemos que crear un segundo TOD, llamado “InicioDesc2”. El caso se representa de forma gráfica en la figura….

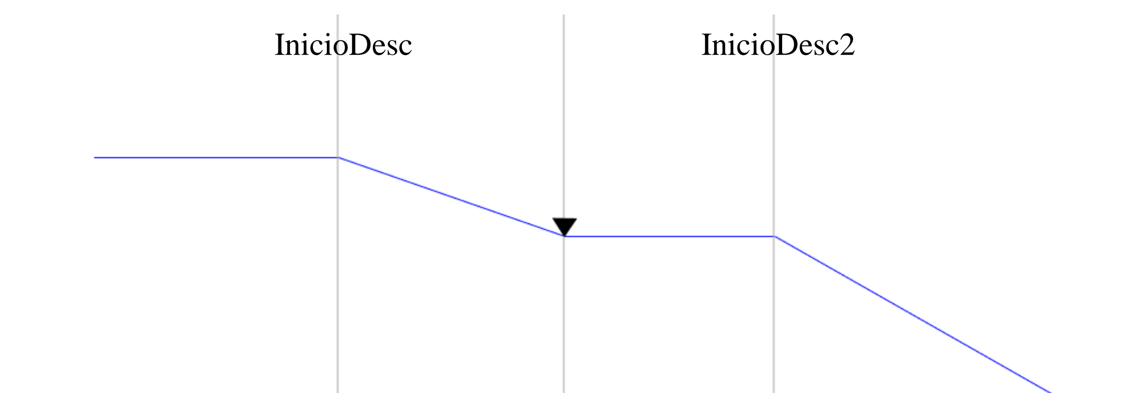


*Figura 19: Caso de descenso escalonado, con dos TODs*

Como podemos observar, el TOD2 no es más que otro *waypoint*, por lo que la forma de proceder es la misma que para el caso de haber un solo TOD. Para crear un segundo TOD, tuvimos que diseñar nuevos métodos, los mismos que se utilizan para el cálculo del primer TOD, con la misma funcionalidad, pero

ahora respecto del TOD2. Se debe detectar cuál es el punto dentro de la ruta teórica que corresponde con el TOD2, calcular sus coordenadas y situar correctamente el fijo mediante el resto de funciones ya mencionadas, de esa forma creamos un segundo punto que también depende de la ruta teórica.

En un primer momento se pensó que podríamos usar los mismos métodos sin la necesidad de crear otros nuevos. De hecho, se implementó la dinámica para realizarlo así, pero el código daba muchos problemas debido a la estructura del programa y al tipo de datos que queríamos obtener.

**Clase *enMovimiento*:**

*enMovimiento* se encarga esencialmente del movimiento del avión a lo largo de toda la simulación. Entre otras funciones, es la encargada de indicarle al avión en qué punto debe empezar a descender, y este punto debe ser el TOD. El código original obedecía al criterio de posicionamiento del TOD antiguo, pero como ya se ha explicado, dicho criterio no era válido.

En nuestro caso, el criterio utilizado ha sido decirle al programa que empiece el descenso, en cuanto el número de direcciones que ha llevado el avión sea el mismo que el índice del TOD + 1. El avión cambia de dirección cada vez que pasa un *waypoint*, ya que debe orientarse hacia el siguiente; este número se va acumulando en una variable, para que en el momento en el que dicho número sea igual al índice en el que se encuentra el *waypoint*, el avión empiece a descender. La dinámica se describe en la figura….

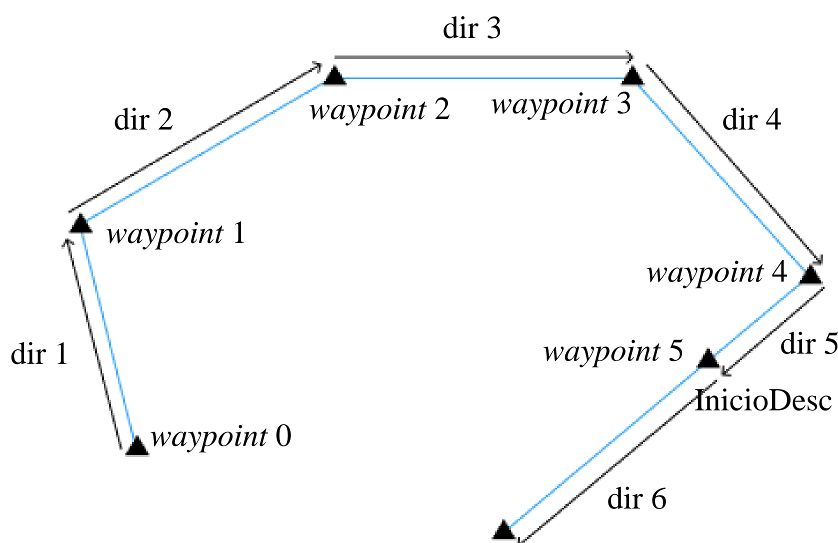


*Figura 20: Numero de direcciones e índice de cada waypoint*

La variable dir*n*, es el número de direcciones que el avión va tomando a lo largo de la ruta, por el contrario, waypoint*n*, es el número del *waypoint* en el orden en el que se sitúan estos en la ruta. En este hipotético caso, el TOD tiene un indice 5, por lo que en el momento en el que el avión, esté en la dirección 6 (TOD + 1), empezará a descender. El cambio de dirección siempre se produce nada más se sobrepasa cada *waypoint*.

Para el caso de un descenso escalonado, el programa está diseñado para que el avión descienda en el primer TOD como en el primer caso, pero que vuelva a estar en fase de crucero en cuanto llegue al siguiente *waypoint*. Seguidamente, en cuanto detecte que tenemos el siguiente TOD (llamado InicioDesc2), vuelva a la fase de descenso.

#### 2.2.2.2. Interfaz gráfica

Después de haber descrito con detalle el programa a nivel de código, procedemos a explicar y mostrar el resultado final a nivel de interfaz. Antes de proceder a describir parte por parte, en la siguiente figura se muestra el resultado general de nuestra interfaz, este es el aspecto que muestra el programa con los valores por defecto.

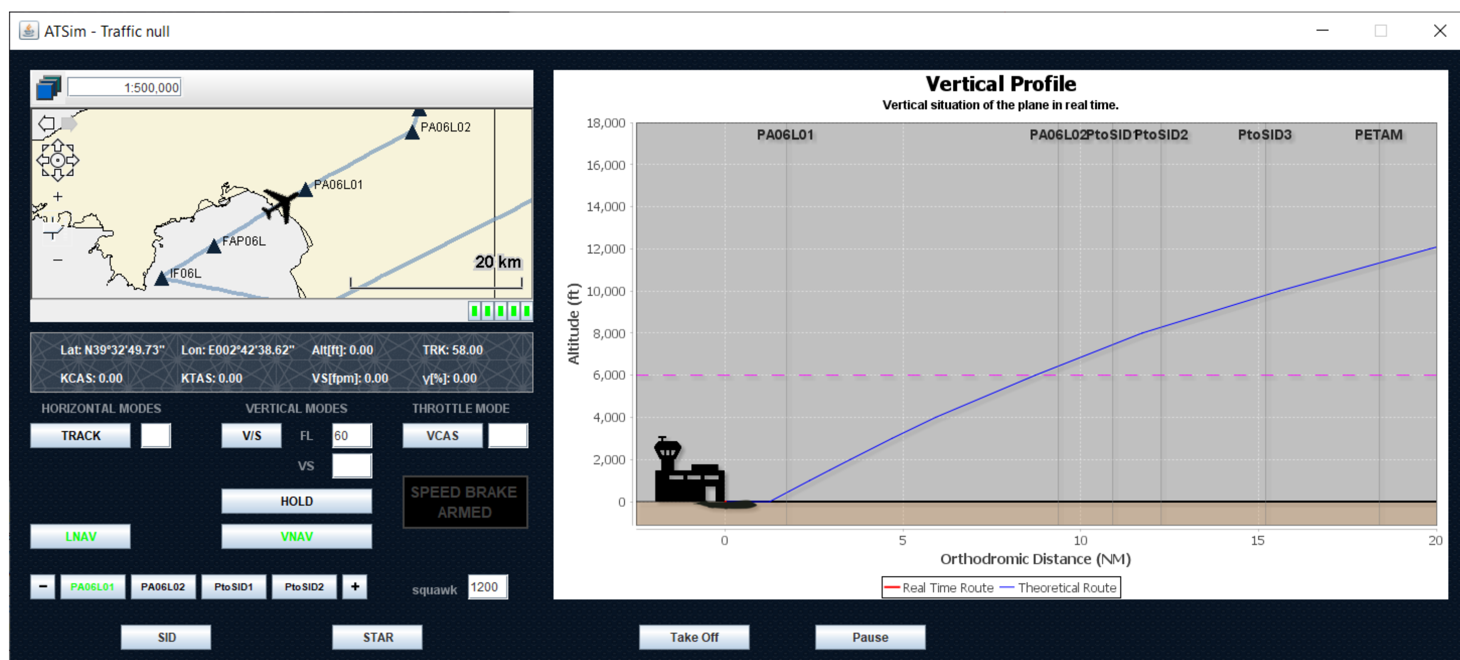


*Figura 21: Diseño final de la interfaz final de usuario*

2.2.2.2.1. Características de la interfaz

El indicador, diseñado para el aviso en caso de estar descendiendo con más ángulo de lo debido, se diseñó lo más parecido al indicador que incorporan los Boeing 737. En las siguientes figuras, se muestra el indicador *SPEED BRAKE ARMED* diseñado cuando está desactivado y en caso de estar activado.

*Figura 22: Indicador SPEED BRAKE ARMED desactivado*

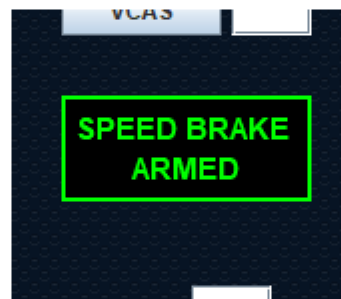


*Figura 23: Indicador SPEED BRAKE ARMED activado*

Al indicador se le introdujo un color de fondo negro, que no cambiaría en cualquier caso; por otra parte, lo que sí iba a cambiar de color, las letras y el contorno, se puso de color gris, simulando la ausencia de luz en el indicador. En caso de activarse, tanto las letras como el contorno se cambian a un color verde lo más similar al que hay en los indicadores reales.

Las medidas son las que se ilustran en la figura 24. Cabe señalar, que la proporción entre el ancho y el alto del indicador es aproximadamente la misma que la de los aviones reales, además también se intentó que hubiera la mayor semejanza en cuanto al tipo de letra y al ajuste de las mismas dentro del recuadro.

*Figura 24: Dimensiones del indicador SPEED BRAKE ARMED en pixels*

En cuanto al posicionamiento dentro de la interfaz, se ha intentado situar de forma que quede lo mejor integrado posible dentro del carácter de la interfaz.

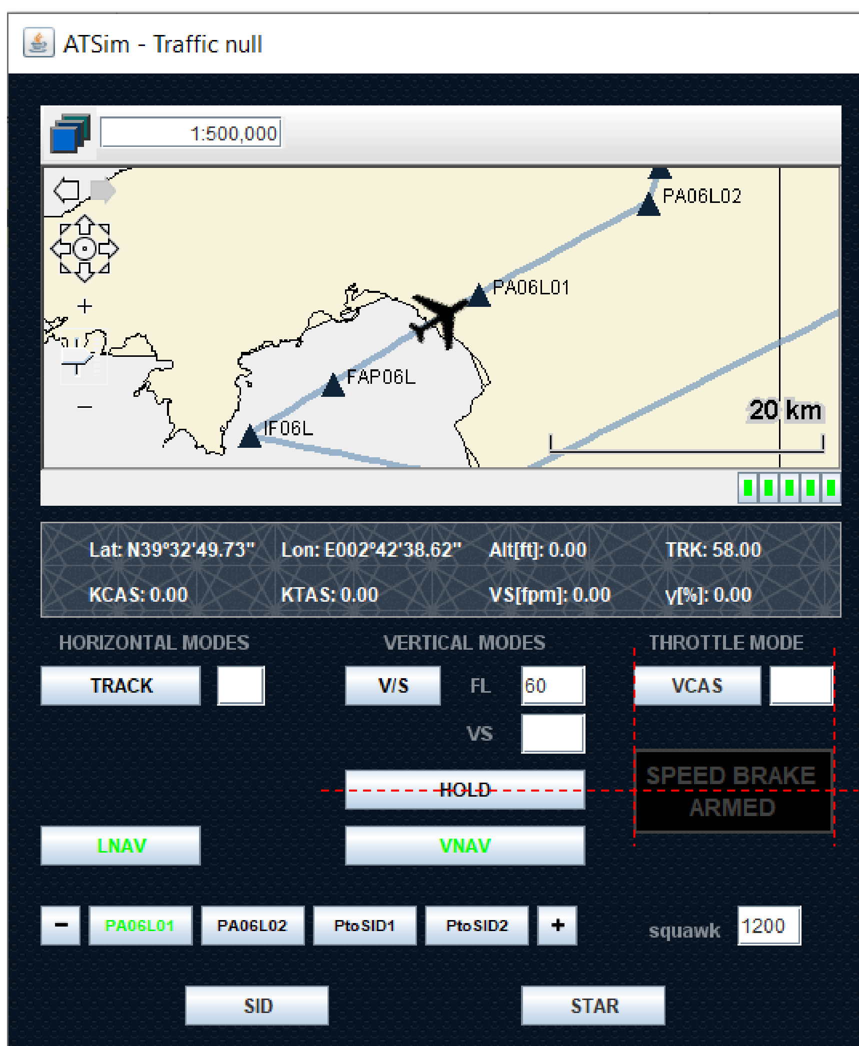


*Figura 25: Localización del indicador SPEED BRAKE ARMED*

Como podemos ver en la figura 25, el indicador está situado debajo del *THROTTLE MODE*, y a la altura del botón del modo *HOLD*, en los dos casos, de forma centrada. Se sitúa en una posición lo más simétrica posible con respecto de los elementos de su alrededor, con el objetivo de que visualmente su localización no moleste al usuario.

En cuanto a la gráfica *JFreeChart*, el diseño se describe a continuación y se muestra en las figuras 26 y 27. La primera de ellas, muestra el estado de la gráfica sin haber tocado aún nada de ella. La segunda, muestra el movimiento del avión y como se va creando a la vez la ruta en tiempo real en color rojo.

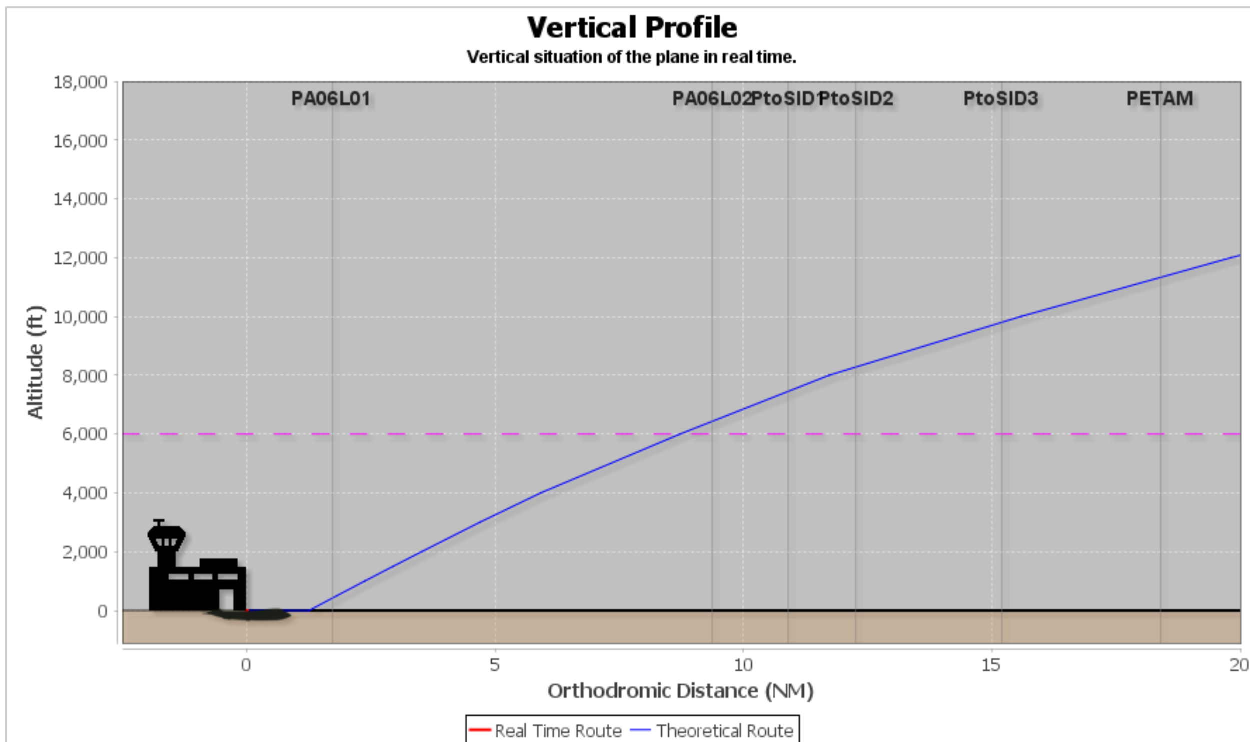


*Figura 26: Diseño de la gráfica para el control del perfil vertical del avión*

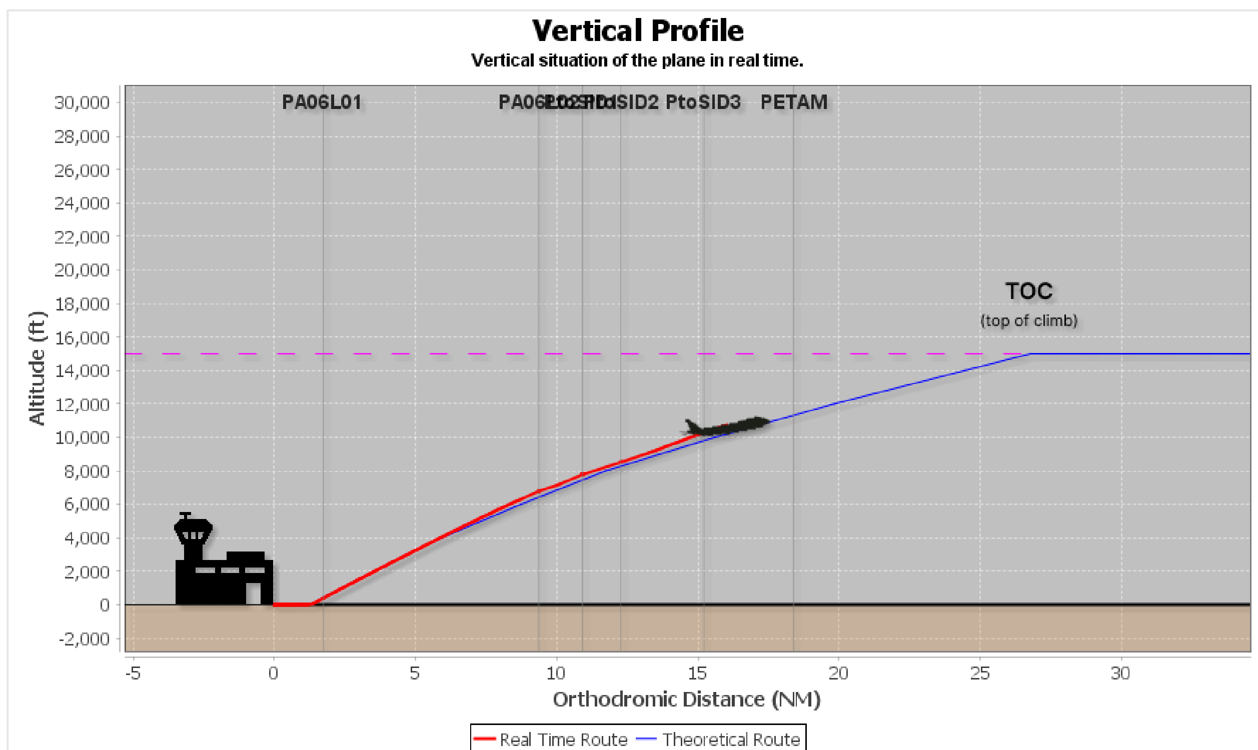


*Figura 27: Grafica del perfil vertical del avión en movimiento*

Como se puede observar, la gráfica está formada por varios elementos. En primer lugar, se le introdujo un título principal y además un subtítulo. Por otra parte, también incorpora un breve título en cada uno de los ejes, para indicar qué representa cada uno de ellos, y en qué unidades se trabaja; todo ello para mejorar el aspecto visual del trabajo. Además, la leyenda se configuró de forma que en ella sólo aparecieran los elementos principales: la ruta en tiempo real en rojo y la ruta teórica en azul. El resto de elementos que por defecto se mostraban en la leyenda, como la recta indicadora de altitud y las rectas que se sitúan en tierra, se han ocultado.

En esta gráfica nos hemos intentado basar en los sistemas VSD reales, en estos sistemas habitualmente el eje X está expresado en millas náuticas (NM) y el eje Y en pies (ft), además, también se suele diseñar este tipo de *displays* con la misma proporción entre ejes, esta proporción es 2.000 ft por cada 5 NM. Además, los números en los ejes se muestran en el caso del eje X, cada 5 NM y en el eje Y cada 2.000 ft o cada 4.000 ft. En cuanto a las proporciones del mapa, generalmente también se suele diseñar con el mismo criterio: la longitud del eje X debe ser el doble que la longitud del eje Y. Como puede verse en la figura anterior, todas estas condiciones también se dan en nuestro gráfico. Para poder compararlo con sistemas reales, a continuación mostramos tres ejemplos de sistemas VSD que cumplen con estas características.

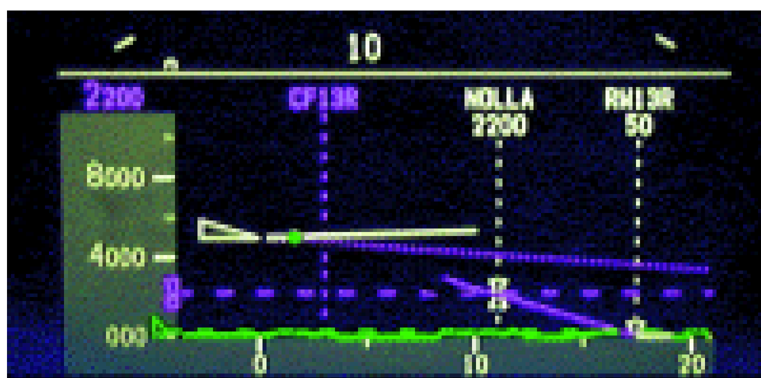


*Figura 28: Ejemplo sistema VSD 1*

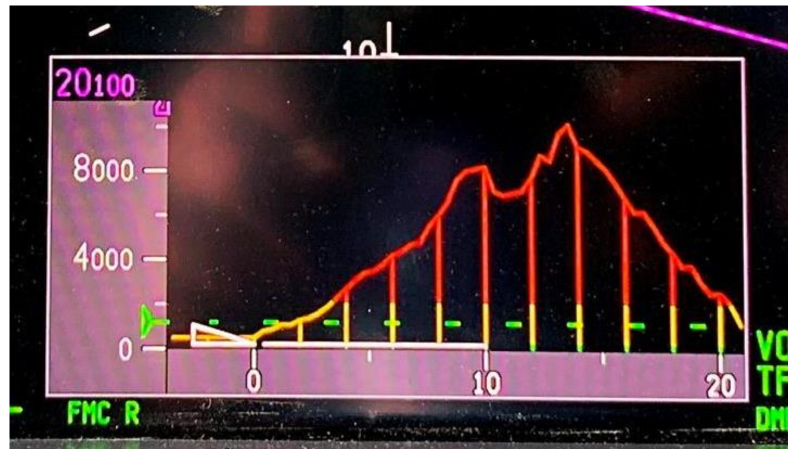


*Figura 29: Ejemplo sistema VSD 2*

*Figura 30: Ejemplo sistema VSD 3*

Las dimensiones del gráfico se pueden ver a continuación en la figura….

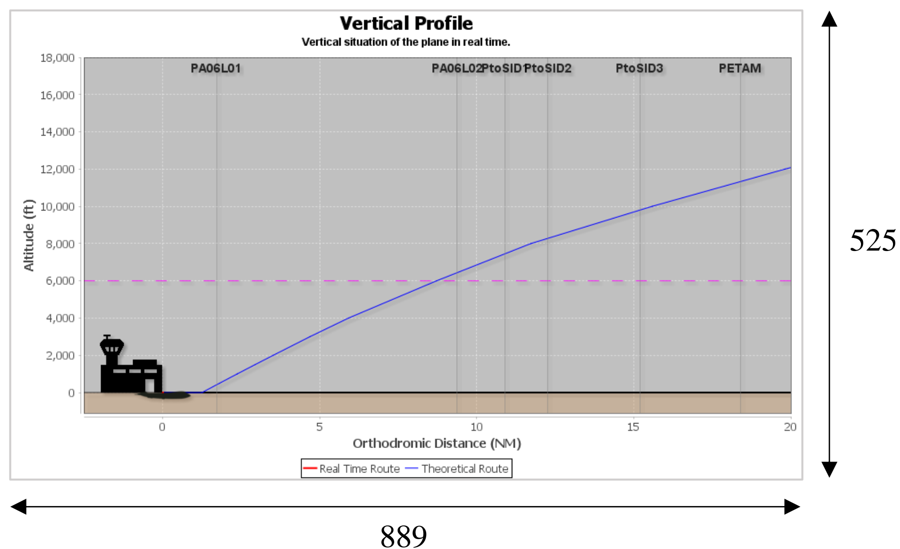


*Figura 31: Dimensiones de la gráfica JFreeChart en pixels*

En cualquier gráfico *JFreeChart*, el panel muestra todo el contenido que haya en la gráfica por defecto. En nuestro caso, al ser un gráfico tan largo, el aspecto iba a quedar totalmente deformado y apretado, por lo que decidimos programar el sistema para que inicialmente, el panel mostrara los datos dentro de un rango determinado. En cuanto al eje X, decidimos que nos mostrara únicamente los datos que hubiera hasta los 20 NM. En el caso del eje Y, el sistema limita su valor máximo inicial en 3.000 ft por encima de la altura de crucero, este criterio fue propio y en ningún caso se respetan en casos reales.

En cuanto a la localización del gráfico, se buscaba una buena integración dentro de la interfaz y un tamaño proporcional al tamaño general de la interfaz. No obstante, además se quería respetar la proporción entre los dos ejes, por lo que finalmente se decidió situar tal y como se muestra en la figura 32, para integrarlo un poco más en la interfaz general, se decidió desplazar los botones de "*Take Off*" y "*Pause*" a la altura de nuestra gráfica, de forma que además fuera más cómo visualmente utilizar esos dos botones mientras se analiza la gráfica creada, ya que están más al alcance.

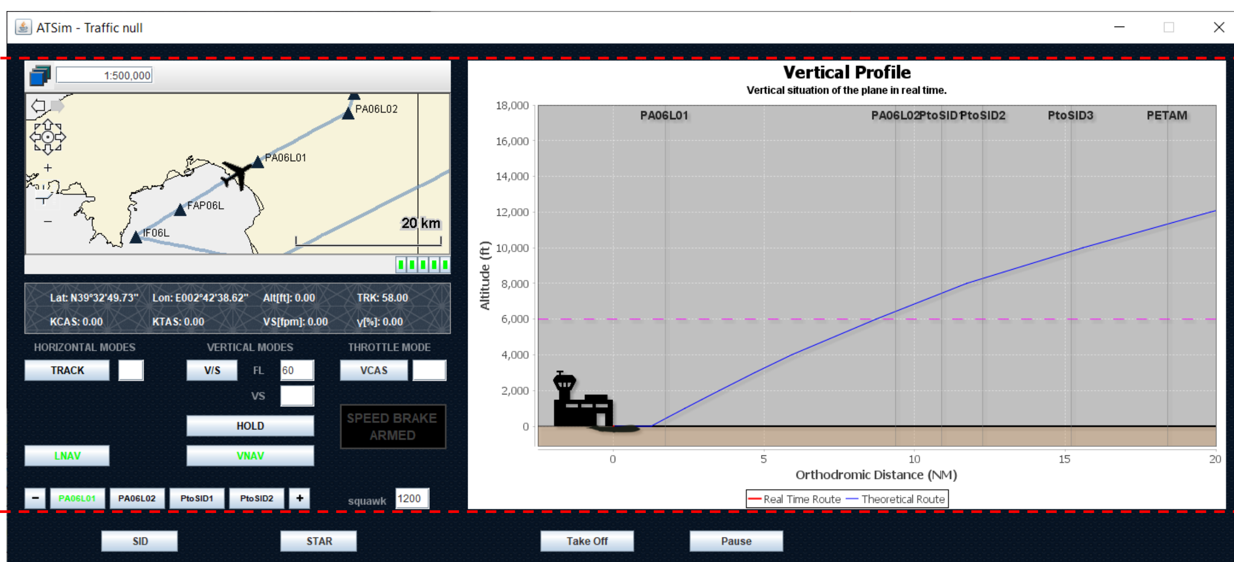


*Figura 32: Localización gráfica perfil vertical*

Centrándonos ahora en el contenido del gráfico, su diseño intenta ser lo menos cargante posible, pero a la vez con la cantidad suficiente de elementos dentro de él como para permitir una buena comprensión de la evolución de la simulación. El fondo se pensó que debía ser lo más neutro posible, de forma que desviara la atención del usuario lo menos posible. Pese a ello, sí que se diseñó un fondo de color marrón simulando tierra, para que quedara visualmente más claro los límites entre el suelo y el espacio aéreo. Se escogió un color semejante al del perfil horizontal, de forma que tampoco hubiera mucho contraste de colores.

En cuanto a las figuras que se pueden observar, para hacer visualmente más atractiva la simulación, se pensó que la silueta de un avión sería un buen sustituto de cualquier forma geométrica que quisiera hacer el papel de aeronave. Además, también aparecen las figuras de las torres de control de salida y de llegada. Visualmente también es más rápido darse cuenta de dónde empieza y acaba la ruta, si en cada uno de los extremos tenemos algo que lo señaliza.

Una de las capacidades por parte del gráfico que se ha programado, es la de poder visualizar el nombre y las restricciones de cada una de las restricciones. Por defecto, el nombre se situaba demasiado arriba de la gráfica y se veían partidos, por lo que hubo que situarlos con cierto valor de *offset*. En cuanto al tamaño de los nombres, queríamos que fuesen del tamaño suficiente como para permitir una lectura fácil y rápida. En el caso de las restricciones, decidimos que fueran de un color diferente ya que eran indicadores distintos. Además, también se les cambió ligeramente el tamaño debido a que con el mismo tamaño que los nombres, visualmente eran demasiado voluminosos. El tamaño y el color de las líneas verticales también se modificó, ya que eran más oscuras y gruesas. Por otra parte, buscábamos un equilibrio entre dichas líneas y la línea horizontal indicadora de altitud, lo que nos obligó a cambiar también el aspecto de dicha línea horizontal, haciéndola también más fina y aumentando el espacio entre las líneas discontinuas. Este tipo de detalles son los que han hecho que la gráfica tenga un aspecto limpio, espacioso y sea de fácil lectura. La ventaja de esta herramienta de visualización de restricciones, es que a la hora de hacer *zoom*, que tanto nombres como restricciones siguen situándose en todo momento en la parte superior de la gráfica, por lo que siempre van a ser visibles.

Otro recurso visual del cual disponemos, es el de las figuras que nos indican, según la ruta teórica, en qué punto llegamos al final de la fase de ascenso, el TOC, o en qué punto debemos empezar el descenso, el TOD.

Todos estos recursos se muestran en las siguientes figuras.

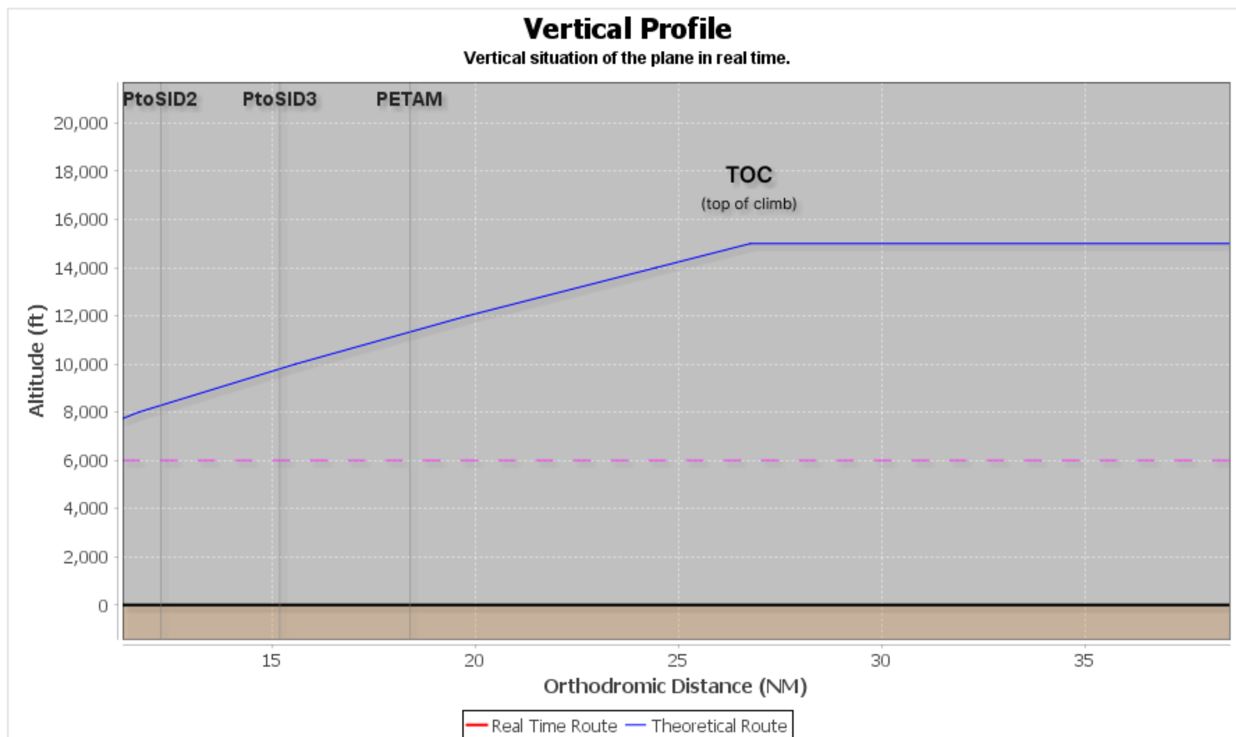


*Figura 33: Localización del TOC y visualización de las restricciones*

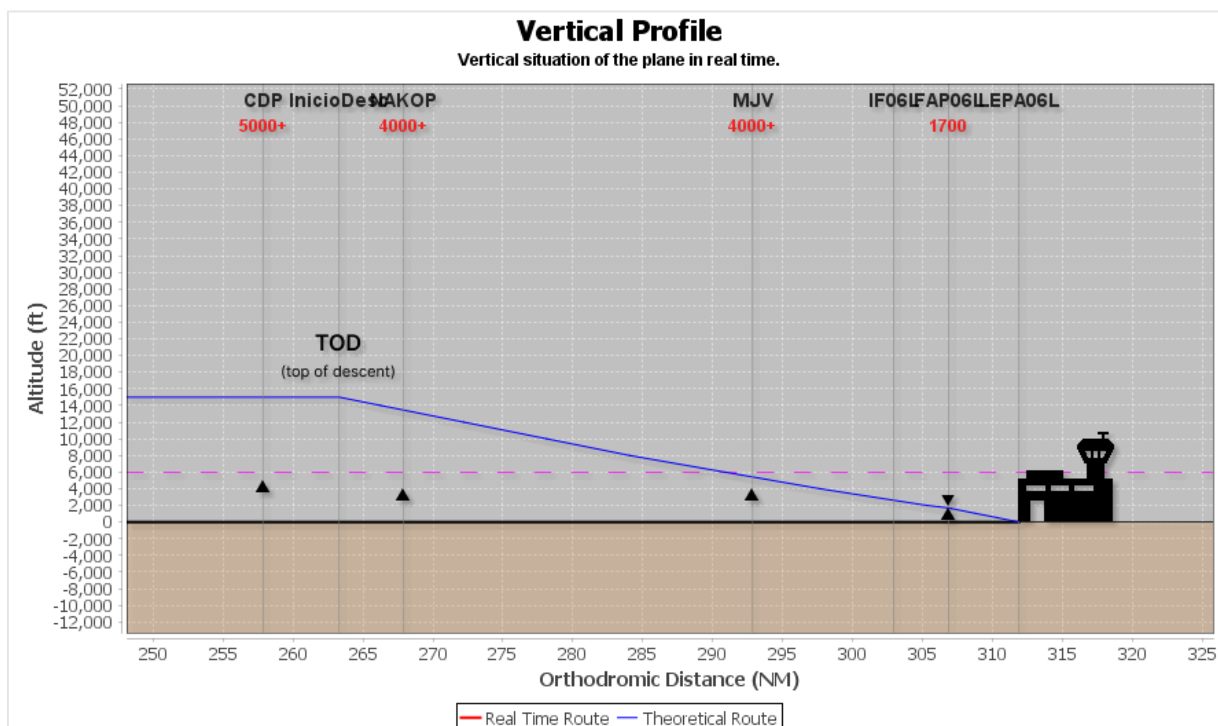


*Figura 34: Localización del TOD y visualización de las restricciones*

En el siguiente caso de descenso, podemos observar cómo aparecen varias figuras de señalización. El TOD1 es el principal, y es el que aparece siempre. No obstante, al ser un descenso escalonado, se señaliza también cuál es el punto donde se debe finalizar el primer descenso, el EOD y también el siguiente TOD (InicioDesc2). Como puede notarse, al ser puntos secundarios y que no aparecen en todos los casos, tanto el EOD como el TOD2 se representan con un tamaño ligeramente menor al TOC y al TOD1. En la figura 36 podemos ver dichas señalizaciones más de cerca.

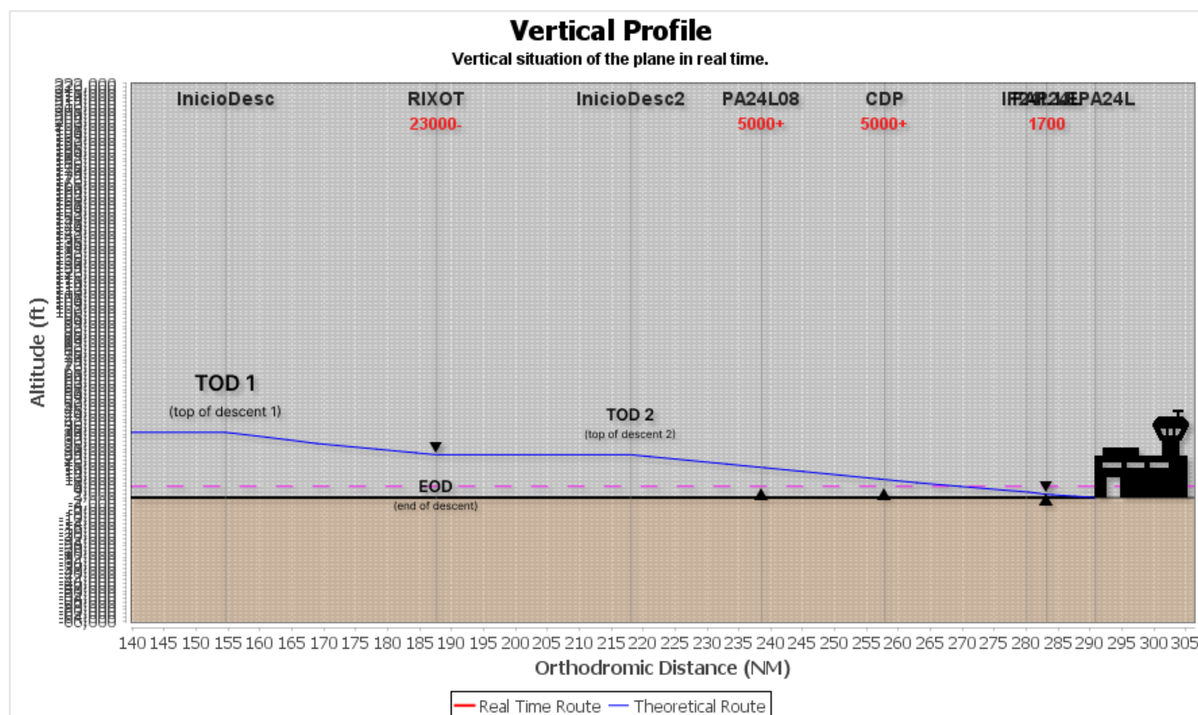


*Figura 35: Localización TODs descenso escalonado y visualización de las restricciones 1*

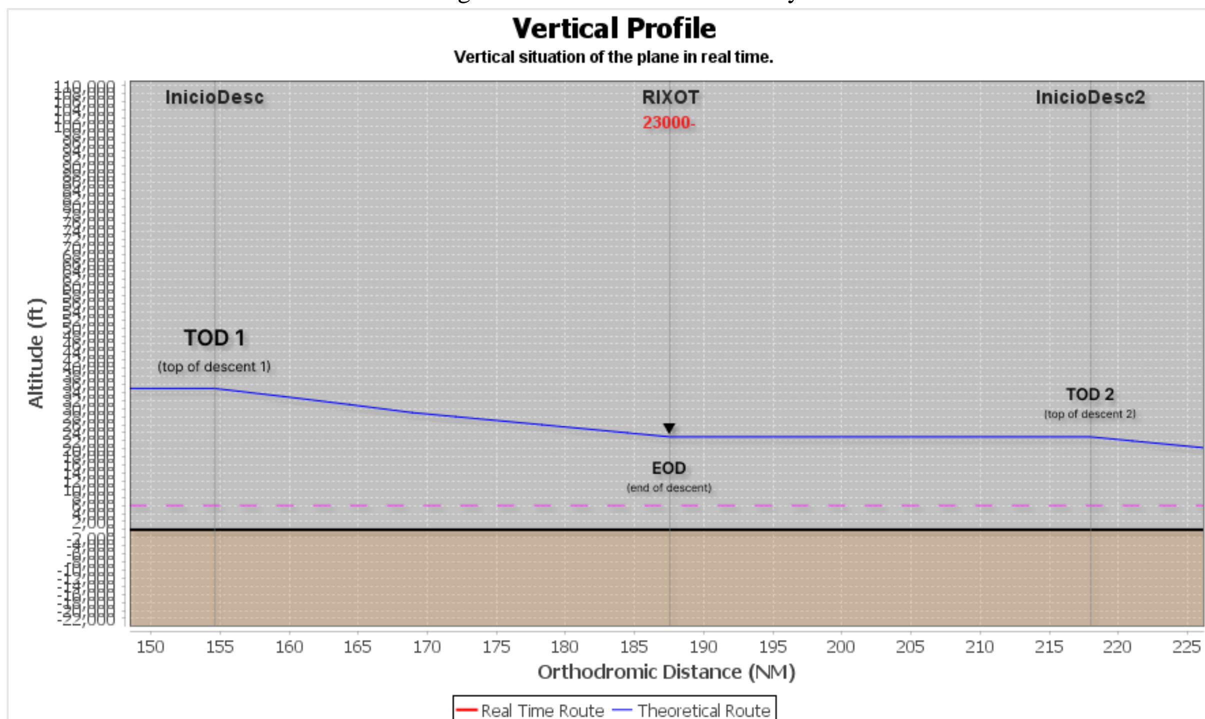


*Figura 36: Localización TODs descenso escalonado y visualización de las restricciones 2*

En los casos de las figuras 35 y 36, podemos observar cómo aparecen triángulos de restricción en el caso de haber y cómo responden perfectamente a la dinámica descrita en anteriores puntos. Estos triángulos aparecen también en los sistemas reales, no obstante, el diseño en este programa no ha respondido a ningún diseño ya creado en otros *displays* VSD. El tamaño de estos triángulos debía ser lo suficientemente grande como para ver con facilidad qué tipo de restricciones estaba marcando, pero a la vez ser lo bastante pequeño como para que no destacara mucho en comparación con el avión.

Además, con un objetivo puramente estético, se ha dotado a la gráfica de la capacidad de generar sombras en cada elemento que aparezca dentro de ella. De esta manera, la simulación puede dar una ligera sensación de profundidad muy atractiva a la vista. Por otra parte, puede diferenciarse también a nivel FL0, cómo hay creadas dos rectas, una más gruesa a lo largo de la ruta, y otra más fina que divide todo el gráfico entre tierra y espacio aéreo.

Para terminar con la descripción de la interfaz gráfica, vamos a describir la última funcionalidad implantada en el sistema. Como ya se ha explicado en anteriores apartados, el sistema se ha diseñado para tener la capacidad de limitar el espacio aéreo vertical por el que puede volar el avión. En caso de querer sobrepasar dichos límites, tanto por arriba como por debajo, el programa lanza un mensaje de aviso, obligando a introducir una nueva altitud que sea correcta. En caso contrario, el avión seguirá volando con la última altitud correcta introducida.

En la figura 37 podemos observar la ventana de aviso que le salta al usuario en caso de no introducir un valor válido de altitud en el *JTextField* dedicado a ello. En este caso, en el plan de vuelo se ha introducido una altitud de crucero de 20.000 ft (FL200), por lo que el avión va a poder volar entre 0 y 20.000 ft. En el momento en el que se le indique en el *JTextField*, que vuele a una altitud por debajo de 0 ft o superior a FL200, saltará un mensaje como el que aparece en la ilustración siguiente.

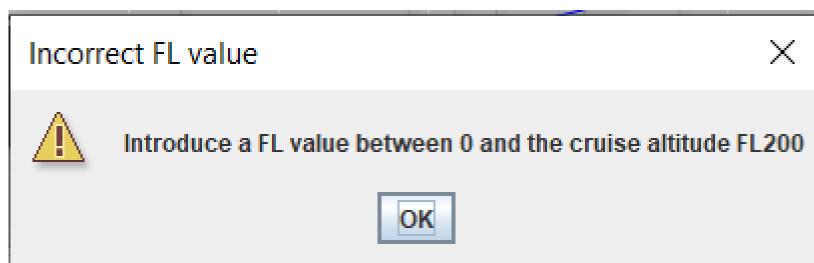


*Figura 37: Mensaje de aviso de valor FL incorrecto*

2.2.2.2.2. Evolución de la interfaz

En este punto se ha querido mostrar brevemente cuál ha sido a grandes rasgos el camino que ha ido tomando el aspecto de la interfaz, desde su estado original hasta ahora.

El estado de la interfaz desde la cual se empezó este proyecto, era el siguiente.

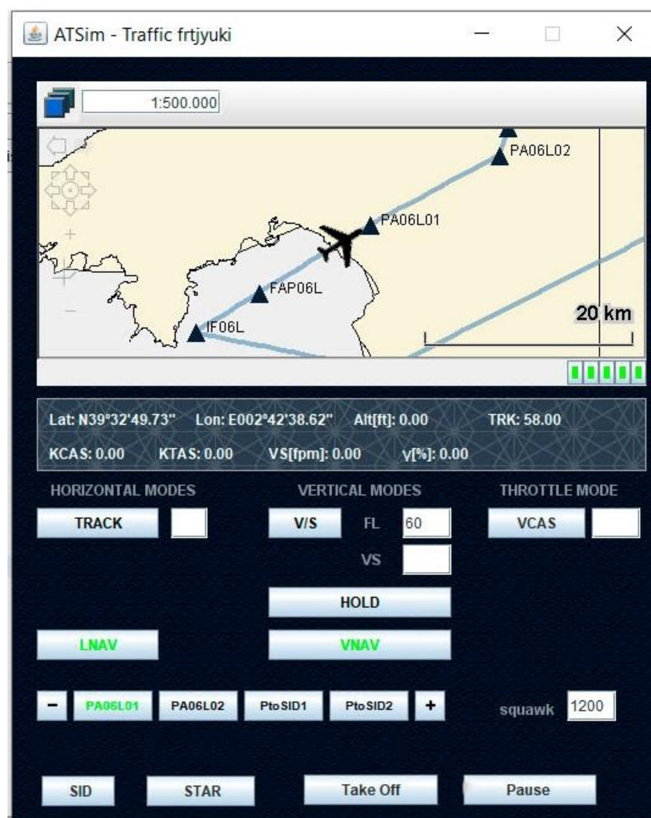


*Figura 38: Diseño original de la interfaz*

Lo primero que se modificó por motivos estéticos, fue la correcta disposición de los elementos dentro del panel de datos dinámicos. Lo único que hicimos fue alinearlos de forma que su aspecto fuera más simétrico.

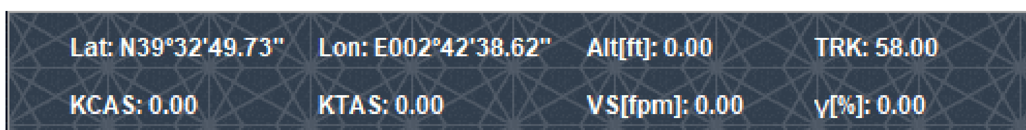


*Figura 39: Alineación de los elementos de la tabla de datos dinámicos*

Con anterioridad se ha mencionado que se intentó realizar el perfil vertical con *OpenMap*, pero acabó por no resultar. La primera idea era colocar dicho mapa justo al lado del perfil horizontal, pero estaba todo demasiado apretado y no transmitía claridad al usuario. El espacio dedicado a ello se observa en la figura 40. Se puede ver también el diseño del primer indicador que se creó. No estaba tan integrado en la interfaz debido a su tamaño, su localización y su color. Además, tampoco obedecía al diseño de ningún indicador real.

*Figura 40: Primer diseño de la interfaz en nuestro proyecto*

Uno de los primeros diseños que se crearon una vez decidido que íbamos a usar *JFreeChart* para la creación de la gráfica, fue el que se muestra en la figura 41. En este momento la gráfica tampoco obedecía a ningún estándar real y aun se estaban intentando situar correctamente los datos de la ruta de ascenso. Como podemos ver, en esta gráfica aun no habíamos modificado el aspecto de la leyenda, ni tampoco habíamos introducido subtítulo. Por otra parte, se puede observar que los datos del eje X se representaban en km.

En cuanto a los elementos dentro de la gráfica, aún se podían ver los valores de altura de cada punto creado, junto con un *shape* para localizar perfectamente dónde están situados cada uno de ellos. Creamos y situamos a modo de prueba los primeros triángulos indicadores de restricción, que posteriormente se acabaron cambiando por otros de mayor calidad. También podemos ver un primer diseño muy básico de lo que iban a ser los *waypoints* y la distancia entre ellos, la recta negra es la dedicada a ello. Por otra parte, podemos observar como el indicador ya tiene el nombre que le corresponde y su aspecto ha cambiado respecto a la primera versión, pero seguía sin tener un aspecto aceptable.

Nos podemos dar cuenta de que está todo muy desproporcionado, pero debíamos tener un primer punto de partida por el que poder empezar, además, había que familiarizarse con los elementos gráficos que ofrece Java, ya que nunca habíamos tratado con ello.

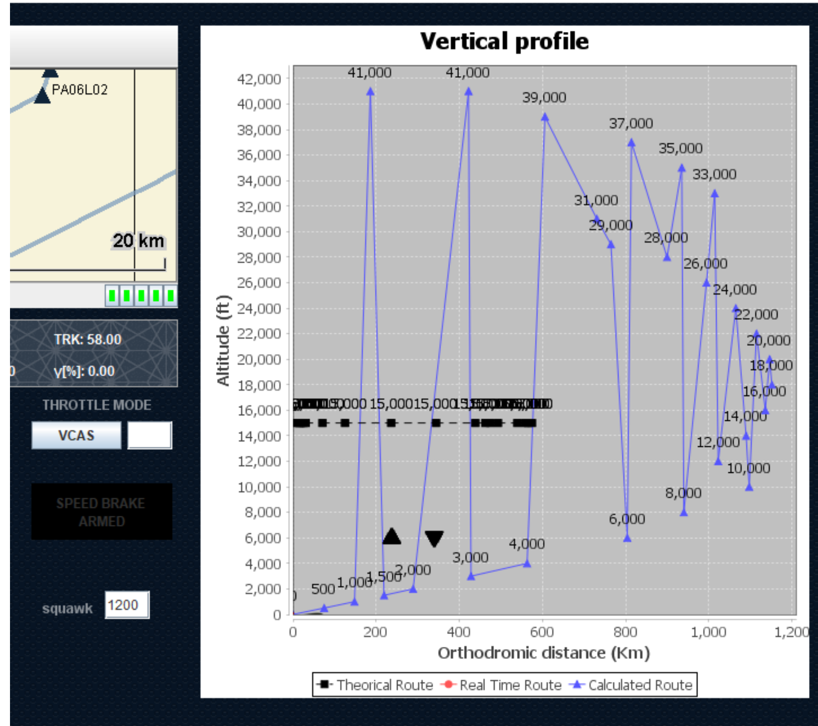


*Figura 41: Evolución del perfil vertical 1*

La siguiente imagen, muestra los primeros casos en los que dotamos de movimiento al icono del avión. En un primer momento simplemente dijimos que el movimiento obedeciera a los datos de altitud y velocidad TAS, pero en el caso del movimiento en el eje X, iba a ser más complicado, como ya se ha explicado en apartados anteriores. Además, el sistema situaba un nuevo icono del avión en la nueva posición que se le indicara, pero no borraba el anterior avión creado, por lo que también tuvimos que programar dicha dinámica para que sólo se viera el último icono del avión, actualizando la simulación para que borrara el inmediatamente anterior.

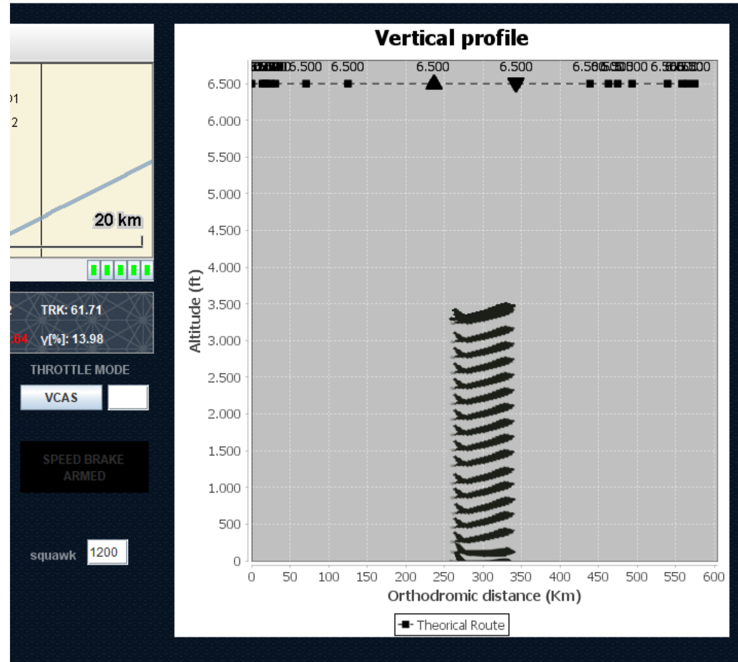


*Figura 42: Evolución del perfil vertical 2*

Con el tiempo acabamos dándole la forma que más o menos tiene ahora, añadiendo y suprimiendo elementos o aspectos que han hecho posible el resultado final. El proceso de dicha evolución se muestra en las figuras siguientes. A modo de comentario, decir que la capacidad de hacer *zoom* con la rueda del *mouse*, no fue posible hasta bien avanzado el aspecto de la interfaz. En ningún caso se implantó en las primeras versiones del diseño.

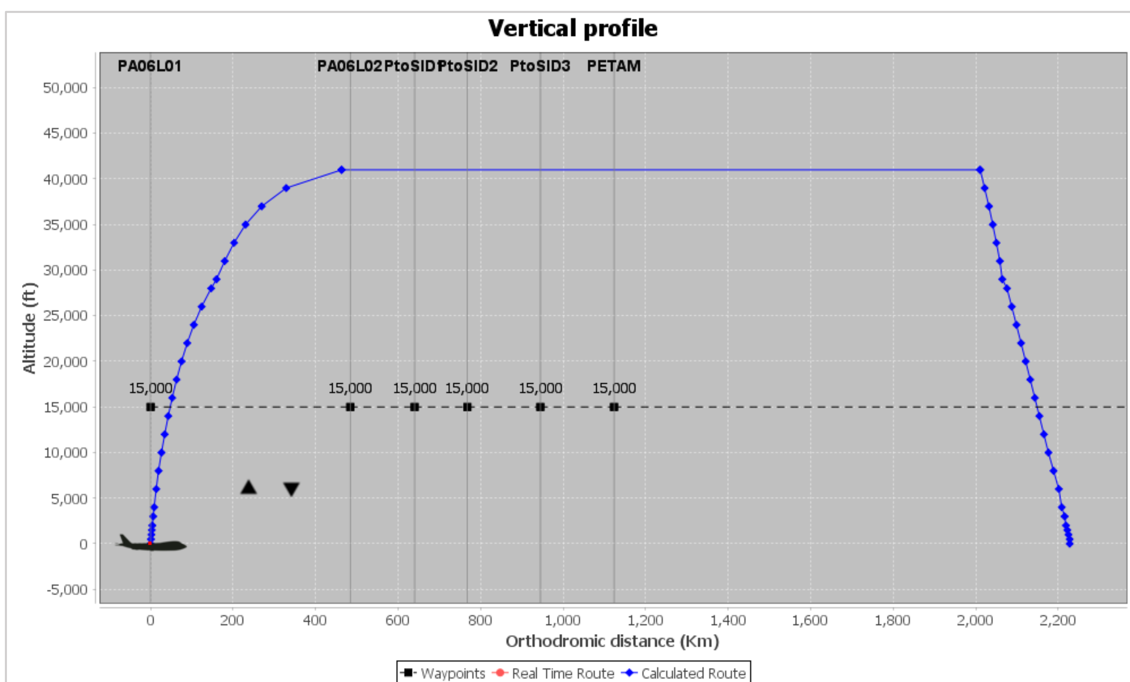


*Figura 43: Evolución del perfil vertical 3*

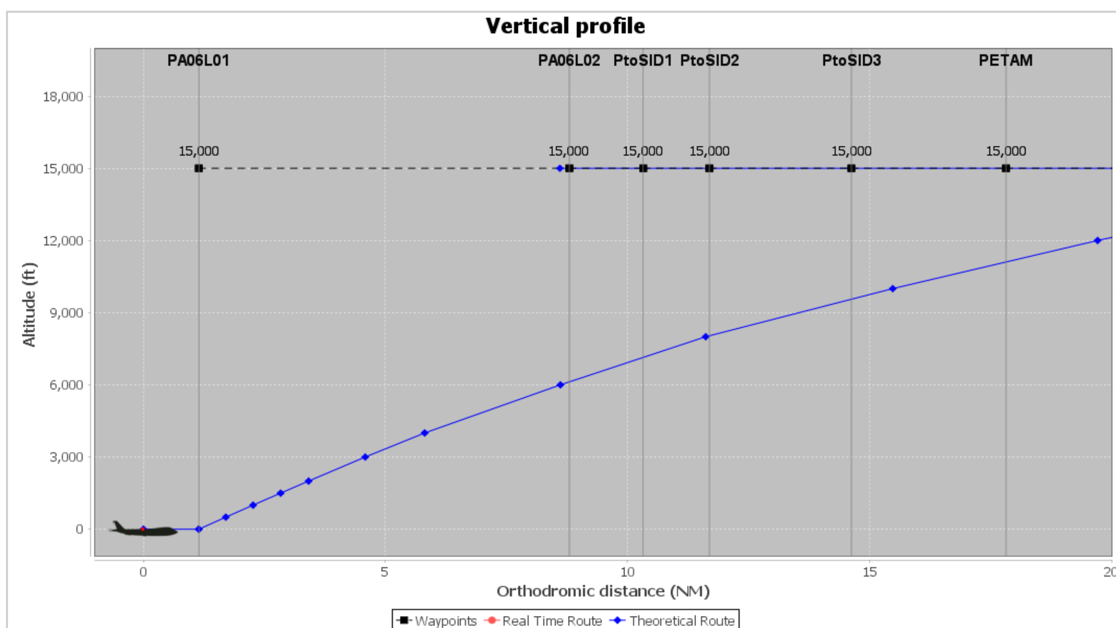


*Figura 44: Evolución del perfil vertical 4*

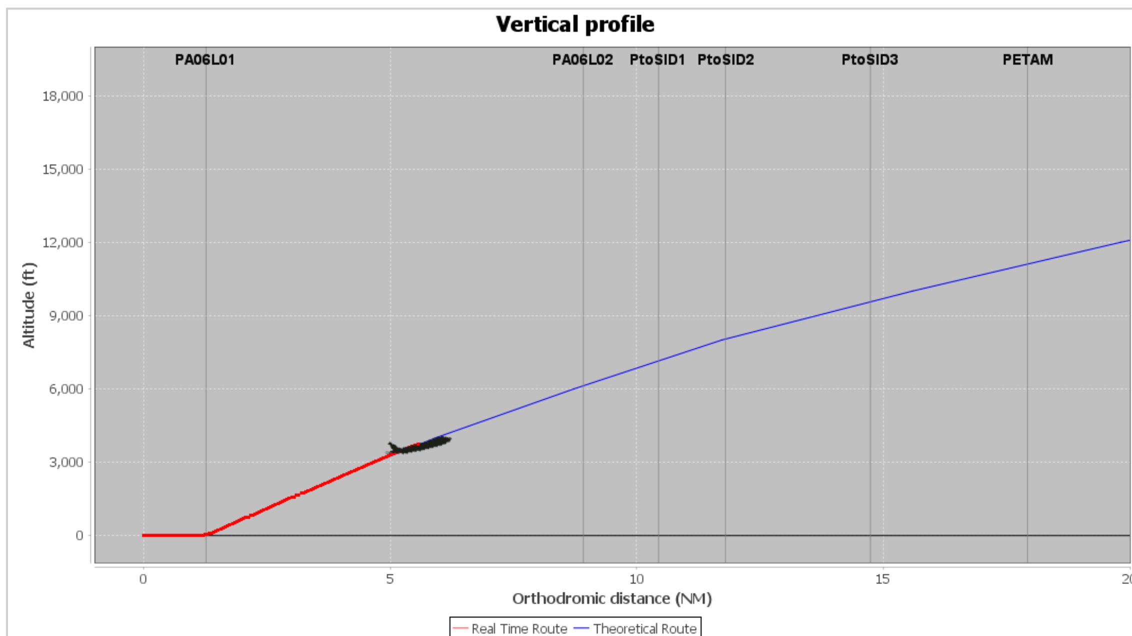


*Figura 45: Evolución del perfil vertical 5*

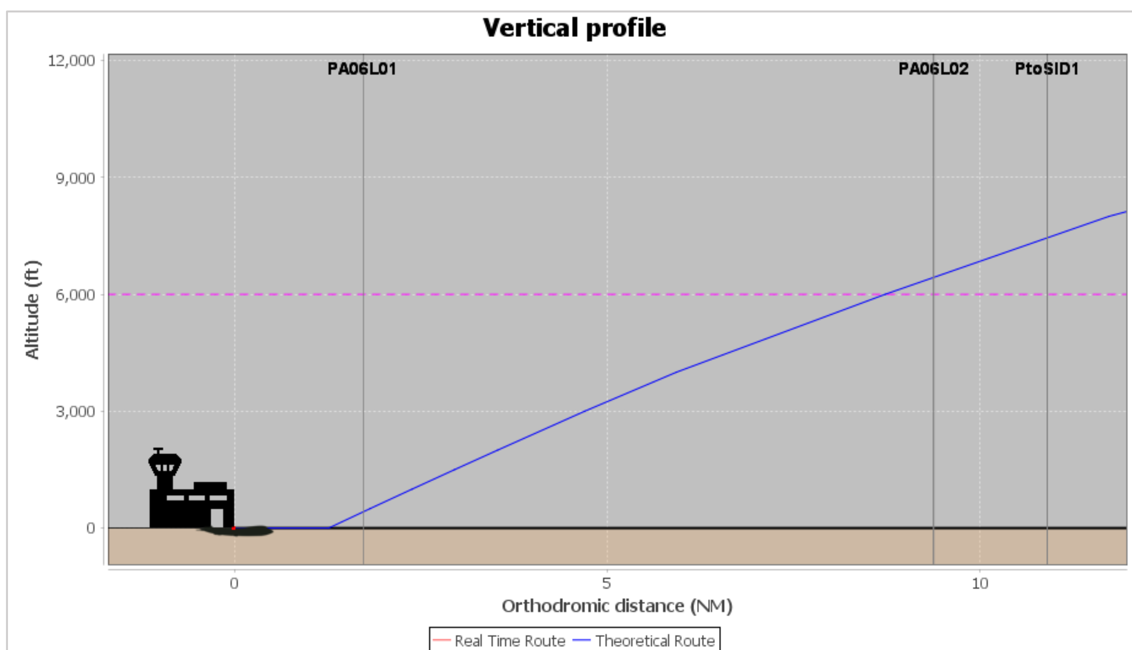


*Figura 46: Evolución del perfil vertical 6*

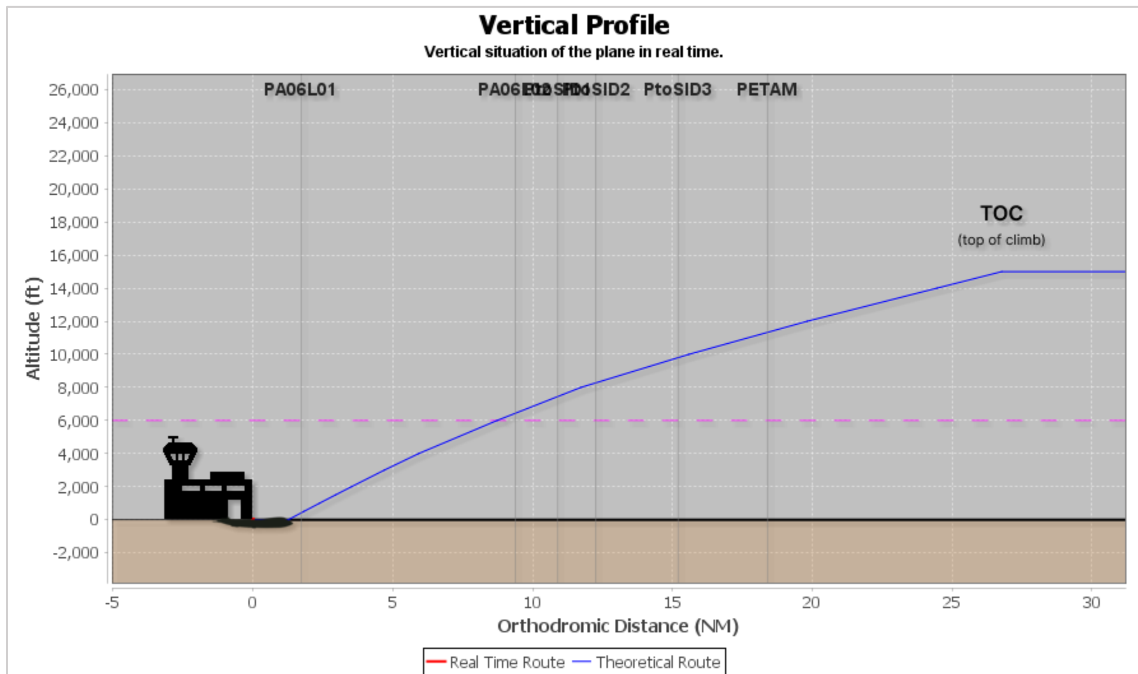


*Figura 47: Evolución del perfil vertical 7 – estado final*

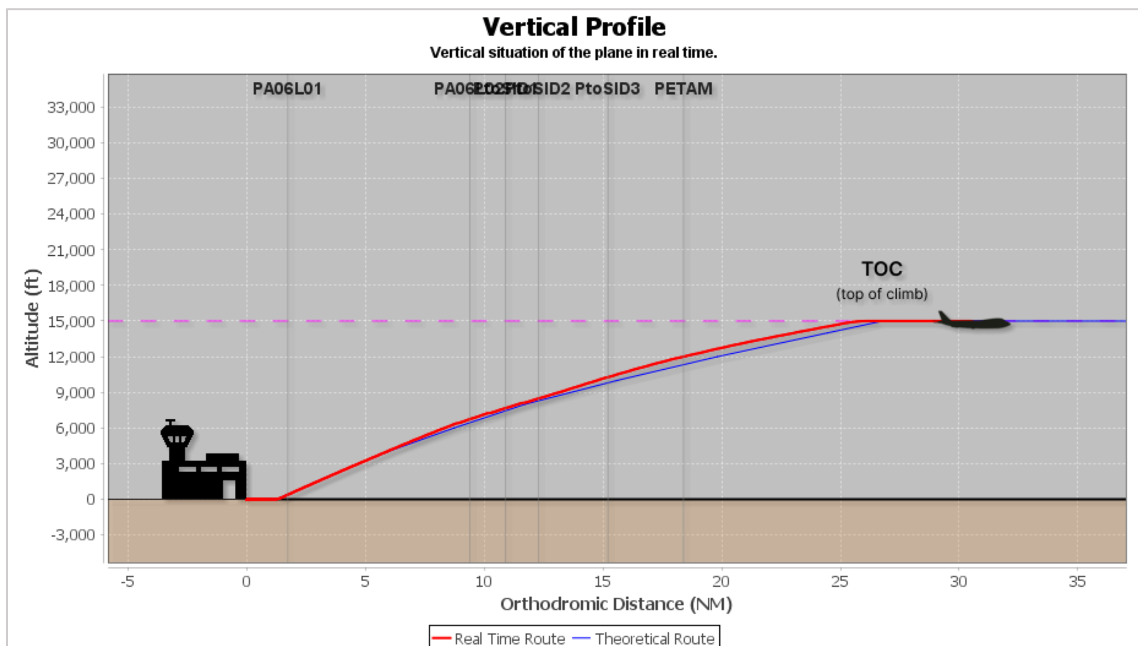


*Figura 48: Evolución del perfil vertical 8 – estado final*

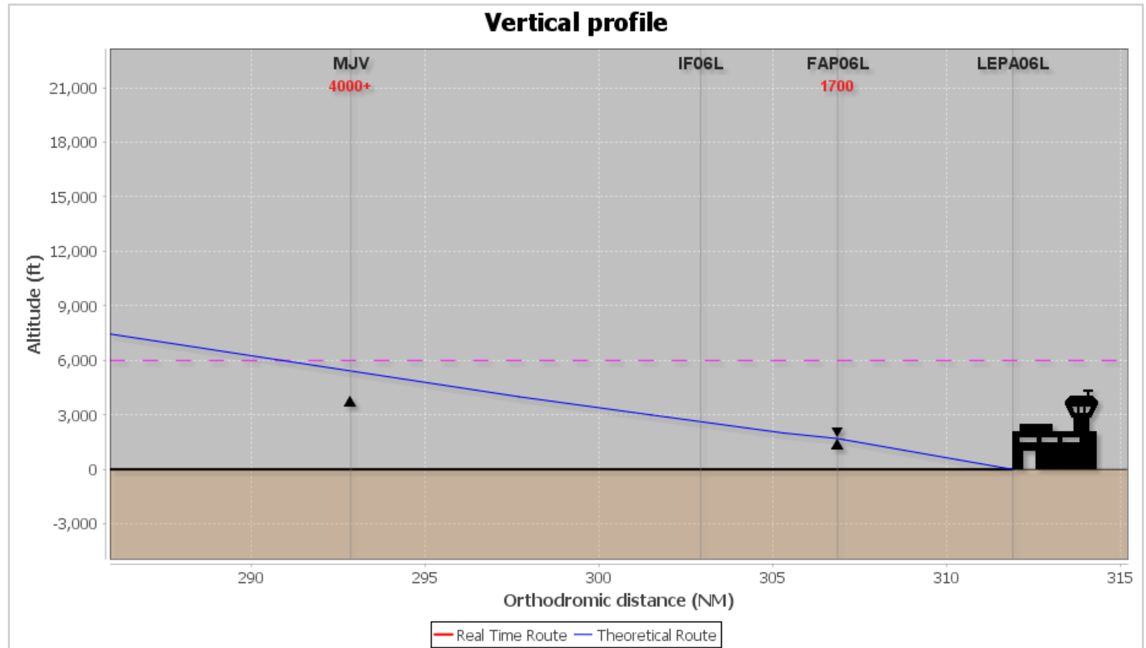


*Figura 49: Evolución del perfil vertical 9 – estado final*

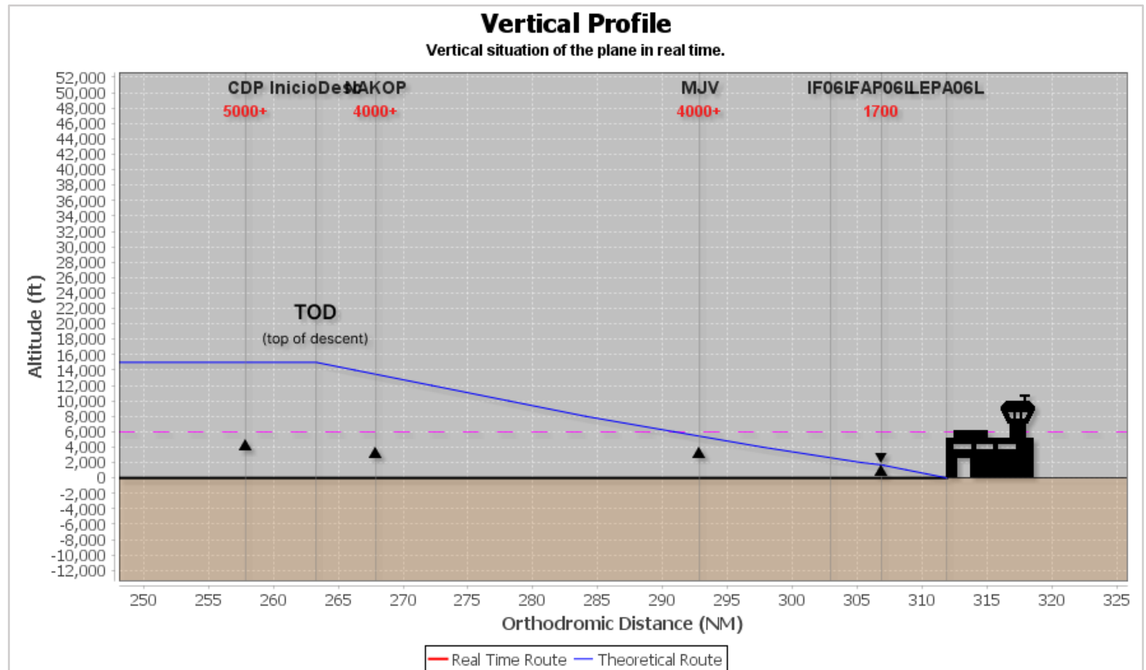


*Figura 50: Evolución del perfil vertical 10 – estado final*

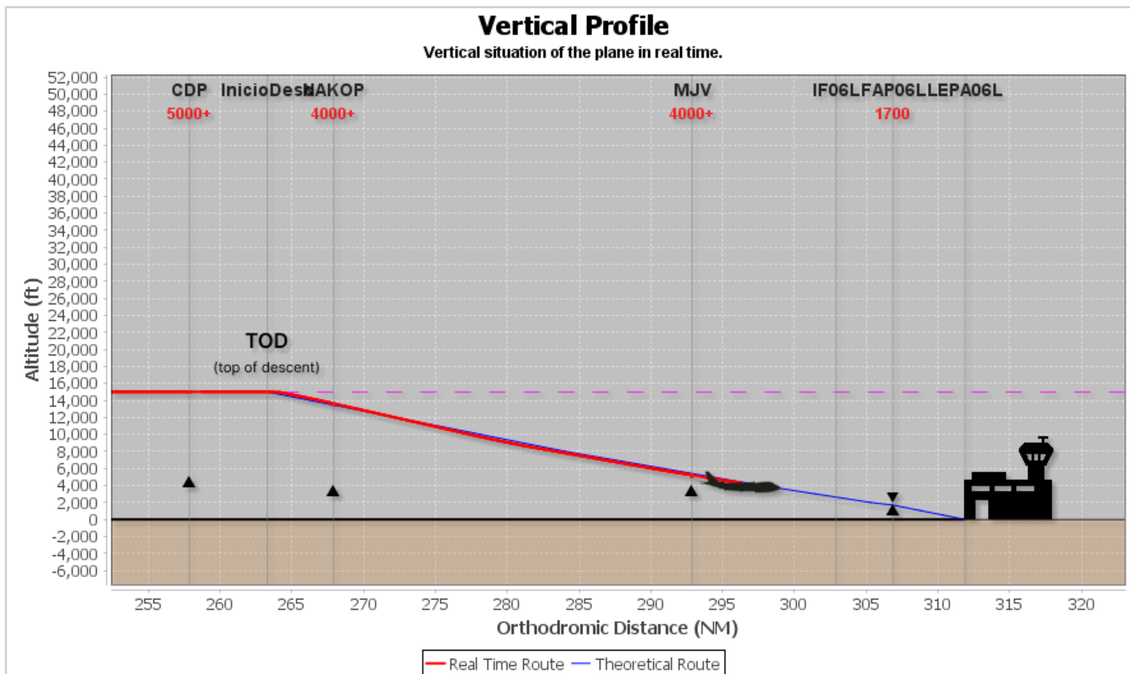


*Figura 51: Evolución del perfil vertical 11 – estado final*

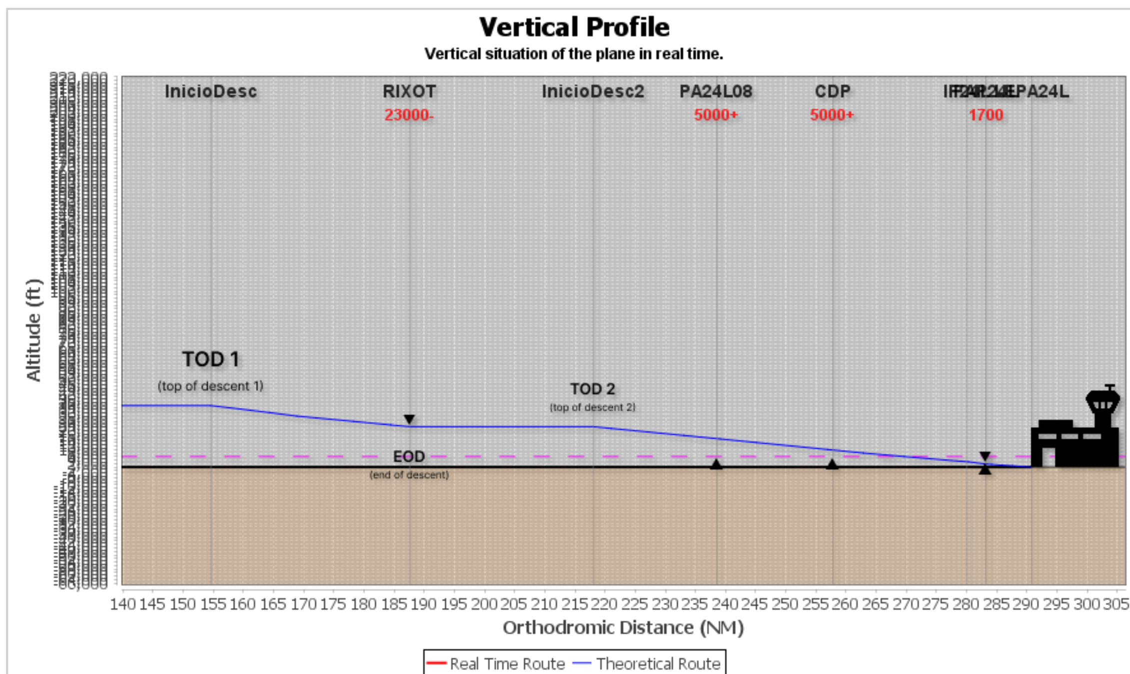


*Figura 52: Evolución del perfil vertical 12 – estado final*

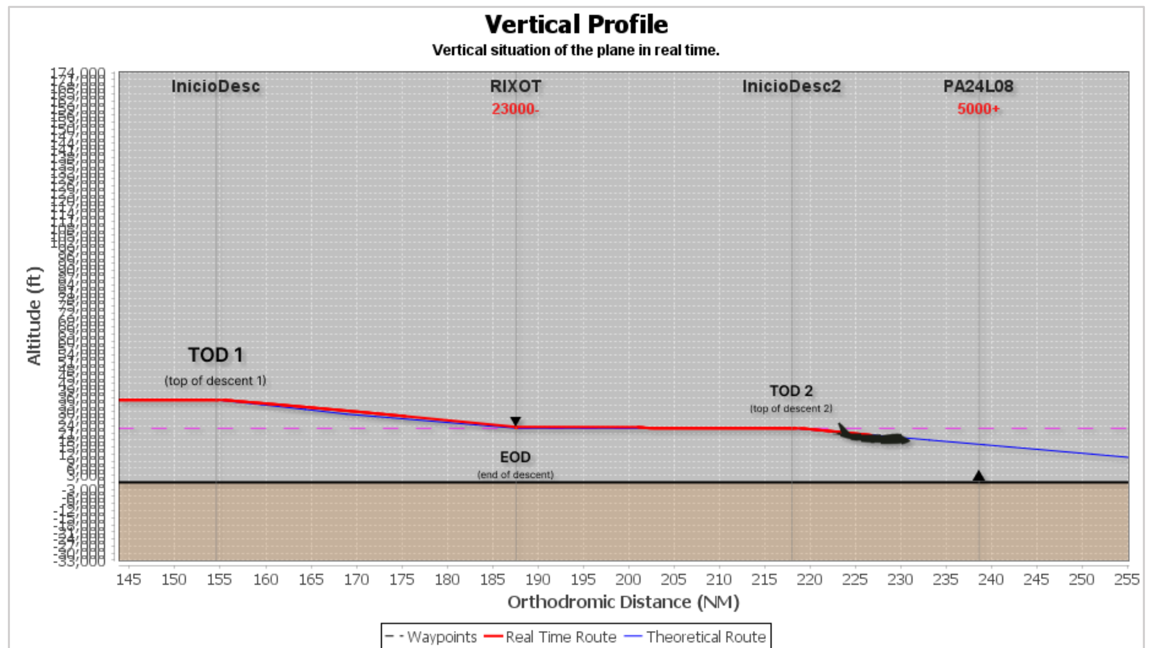


*Figura 53: Evolución del perfil vertical 13 – estado final*

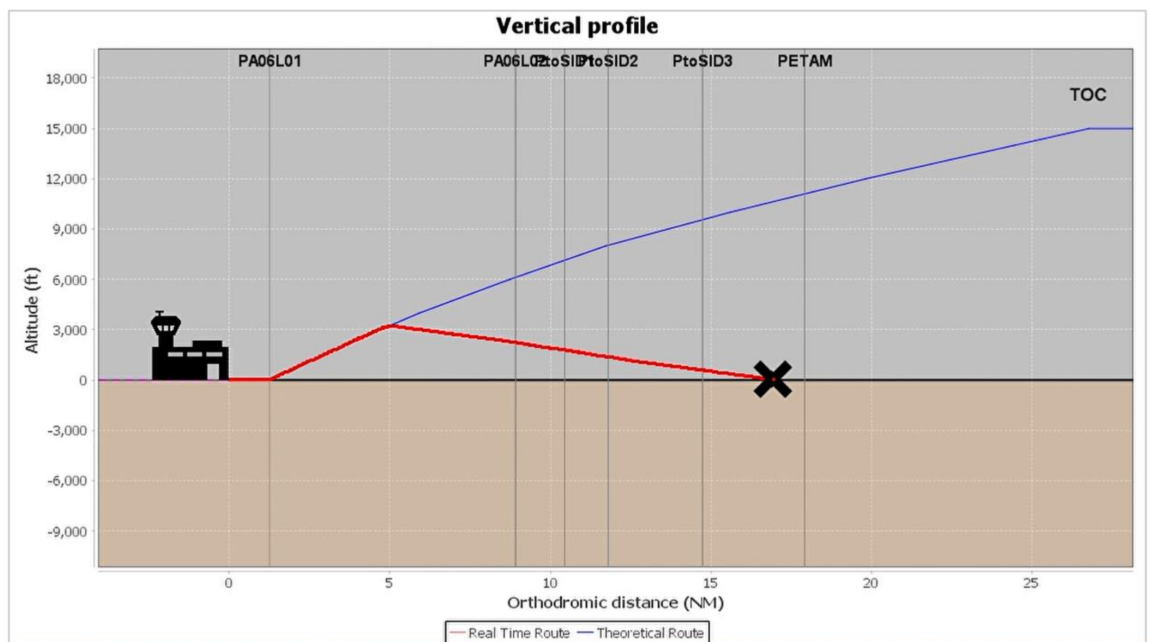


*Figura 54: Evolución del perfil vertical 14 – caso de colisión y paro de la simulación*

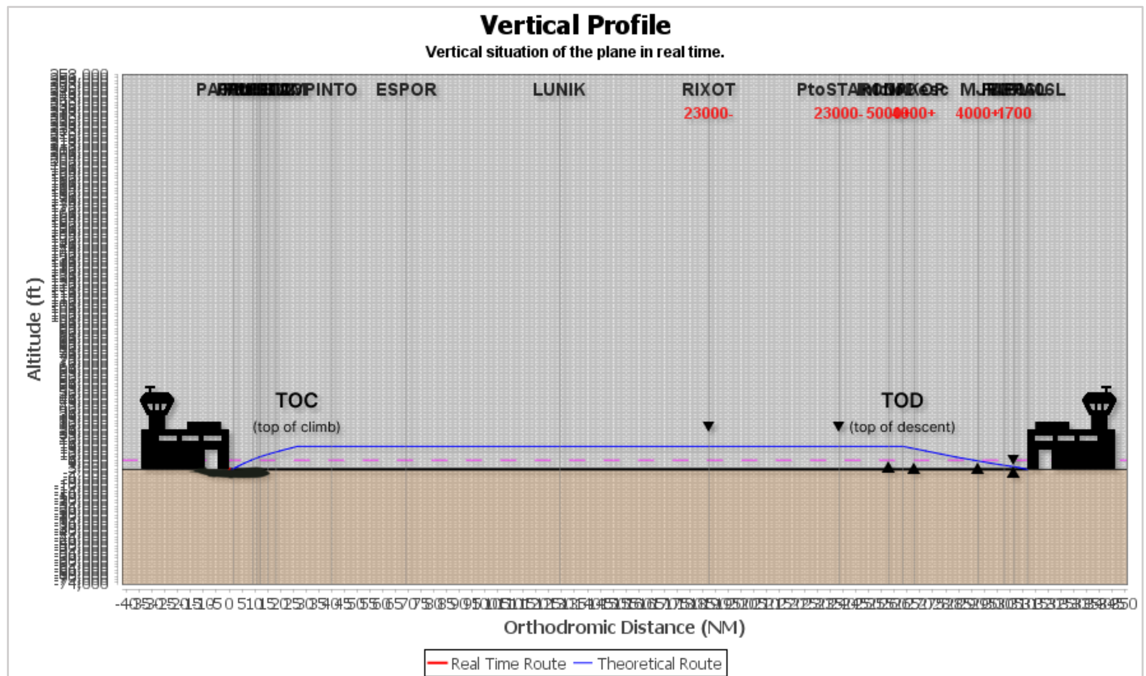


*Figura 55: Evolución del perfil vertical 15 – aspecto de una ruta completa*

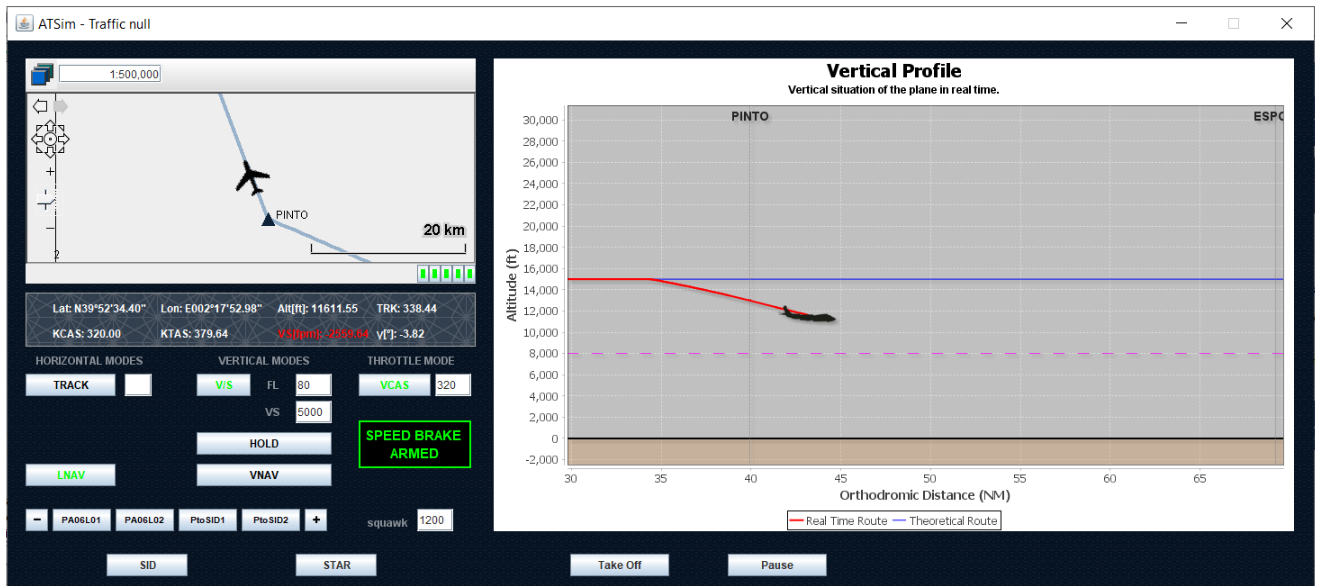


*Figura 56: Evolución del perfil vertical 16 – uso automático del indicador de aviso SPEED BRAKE ARMED*

# 3. Planos

Puesto que nuestro trabajo trata sobre el desarrollo de un software, no existen planos propiamente dichos para poder introducir en este apartado. No obstante, después de indagar sobre la posibilidad de dar forma a un plano que represente de forma gráfica la estructura del código, descubrimos que en el mundo de la programación existe un lenguaje que permite modelar, construir y documentar los elementos que forman un software orientado a objetos, cuyo caso es el nuestro. Dicho lenguaje se hace llamar lenguaje UML.

## 3.1. Introducción al lenguaje UML

En UML existen diferentes tipos de diagramas, para ser exactos en la primera versión de UML hay 10 diagramas, mientras que UML 2.0 se compone de trece diagramas. Todos ellos se categorizan jerárquicamente, habiendo tres subgrupos diferenciados:

- **Diagramas de estructura**, enfatizan en los elementos que deben existir en el sistema:

    1. Diagrama de clases
    2. Diagrama de componentes
    3. Diagrama de objetos
    4. Diagrama de estructura compuesta (UML 2.0)
    5. Diagrama de despliegue
    6. Diagrama de paquetes

- **Diagramas de comportamiento**, se centran en lo que debe suceder en el sistema modelado.

    7. Diagrama de actividades
    8. Diagrama de casos de uso
    9. Diagrama de estados

- **Diagramas de interacción**, un subtipo de los diagramas de comportamiento, cuyo objetivo es enfatizar en el flujo de control y de datos entre los elementos del sistema.

    10. Diagrama de secuencia
    11. Diagrama de comunicación
    12. Diagrama de tiempos (UML 2.0)
    13. Diagrama de vista de interacción (UML 2.0)

A pesar de la gran variedad de diagramas que brinda UML, se ha optado por la utilización de únicamente uno de ellos. Las dimensiones del presente proyecto hacen que el hecho de realizar todos los diagramas represente un reto más que considerable dentro de esta memoria, ya que sería muy complejo y se ha considerado que ese no es el objetivo principal de la misma.

Dicho esto, se ha optado por la utilización del diagrama de clases, ya que muestra de forma global la estructura del programa.

## 3.2. Introducción al diagrama de clases

Un diagrama de clases muestra la estructura de un sistema describiendo sus clases, atributos y relaciones entre ellos. Habitualmente los diagramas de clases son utilizados durante el proceso de análisis y diseño de los sistemas.

A continuación, se muestra de qué se compone principalmente un diagrama de este tipo, donde se pueden diferenciar principalmente dos elementos: las clases y sus relaciones.

### 3.2.1. Clases

Dentro de cada clase podemos encontrar básicamente dos elementos:

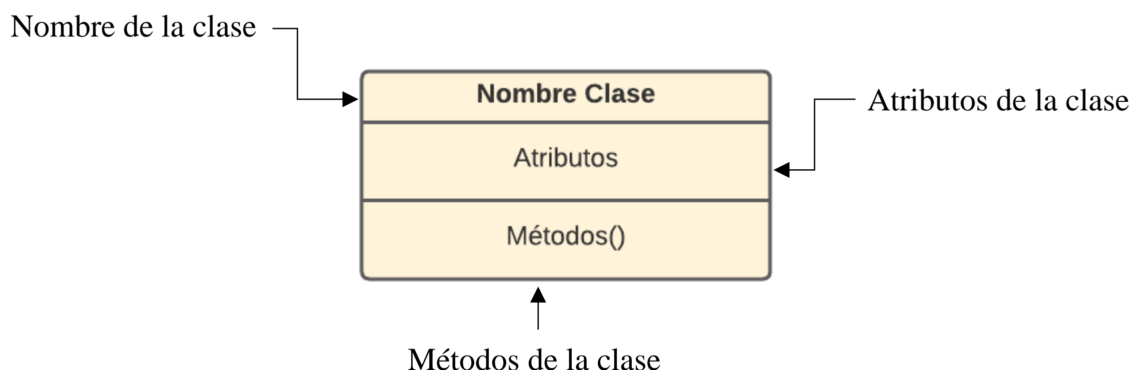


*Figura 57: Representación genérica de una clase en UML*

Tanto los atributos como los métodos pueden ser:

- *Public* (+): Indica que el atributo/método podrá ser visible tanto dentro como fuera de la clase, lo que quiere decir que es accesible desde cualquier lugar.

- *Private* (-): Indica que el atributo/método únicamente será accesible desde dentro de la clase a la que pertenece, es decir, sólo otros métodos de la misma clase lo pueden utilizar.
- *Protected* (#): Indica que el atributo/método no será accesible desde fuera de la clase, pero sí podrá ser manipulado por métodos de la clase y de sus subclases.

### 3.2.2. Relaciones

En los diagramas UML, hay una gran variedad de relaciones entre clases, y cada una de ellas se representa con un tipo de flecha o en general, con diferentes simbologías. En nuestro caso, hemos tomado el siguiente criterio.

En caso de tener dos clases, y que la primera llame a la segunda, las relaciones se representan como en la siguiente figura.

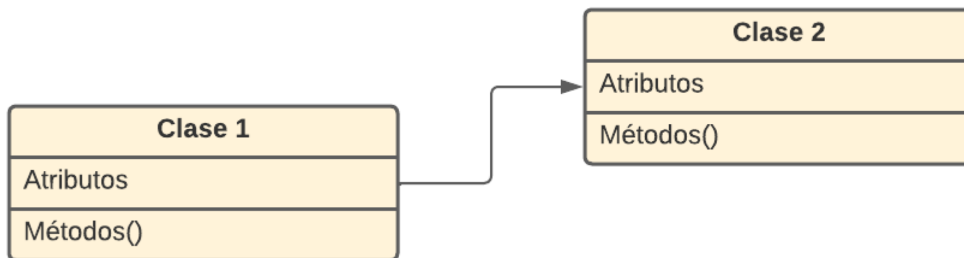


*Figura 58: Llamada de la clase 2 desde la clase 1*

No obstante, en el caso de que las dos clases se llamen entre sí, la forma de indicarlo será como el ejemplo de la figura 59.

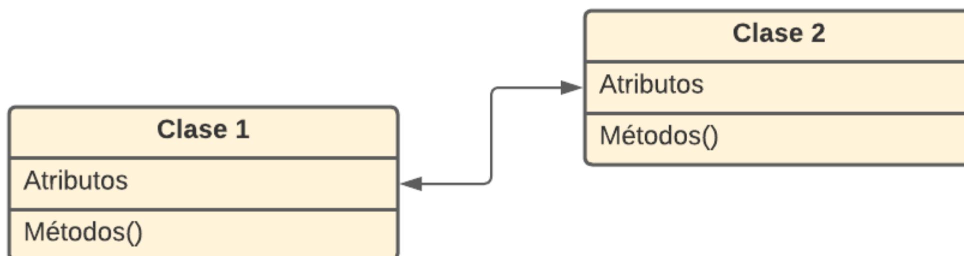


*Figura 59: Llamada de la clase 2 desde la clase 1 y viceversa*

## 3.3. Diagrama de clases

Nuestro diagrama de clases es un diagrama de clases simplificado, puesto que no se han tenido en cuenta el tipo de relaciones reales que existe entre clases (asociación, dependencia, generalización, realización…), ni tampoco la multiplicidad. Habitualmente, este tipo de desarrollos se hacen antes de la realización del proyecto como tal, en este caso, el diagrama se ha creado a partir del proyecto ya acabado, lo que dificulta bastante categorizar las relaciones entre clases. Nótese también, que este proyecto no está pensado para hacer un diagrama de este tipo, debido principalmente a su gran extensión y complejidad a nivel de código.

### 3.3.1. Módulos del diagrama

Debido a que hay algunas clases con un gran número de atributos y/o métodos, vamos a introducir en este apartado cada uno de ellos por separado detalladamente.

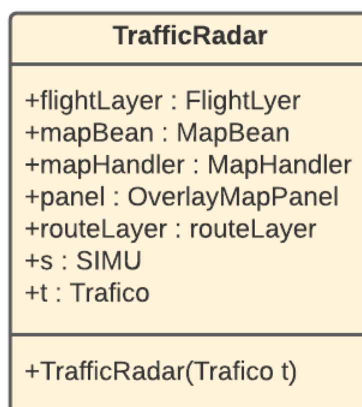


*Figura 60: Clase TrafficRadar*

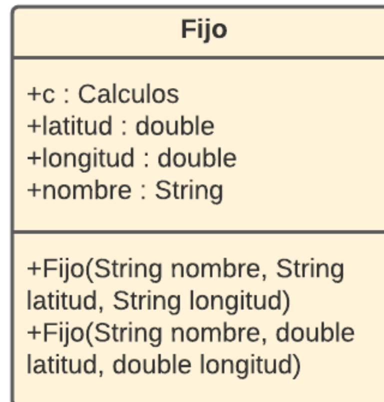


*Figura 61: Clase Fijo*

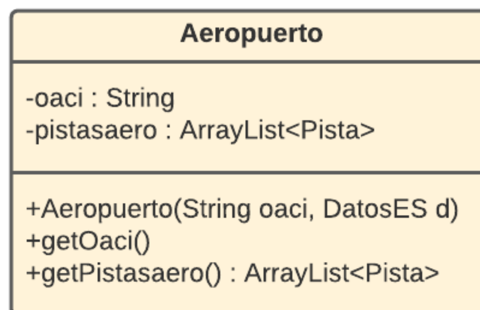


*Figura 62: Clase Aeropuerto*

```
VerticalRoute
----------------------------------------
+ANSI_RED : String
+ANSI_RESET : String
-DASHED_LINE : BasicStroke
+EODIcon : BufferedImage
+L1 : double
+TOCIcon : BufferedImage
+TOD1Icon : BufferedImage
+TOD2Icon : BufferedImage
+TODIcon : BufferedImage
+acum_dist_global : double
+acum_dist_real : double
+altura_correcta : double
+angulo : double
+angulos_descenso : ArrayList <Double>
+angulos_descenso_invert : ArrayList<Double>
+array_L1 : ArrayList<Double>
+array_acum_dist_giro : ArrayList<Double>
+arrya_alturas : ArrayList<Double>
+array_dist_giro : ArrayList<Double>
+array_dist_real : ArrayList<Double>
+array_dist_wpts : ArrayList<Double>
+array_distancias : ArrayList<Double>
+c : Calculos
+carreraTO : double
+chart : JFreeChart
+dist : double
+dist_giro : double
+dist_ini : double
+dist_total : double
+dist_wpt_avion : double
+distancia : double
+distancia_acum : ArrayList<Double>
+distancia_inicio_wpt_0 : double
+e : enMovimiento
+lastAngle : double
+lat_dist_wpts : ArrayList<Double>
+lat_ini : double
+lat_wpt : double
+latitud : ArrayList<Double>
+lon_dist_wpts : ArrayList<Double>
+lon_ini : double
+lon_wpt : double
+longitud : ArrayList<Double>
+negativetriangleicon : BufferedImage
+nombre : String
+oDataset : XYSeriesCollection
+oSeries1 : XYSeries
+oSeries2 : XYSeries
+oSeries3 : XYSeries
+oSeries4 : XYSeries
+oSeries5 : XYSeries
+oSeries6 : XYSeries
+panel : ChartPanel
+planeIcon : BufferedImage
+positivetriangleIcon : BufferedImage
+rotatedPlaneIcon : BufferedImage
+torreIcon1 : BufferedImage
+torreIcon2 : BufferedImage
+triangleIcon : BufferedImage
+puntos_descenso : ArrayList<Double>
+s : SIMU
+t : Trafico
+xyannotation : XYAnnotation
+xyplot : XYPlot
+crashIcon : BufferedImage
----------------------------------------
+VerticalRoute(Trafico t)
+getPanel() : ChartPanel
+getTrafico() : Trafico
+initWaypoints()
-loadTheoreticalRoute()
+makeTurns(enMovimiento e)
-obtainDistance()
+rotate(BufferedImage image, Double degrees) :
BufferedImage
+speedBrakeArmed()
+updateChartPanel() : ChartPanel
+updatePlaneMovement()
```

*Figura 63: Clase VerticalRoute*

```
PlanDeVuelo
----------------------------------------
+CI : JLabel
+Sim : String
+VI : String
+adep : JLabel
+ades : JLabel
+ads : JLabel
+altn : JLabel
+arcid : JLabel
+b1 : JButton
+barra1 : JLabel
+barra2 : JLabel
+barra3 : JLabel
+callsign : String
+cerrada : boolean
+com : JLabel
+comboestela : JComboBox
+comboreglas : JComboBox
+combotip : JComboBox
+combotipos : JComboBox
+d : DatosES
+desc : JLabel
+diahora : JLabel
+dosp : JLabel
+dosp2 : JLabel
+eobt : JLabel
+eobt2 : JLabel
+estela : JLabel
+fecha : Date
+incorrectos : String
+infosup : JButton
+masa_despegue : JLabel
+nivel : JLabel
+numero : JLabel
+paneles : ArrayList<JPanel>
+panelprin : JPanel
+pulsado : boolean
+reglas : JLabel
+ruta : JLabel
+rutavalida : boolean
+ssr : JLabel
+tCI : JTextField
+tadep : JTextField
+tades : JTextField
+tads : JTextField
+taltn : JTextField
+tarcid : JTextField
+taño : JTextField
+tcom : JTextField
+tdia : JTextField
+thh : JTextField
+thh2 : JTextField
+tip : JLabel
+tipo : JTextField
+tiposv : JLabel
+titulos : ArrayList<JLabel>
+tmasad : JTextField
+tmes : JTextField
+tmasad : JTextField
+tmm : JTextField
+tmm2 : JTextField
+tnivel : JTextField
+tnumero : JTextField
+totros : JTextArea
+tpres : JTextField
+truta : JTextArea
+tssr : JTextField
+ttexto : JTextArea
+tvelo : JTextField
+velocidad : JLabel
+wptañadidos : ArrayList<Fijo>
----------------------------------------
+PlanDeVuelo(String callsign, String Sim, DatosES d)
+actionPerformed(ActionEvent ae)
+añadirbarra(int x, int y, int width, int height, String titulo)
+añadirpanelint x, int y, int width, int height, String titulo)
+buscarAeropuerto(String aero) : boolean
+focusGained(FocusEvent fe)
+focusLost(FocusEvent fe)
+getActTime() : String
+getAlt() : String
+getAlternativo() : String
+getAtype() : String
+getCI() : int
+getDep() : String
+getDepTime() : String
+getDes() : String
+getETTotal() : String
+getMasaD() : double
+getOtros() : String
+getRoute() : String
+getTAS() : String
+getVI() : String
+iniciar()
+obtenerAeronaves() : String[]
+reconocerRuta() : boolean
+windowActivated(WindowEvent we)
+windowClosed(WindowEvent we)
+windowClosing(WindowEvent we)
+windowDeactivated(WindowEvent we)
+windowDeiconified(WindowEvent we)
+windowIconified(WindowEvent we)
+windowOpened(WindowEvent we)
```

*Figura 64: Clase PlanDeVuelo*

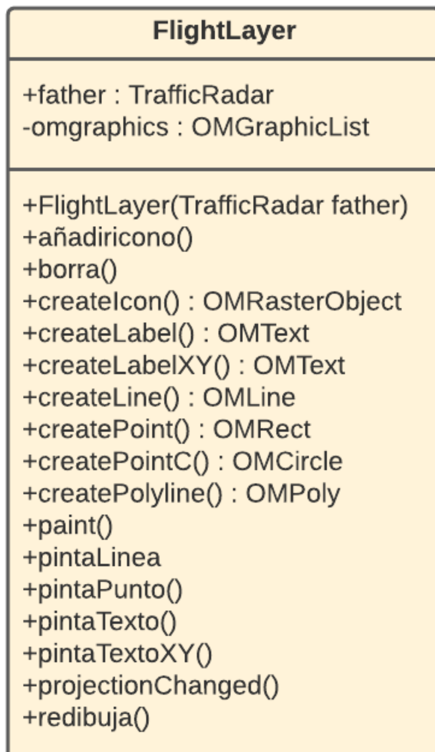


*Figura 65: Clase FligthLayer*

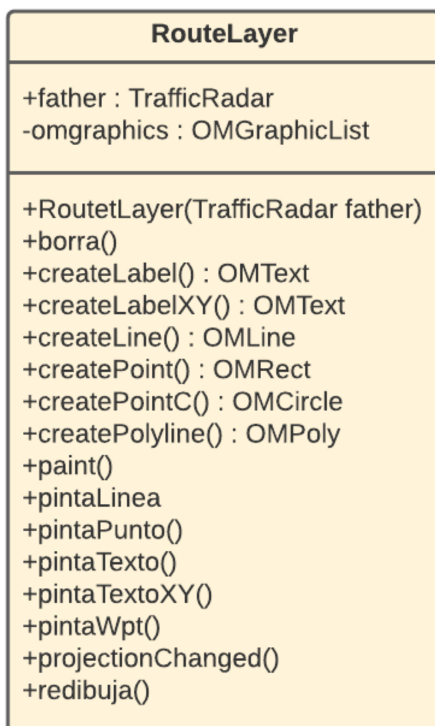


*Figura 66: Clase RouteLayer*

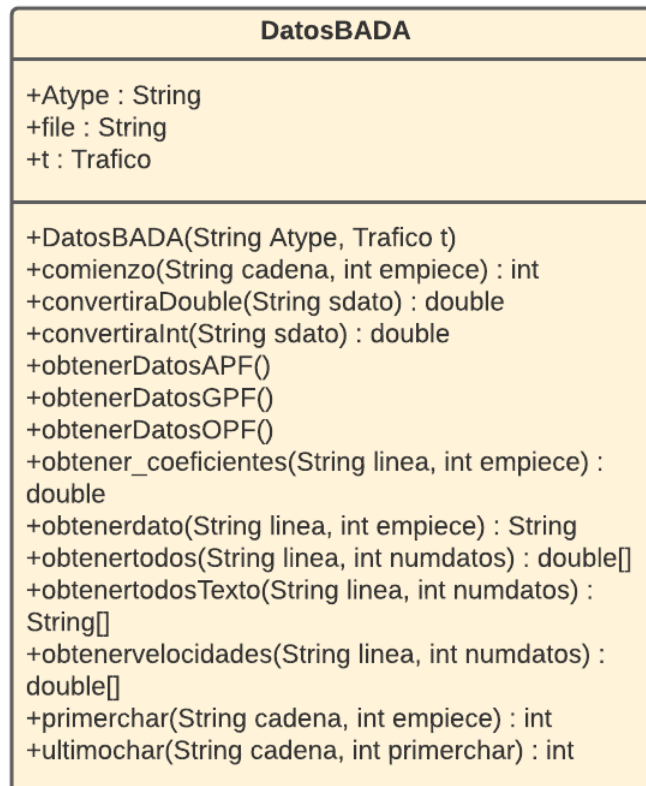


*Figura 67: Clase DatosBADA*

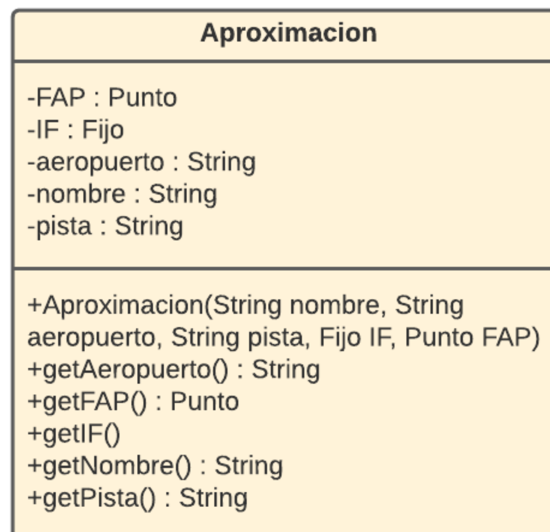


*Figura 68: Clase Aproximación*

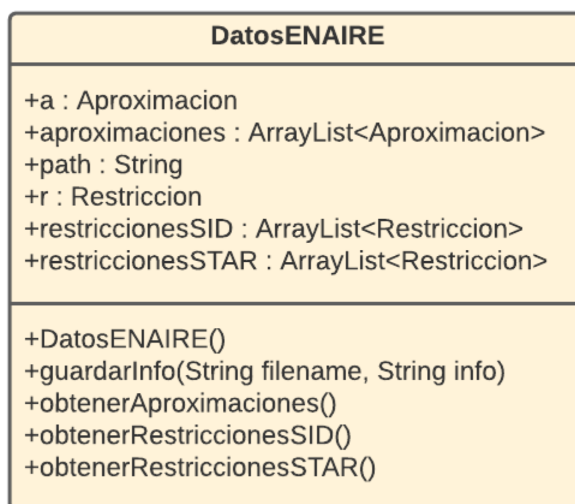


*Figura 69: Clase DatosENAIRE*

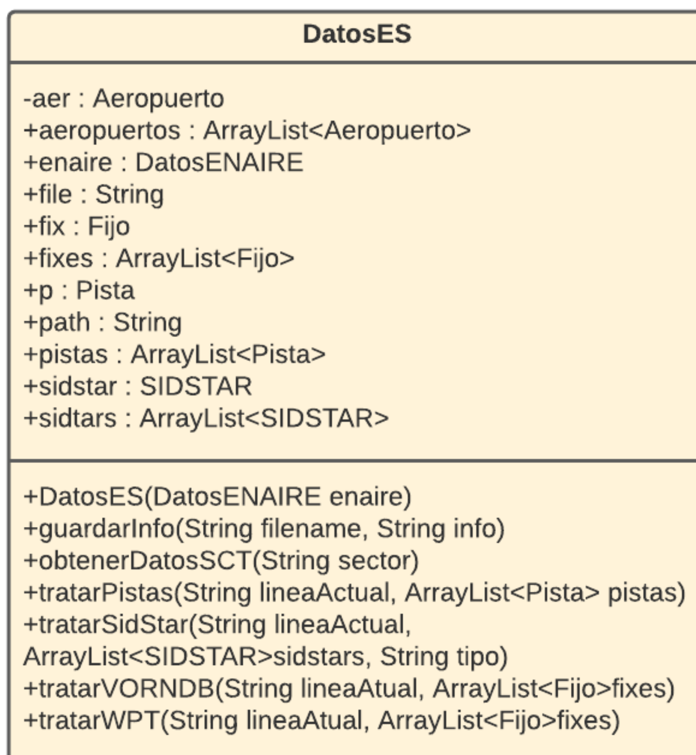


*Figura 70: Clase DatosES*

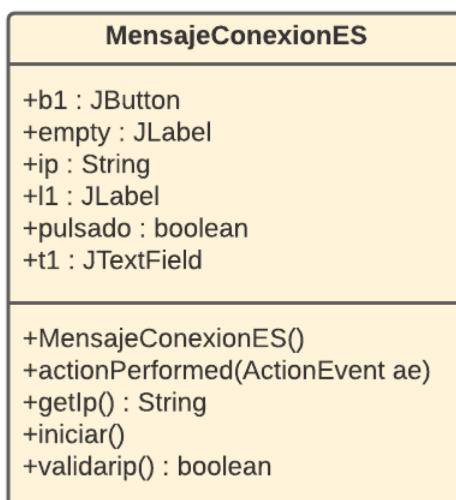


*Figura 71: Clase MensajeConexionES*

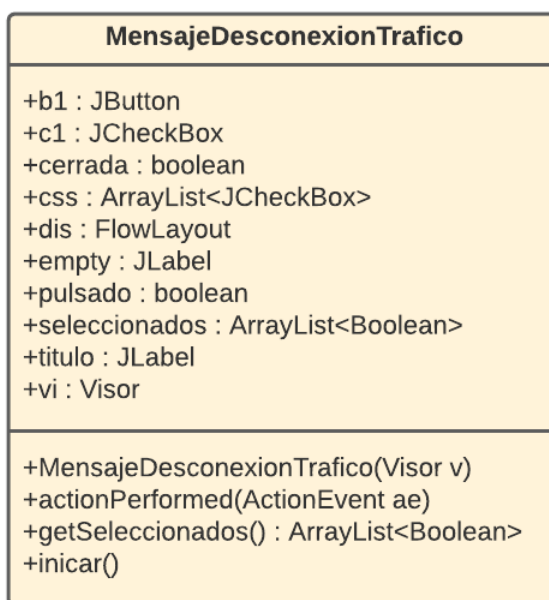


*Figura 72: Clase MensajeDesconexiónTrafico*

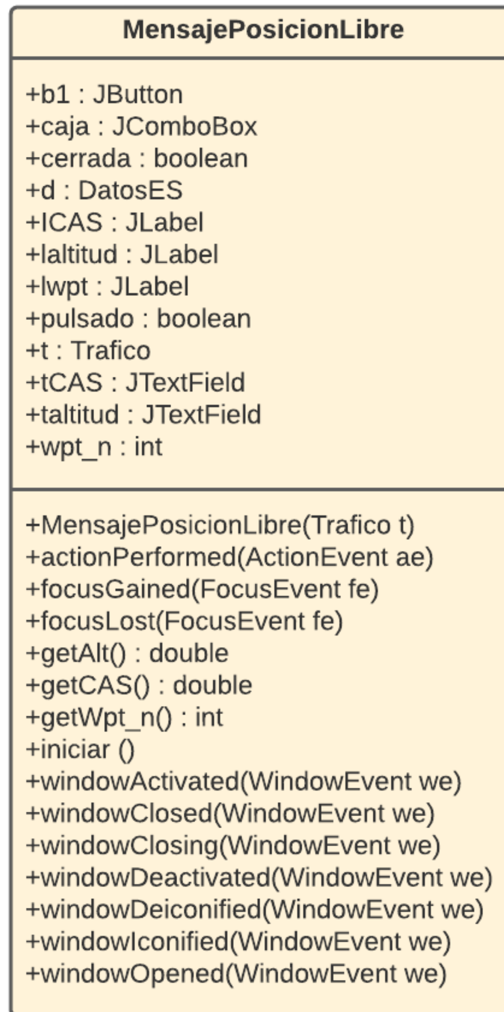


*Figura 73: Clase MensajePosicionLibre*

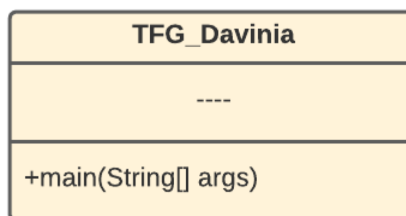


*Figura 74: Clase TFG_Davinia*

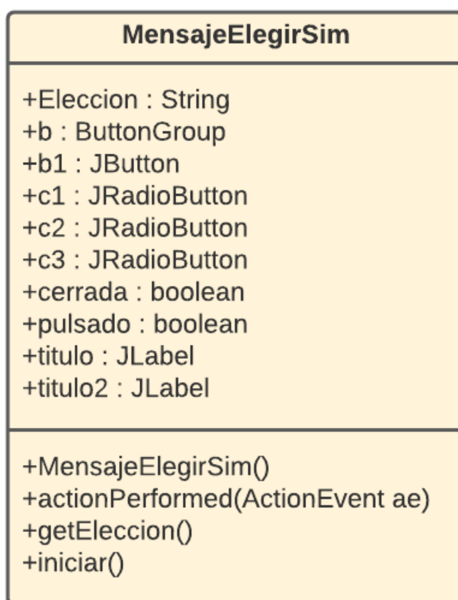


*Figura 75: Clase MensajeElegirSim*

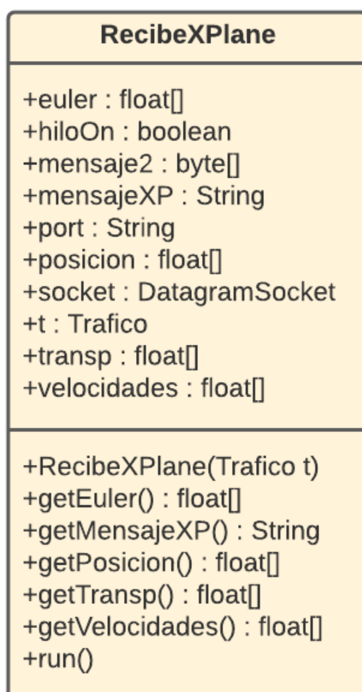


*Figura 76: Clase RecibeXPlane*

```
Calculos
-----------------------------------------------
+Cvmin : double
+FL : int[]
+P0 : double
+R : double
+Rair : double
+T0 : double
+Vcli : double[]
+Vdesi : double[]
+a0 : double
+alfa_asc : double[]
+alfa_desc : double[]
+amaxl : double
+amaxn : double
+bt : double
+g0 : double
+k : double
+rho0 : double
+t : Trafico
-----------------------------------------------
+Calculos()
+Calculos(Trafico t)
+CAS_CI(double alt) : double
+CAStoTAS(double alt, double CAS) : double
+CD(double alt, double TAS) : double
+CL(double alt, double TAS) : double
+ComienzoDesac(double vs, double vscalculada, double alttarget, double alt) : double
+Configuracion(double alt, double CAS, double vs) : int
+Drag(double alt, double TAS) : double
+ Fvc() : double
+Hptrans(double CAS, double M) : double
+Ma_CI(double alt) : double
+MachtoTAS(double alt, double Mach) : double
+Radiogiro(double TAS) : double
+RumboObjetivo(int wptdir, int wpt2) : double
+TASLimitadaVS(double VS, double alt, double TAS0, double Empuje) : double
+TAStoCAS(double alt, double TAS) : double
+TAStoMatch(double alt, double TAS) : double
+Tclimb(double alt, double TAS) : double
+Tdesc(double alt, double TAS) : double
+Throttle(double TAS, double VS, double alt) : double
+ThrottleCR(double alt, double TAS) : double
+TmaxCR(double alt, double TAS) : double
+Tmaxclimb(double alt, double TAS) : double
+VMODanger(double alt, double CAS) : boolean
+VSDCM(double TAS, double pendiente, double alt) : double
+VSspeedEnergyModel(double alt, double TAS) : double
+V_max_alcance(double alt) : double
+VstallDanger(double CAS) : boolean
+acelLD() : double
+acelTO() : double
+altitudRestriccion(int wptdirecto, Restriccion r) : String
+altitudRestriccion(int wptdirecto, Restriccion r) : String
+anguloviraje(int wptdir, int wpt2) : double
+cadenaAlt2Double(Sring altitud_restr) : double
+calculoDtotal(double TAS, double seg, double pendiente) : double
+calculoLat2(double dTotal, double rumbo, double lat1) : double
+calculoLon2(double dTotal, double rumbo, double lon1, double newlat, double lat1) : double
+calculoPuntosRutaVerticalAscenso(int[] FL, double[] alfa_asc) : double[]
+calculoPuntosRutaVerticalDescenso(int[] FL, double[] alfa_desc) : double[]
+calculoRumbo(double latf, double lonf, double lat, double lon) : double
+calculoRumboPlan(int pos, double TAS) : double
+calculoTASactual(double TAS0, double TAStarget, double tiempo) : double
+calculoheading(double headingf, double TAS) : double
+carreraLD() : double
+carreraTO() : double
+comprobarAceleracionNormal(double pendiente, double TAS, double tiempo, double
vscalculada, double alttarget, double alt) : double
+comprobarEmpujeMaxMin(double Empuje, double alt, double TAS) : double
+consumoespecifico(double TAS) : double
+deg2degminsec(double degr, String LATLON) : String
+deg2rad(double deg) : double
+degminsec2deg(String str) : double
+degminsec2rad(char a, double deg, double min, double sec) : double
+determinarTAS(double alt) : double
+determinarTASTurboprop(double alt) : double
+distancia(double v0, double t) : double
+distanciaLD(double v0, double t) : double
+distanciaOrto(double lat1, double lat2, double lon1, double lon2) : double
+distcomienzoviraje(int wptdir, int wpt2m double TAS) : double
+flujo_comb_fase(double TAS, double alt) : double
+flujo_combustible(double TAS) : double
+flujo_combustible_appld(double TAS, double alt) : double
+flujo_combustible_cr(double TAS) : double
+flujo_combustible_min(double alt) : double
+fmcCAS(double alt, double TAS) : double
+fmcMa(double alt, double TAS) : double
+formatoPendiente(double VS, double pte) : double
+fpm2ms(double fpm) : double
+ft2m(double ft) : double
+greatCircleDistance(double lat1, double lat2, double lon1, double lon2) : double
+kmh2knot() : double
+knot2kmh() : double
+knot2ms(double knots) : double
+knot2kms(double knots) : double
+m2ft(double m) : double
+ms2fpm(double ms) : double
+ms2knot(double ms) : double
+pendientemaxima(double pendiente, double TAS, double tiempo) : double[]
+pteRestriccion(double alt, double alt_restr, double lat, double lon, int wptdirecto) : double
+rad2deg(double rad) : double
+rad2degminsec(double rad) : String
+tasadegiroBADA(double TAS) : double
+tieneRestriccionAlt(int wptdirecto, Restriccion r) : boolean
+tieneRestriccionAlt(String nombre, Restriccion r) : boolean
+unidadesPendiente(double VS, double pte) : String
+velocidad(double v0, double t) : double
+velocidadLD(double v0, double t) : double
+vs3degreeSlope(double TAS) : double
+vspteRestriccion(double TAS, double alt, double alt_restr, double lat, double lon, double
wptdirecto) : double
+restarFL() : double[]
+intermediateGreatCirclePoint(double lat1, double lat2, double lon1, double lon2, double dist1,
double dist_total) : Arraylist
+km2NM(double km) : double
```

*Figura 77: Clase Calculos*

**Trafico**

-ADL : double
-AP : double[]
-ActTime : String
-Atype : String
-CAS : String
-CAStarget : double
-CI : int
-CR : double[]
-Cf : double[]
-Cfcr : double
-Cfdes : double[]
-Cvmin : double
-Cvminto : double
-DCM : boolean
-DepTime : String
-EETTotal : String
-Empuje : double
-Escalonado : boolean
-Hmaxapp : double
-Hmaxic : double
-Hmaxld : double
-Hmaxto : double
-IC : double[]
-LD : double[]
-LDL : double
+Llegada : Fijo
+cruisealt_2 : double
+dentro : boolean
-MEGABYTE : long
-MMO : double
+Mcl : double
+Mcr : double
+Mdes : double
-Rumbo : double
-S : double
-Sim : String
-TAS : double
-TASCR : String
+TFin : long
+TInicio : long
-ip : String
-ipXP : String
-lat : double
-lon : double
-masa : double
-masaref : double
+noiac : Aproximacion
-nombre : String
-orientacion : int
-TO : double[]
+TOD : Fijo
-TOL : Fijo
-VI : String
-VMO : double
-VS : double
+Vcl1 : double
+Vcl2 : double
-Vcli : double[]
+Vcr1 : double
+Vcr2 : double
+Vdes1 : double
+Vdes2 : double
-Vdesi : double[]
-acel_long : double
-acel_n : double
+adep : Aeropuerto
+ades : Aeropuerto
-altitud : double
-altn : double
-arrayconfi : String[]
+avisadosocket : boolean
+bada : DatosBADA
-bank_others : double
-bank_told : double
+buscandoNuevoWpt : boolean
+c : Calculos
-callsign : String
+cli : Socket
-configuracion : int
-cruise : boolean
+cruisealt : String
-ctc : double[]
-ctcr : double
-ctdes : double[]
+d : DatosES
+dep : String
-des : String
+dtescenario : int
+e : enMovimiento
-engineType : String
+escenario : boolean
-estado : int
+fecha : String
+filename : String
+formato : DecimalFormat
-headingES : double
+iac : Aproximacion
+idioma : String
+p : PlanDeVuelo
-password : String
-pendiente : double
+pendienteCI : double
+pistaA : int
+pistaAnombre : String
+pistaD : in
+pistaDnombre : String
+pistadep : Pista
+pistades : Pista
-portEXP : int
-portRXP : int
+distancia_inicio_TOD2 : double
+posiblesIAC : ArrayList<Aproximacion>
+posiblesSID : ArrayList<SIDSTAR>
+posiblesSTAR : ArrayList<SIDSTAR>
+puntosIAC : ArrayList<Fijo>
-puntosnoIAC : ArrayList<Fijo>
+r : RecibeES
+rX : RecibeXPlane
-remarks : String
-route : String
+running : boolean
+sid : SIDSTAR
-simu : SIMU
+situado : boolean
-squawk : String
-squawkident : float
-squawkmsg : String
-squawkstatus : float
+star : SIDSTAR
+tiempo_ : double
+trafficRadar : TrafficRadar
-usuario : String
+verticalRoute : VerticalRoute
-virando : boolean
+wptescenario : String
-wptsRuta : ArrayList<Fijo>
-wptsSID : ArrayList<Fijo>
-wptsSTAR : ArrayList<Fijo>
-wptsSoloRuta : ArrayList<Fijo>

+Trafico(String ip, DatosES d, String idioma, String fecha)
+Trafico(String ip, DatosES d, String idioma, String m_conexion, String m_fp, String m_posicion, String fecha)
+RecibirXPlane()
-asignarIAC()
-asignarSID()
-asgnarSTAR()
+asignar_aeropuerto(String nombre) : Aeropuerto
+asignar_datos_m_conexion(String m_conexion)
+asignar_datos_m_fp(String m_fp)
+asignar_pistas()
+asignar_pos_inical(String m_posicion)
+asignar_ruta_escenario(String ruta)
+avisarSocketNoAbierto()
+bytesToMegabytes(long bytes) : long
+cabeceraTraza(String filename)
+cabeceraTraza(String filename)
+calculo_punto_descenso() : Fijo
+calculo_punto_descenso_2() : Fijo
+distancia_inicio_desc(Fijo findesc, double pendiente, Punto FAP) : double
+distancia_inicio_desc_2(Fijo findesc, double pendiente, Punto FAP) : double
+elegirPosicionLibre() : boolean
-enviarXP()
+enviarES(String cadena, Socket cli) : boolean
+formatoTimepo(double tiempo) : String
-formularioConexionXP() : boolean
+getADL() : double
getAP() : double[]
+getAcel_long() : double
+getAcel_n() : double
+getAltitud() : double
+getBank_others() : double
+getBank_told() : double
+getCAS() : double
+getCAStarget() : double
+getCI() : int
+getCR() : double[]
+getCallsign() : String
+getCf() : double[]
+getCfcr() : double
+getCfdes() : double[]
+getConfiguracion() : int
-inicio() : boolean
+isCruise() : boolean
+isDCM() : boolen
+isEscalonado() : boolean
+isSituadio() : boolean
+isVirando() : boolean
+memoria()
-pedirDatosUsuario() : boolean
-pedirFP() : boolean
-pedirPista() : boolean
-pedirSim() : boolean
+getCruisealt() : String
+getCtc() : double[]
+getCtcr() : double
+getCtdes() : double[]
+getCvmin() : double
+getCvminto() : double
+getDep() : String
+getDes() : String
+getE() : enMovimiento
+getEmpuje() : double
+getEngineType() : String
+getHmaxapp() : double
+getHmaxic() : double
+getHmaxld() : double
+Hmaxto() : double
+getIC() : double[]
+getIp() : String
+getIpXP() : String
+getLD() : double[]
+getLDL() : double
+getLat() : double
+getLon() : double
+getMMO() : double
+getMasa() : double
+getMasaref() : double
+getMcl() : double
+getMcr() : double
+getMdes() : double
+getOrientacion() : int
+getPendiente() : double
+getPortEXP() : int
+getPortRXP() : int
+getRumbo() : double
+getS() : double
+getSim() : String
+getSimu() : SIMU
+getSquawk() : String
+getTAS() : double
+getTASCR() : String
+getTO() : double[]
+getTOL() : double
+getVI() : String
+getVMO() : double
+getVS() : double
+getVcl1() : double
+getVcl2() : double
+getVcli() : double[]
+getVcr1() : double
+getVcr2() : double
+getVdes1() : double
+getVdes2() : double
+getVdesi() : double[]
+getVerticalRoute() : VerticalRoute
+getWptsRuta() : ArrayList<Fijo>
+getWptsSID() : ArrayList<Fijo>
+getWptsSTAR() : ArrayList<Fijo>
+getWptsSoloRuta() : ArrayList<Fijo>
+guardarTraza(String filename, double tiempo)
+guardarTrazaTest(String filename, double tiempo)
+run() ↑ Thread
+selec_punto_final(double pendiente, ArrayList<Fijo> aproxstar, Punto FAP) : Fijo
+selec_punto_inicio(double distanciaInicio, Fijo findesc, ArrayList<Fijo> aproxstar) : Fijo
+setADL(double ADL)
+setAP(double[] AP)
+setAcel_long(double acel_long)
+setAcel_n(double acel_n)
+setAltitud(double altitud)
+setBank_others(double bank_others)
+setBank_told(double bank_told)
+setCAS(double CAS)
+setCAStarget(double CAStarget)
+setCI(int CI)
+setCR(double[] CR)
+setCf(double[] Cf)
+setCfcr(double Cfcr)
+setCfdes(double[] Cfdes)
+setConfiguracion(int configuracion)
+setCruise(boolean cruise)
+setCtc(double[] ctc)
+setCtcr(double ctcr)
+setCtdes(double[] ctdes)
+setCvmin(double Cvmin)
+setCvminto(double Cvminto)
+setDCM(boolean DCM)
+setE(enMovimiento e)
+setEmpuje(double Empuje)
+setEngineType(String engineType)
+setEscalonado(boolean Escalonado)
+setHeadingES(double HeadingES)
+setHmaxapp(double Hmaxapp)
+setHmaxic(double Hmaxic)
+setHmaxld(double Hmaxld)
+setHmaxto(double Hmaxto)
+setIAC(Aproximacion iac)
+setIC(double[] IC)
+setIpXP(String ipXP)
+setLD(double[] LD)
+setLDL(double LDL)
+setLat(double lat)
+setLondouble lon)
+setMMO(double MMO)
+setMasa(double masa)
+setMasaref(double masaref)
+setMcl(double Mcl)
+setMcr(double Mcr)
+setMdes(double Mdes)
-setOnRunwayHeader() : boolean
+setPendiente(double pendiente)
+setPortEXP(int portEXP)
+setPortRXP(int portRXP)
+setRumbo(double Rumbo)
+setS(double S)
+setSid(SIDSTAR sid)
+setSim(SIMU Sim)
+setSquawk(String squawk)
+setStar(SIDSTAR star)
+setStar(ArrayList<Fijo>wptsS)
+setTAS(double TAS)
+setTASCR(String TASCR)
+setTO(double[] TO)
+setTOL(double TOL)
+setVMO(double VMO)
+setVS(double VS)
+setVcl1(double Vcl1)
+setVcl2(double Vcl2)
+setVci(double[] Vcli)
+setVcr1(double Vcr1)
+setVcr2(double Vcr2)
+setVdes1(double Vdes1)
+setVdes2(double Vdes2)
+setVdesi(double[] Vdesi)
+setVirando(boolean virando)
+setnoIAC(Aproximacion noiac)
+obtencionCoordenadasTOD() : ArrayList
+obtencionCoordenadasTOD_2() : ArrayList
+selec_punto_inicio_2(double distanciaInicio, Fijo findesc, ArrayList<Fijo> aproxstar) : Fijo
+setVerticalROute(VerticalRoute verticalRoute)
+pendiente_VMO() : double
+pendiente_maxima_eficiencia() : double
-recogerDesAero()

*Figura 78: Clase Trafico*

**Restriccion**

+aeropuerto : String
+altmantener : double
+altptemin : double
+nombre : String
+pista : String
+ptemin : double
+restr_alt : String

+Restriccion(String nombresid, String aeropuerto, String pista, String altmantener, String ptemin, String altptemin, String restr_alt)
+Restriccion(String nombrestar, String aeropuerto, String pista, String restr_alt)
+getAeropuerto() : String
+getAltmantener() : double
+getAltptemin() : double
+getNombresidstar() : String
+getPista() : String
+getPtemin() : double
+getRestr_alt() : String

*Figura 79: Clase Restriccion*

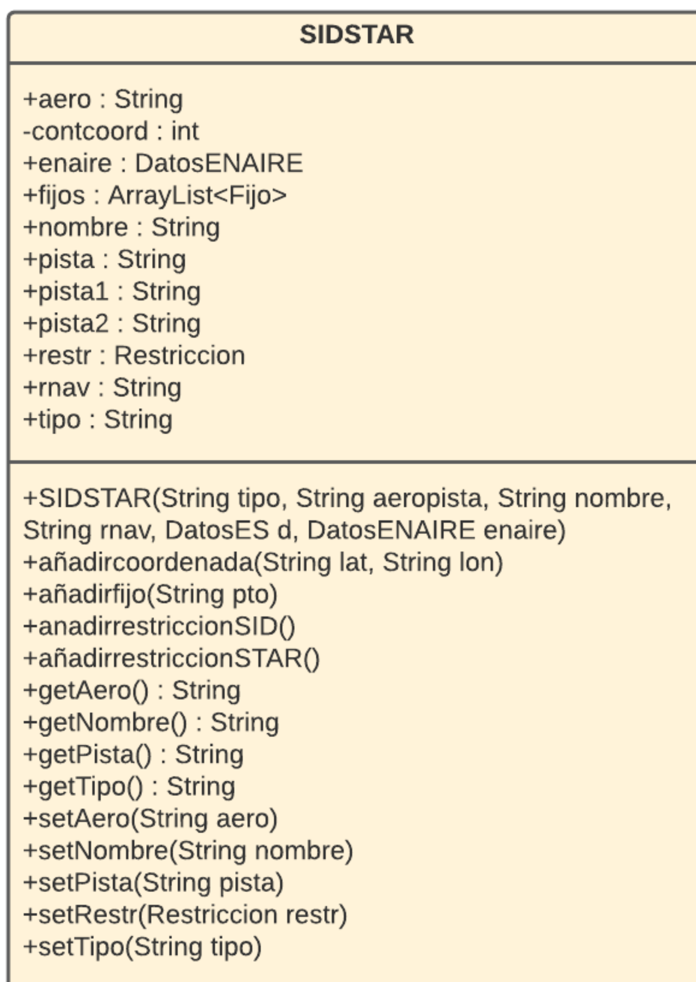


*Figura 80: Clase SIDSTAR*

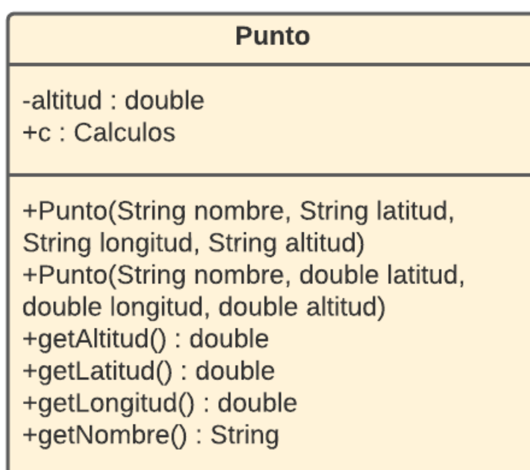


*Figura 81: Clase Punto*

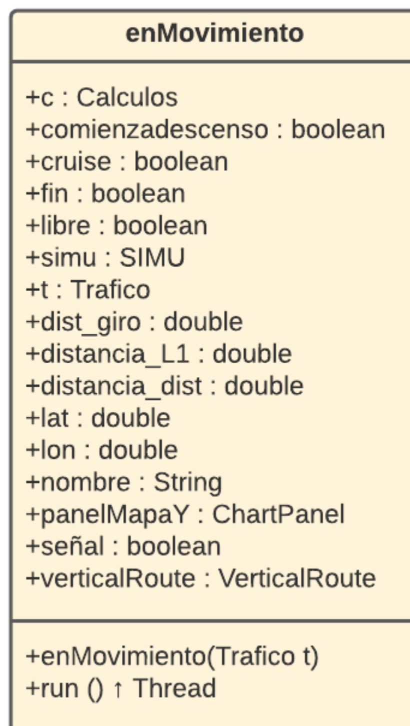


*Figura 82: Clase enMovimiento*

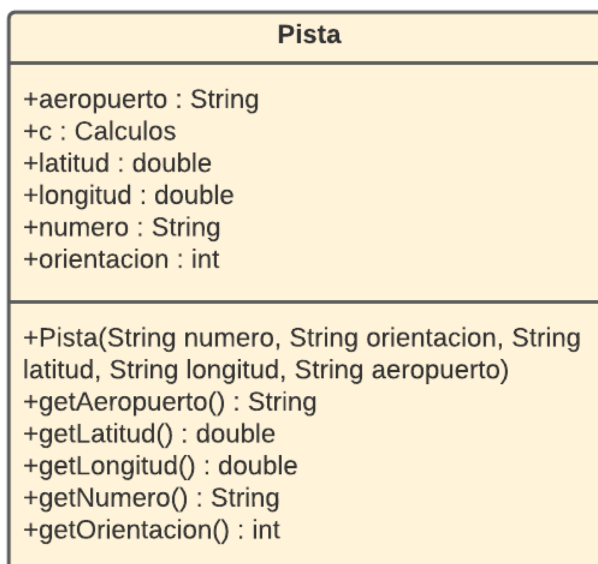


*Figura 83: Clase Pista*

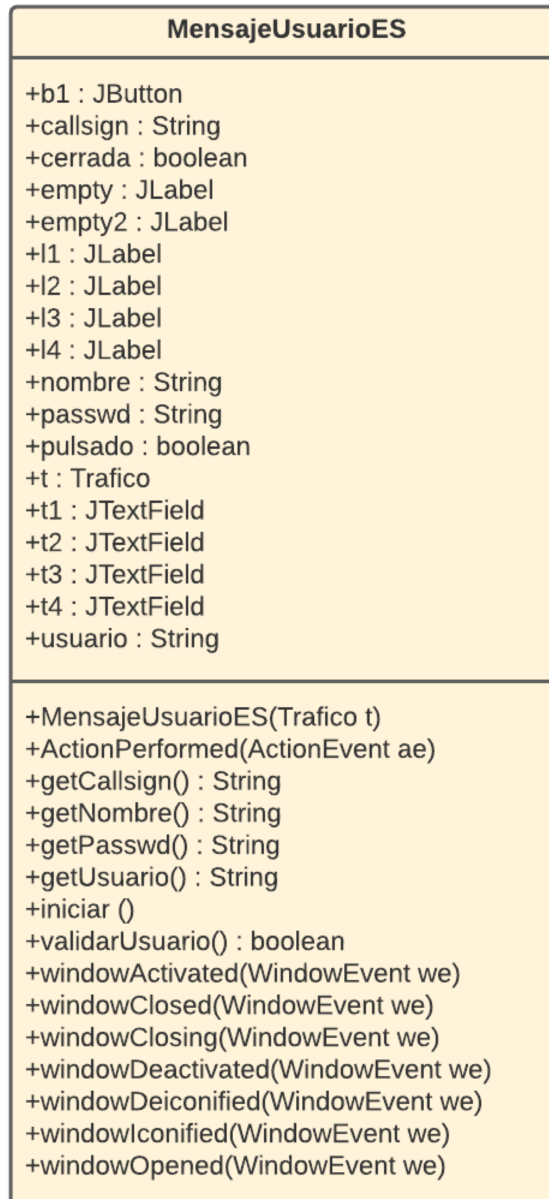


*Figura 84: Clase MensajeUsuarioES*

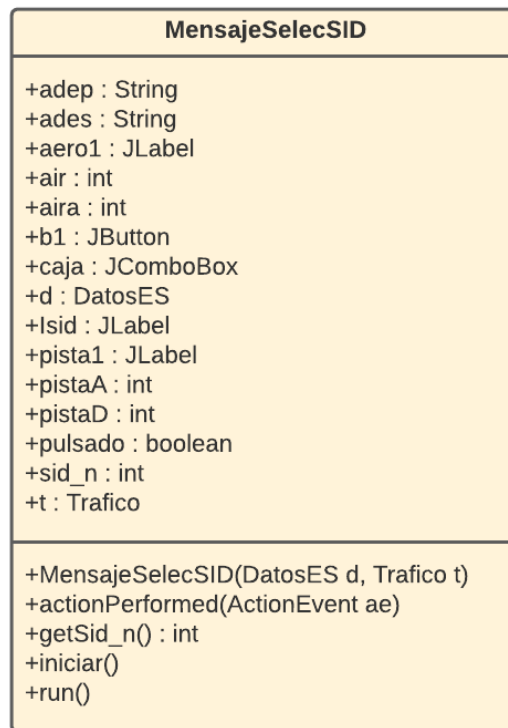


*Figura 85: Clase MensajeSelecSID*

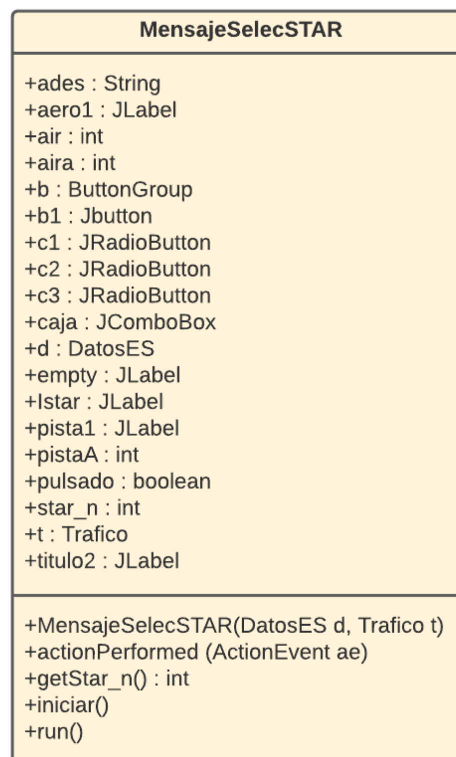


*Figura 86: Clase MensajeSelecSTAR*

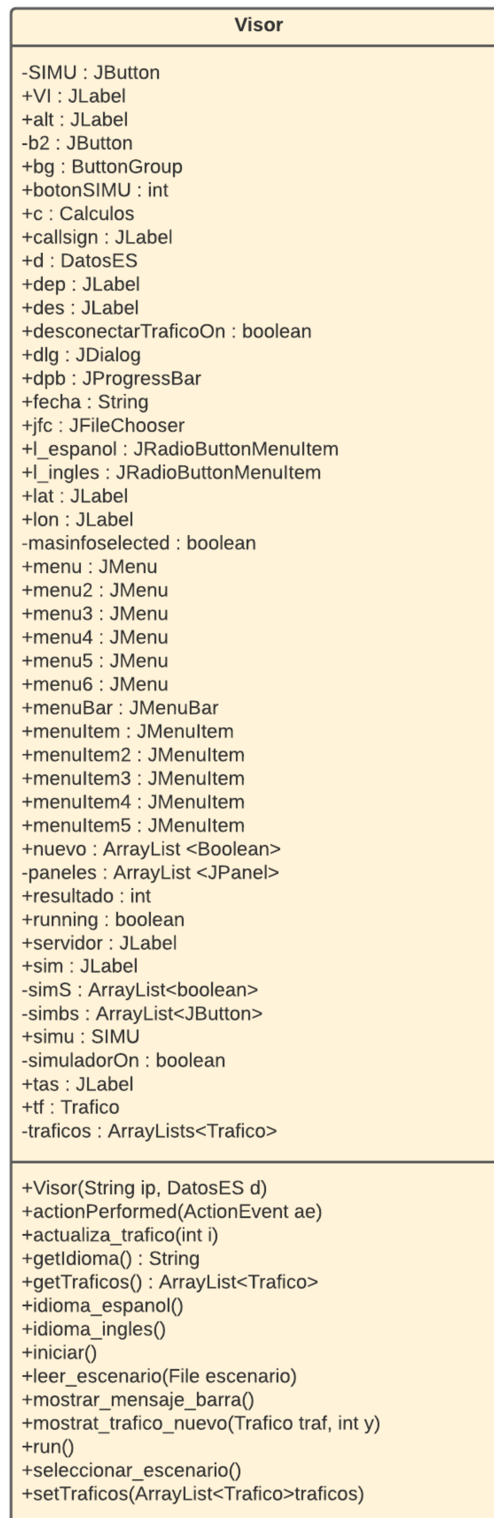


*Figura 87: Clase Visor*

**SIMU**

+Despegue : JButton
+FL : double
+Pausa : JButton
+SID : JButton
+STAR : JButton
+VCAS : JButton
+VS : double
+alt : JLabel
+array_FL : ArrayList<Double>
+b1 : JButtton
+b2 : JButtton
+b3 : JButtton
+b4 : JButtton
+b5 : JButtton
+bmas : Jbutton
+bmenos : Jbutton
+botonHeading : boolean
+botonPlan : boolean
+botonSIDOn : boolean
+botonSTAROn : boolean
+botonVNAV : boolean
+botonVS : boolean
+botonhold : boolean
+bspoilers : JButton
+bwp1 : JButton
+bwp2 : JButton
+bwp3 : JButton
+bwp4 :
+caja1VCAS : JTextField
+caja1VNAV : JTextField
+caja2VCAS : JTextField
+caja2VNAV : JTextField
+caja3VCAS : JTextField
+caja3VNAV : JTextField
+caja4VCAS : JTextField
+caja4VNAV : JTextField
+cajaFL : JTextField
+cajaVS : JTextField
+cajaheading : JTextField
+despegar : boolean
+dir : int
+fin : boolean
+heading : int
+i : int
+l1 : JLabel
+l2 : JLabel
+l3 : JLabel
+lFL : JLabel
+lVS : JLabel
+lat : JLabel
+lon : JLabel
+modoHeadingOn : boolean
+modoPlanOn : boolean
+modoVCASOn : boolean
+modoVNAVOn : boolean
+modoVSOn : boolean
+modoholdOn : boolean
+msid : MensajeSelecSID
+mstar : MensajeSelecSTAR
+panelMapaY : ChartPanel
+paneldatos : JPanel
+panelmapa : OverlayMapPanel
+pausado : boolean
+pte : JLabel
+squawk : JLabel
+t : Trafico
+tVCAS : JTextField
+tituloVCAS : JLabel
+tituloVNAV : JLabel
+trk : JLabel
+tsquawk : JTextField
+vCAS : double
+vcas : JLabel
+verticalRoute : VerticalRoute
+vs : JLabel
+vtas : JLabel
+wptsRuta : ArrayList<Fijo>
+c : Calculos

+SIMU(Trafico t, ArrayList wptsRuta)
+actionPerformed(ActionEvent ae)
+actualizar_datos_trafico()
+iniciar()
+introducirHeading() : int
+run()
+situarwpts()
+validarVCAS() : boolean

*Figura 88: Clase SIMU*

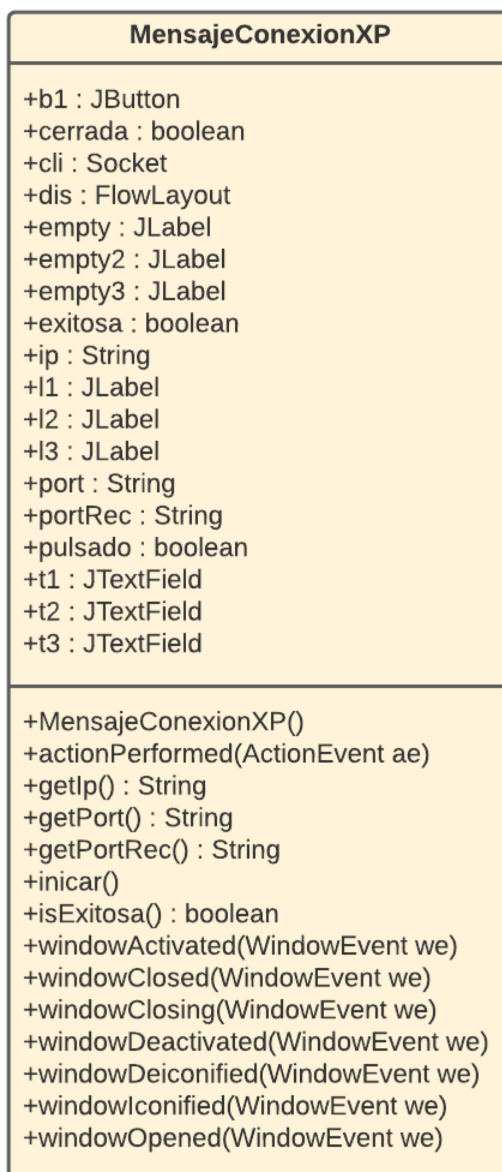


*Figura 89: Clase MensajeConexiónXP*

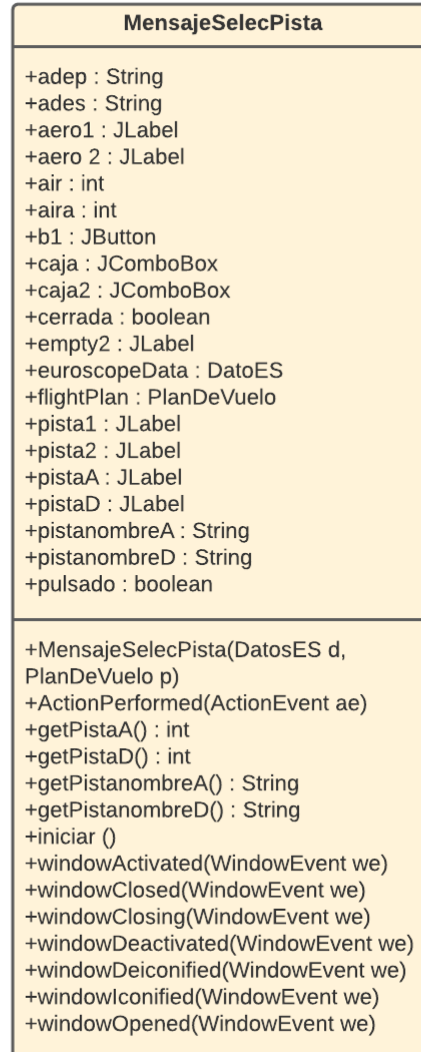


*Figura 90: Clase MensajeSelecPista*

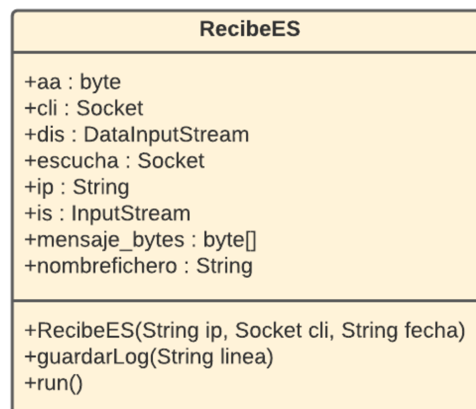


*Figura 91: Clase MensajeRecibeES*

### 3.3.2.Diagrama de clases simplificado

Como podemos observar, en este apartado se muestra el diagrama completo con las clases y sus relaciones, pero sin el contenido de cada una de ellas, para una mayor claridad visual.

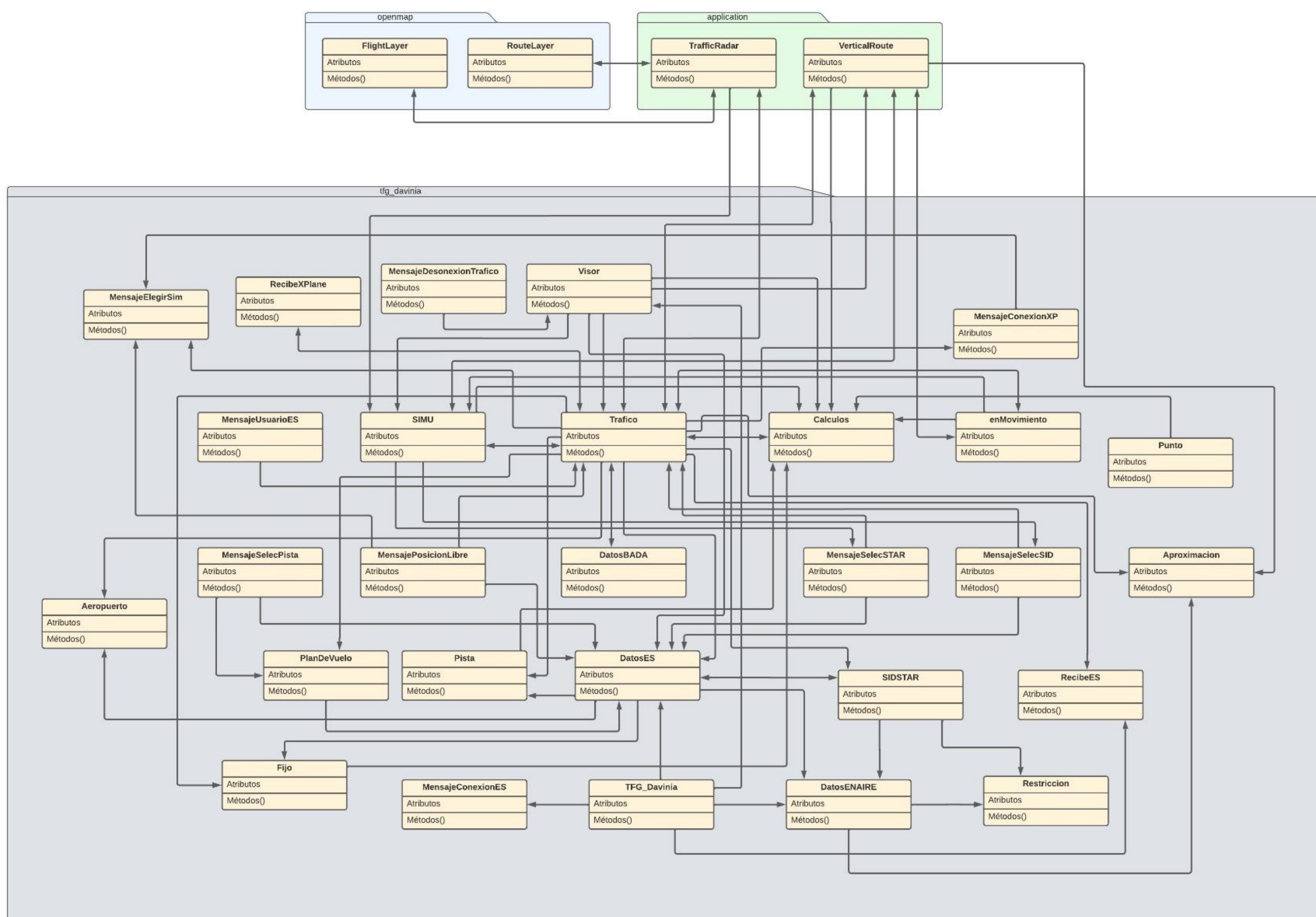


*Figura 92: Esquema UML del proyecto completo*

# 4. Proceso de desarrollo

En este apartado vamos a describir brevemente el proceso que se ha llevado a cabo en este proyecto para alcanzar los objetivos marcados. Dicho proceso consta de una serie de fases que se han ido realizando con tal de asegurarnos de que cumplíamos en todo momento con las necesidades que demandaba el proyecto, intentadlo hacer de la manera más eficiente y sencilla posible. A continuación, describimos detalladamente las fases de las que se ha compuesto nuestro proceso de desarrollo.

## 4.1. Fase preliminar y de aprendizaje

El objetivo de esta primera fase, fue ver nuestro punto de partida. Recordemos que este proyecto trata sobre la mejora e implementación de nuevas opciones y herramientas dentro de un programa ya creado, por lo que no partíamos de cero. Nuestro primer objetivo fue ver de qué recursos disponíamos, además de leer y entender el contenido de la memoria que hizo la primera desarrolladora del programa, con tal de tener una visión general del proyecto que teníamos entre manos, pero para ello necesitábamos entender lo que leíamos, ya que eran temas totalmente nuevos.

Para ello, después de una primera leída al proyecto del que partimos, vimos que nuestro trabajo consistía en un desarrollo software basado en el lenguaje Java y en la POO, y que además tenía que ver con el sector aeroespacial. Dicho esto, el objetivo principal era documentarnos a fondo sobre todo ello, ya que nunca habíamos estudiado ni realizado ningún proyecto de esta índole.

Dicho todo lo anterior, esta primera fase se dividió en dos grandes sub-fases que tienen que ver con el proceso de aprendizaje de dos campos muy diferentes: terminología aeroespacial y el lenguaje Java y la POO.

Esta fase fue una de las más densas y duraderas, pero a la vez una de las más importantes para poder seguir adelante con el proyecto con la soltura suficiente como para hacer un buen trabajo.

### 4.1.1. Sub-fase de aprendizaje sobre terminología aeroespacial

En esta primera sub-fase, nos adentramos de lleno en el sector aeroespacial, en concreto dentro del campo del control de tráfico aéreo. Aprendiendo un sinfín de siglas que aparecen casi a diario dentro del sector (TOD, VOR, SID, STAR, VCAS, VTAS, VNAV, LNAV, etc.), estudiando también cuáles son las fases de un avión, de cuantas partes se compone un descenso, tipos de ascenso y descenso, las ecuaciones que rigen las dinámicas de un avión, las superficies de sustentación que se utilizan (spoilers, flaps, entre otros), las

unidades de cada una de las medidas (ft, NM, knots, fpm, etc.), , entre otros muchos conceptos. Además, era de extremada importancia familiarizarse con los conceptos dentro del manual de BADA, sus ecuaciones y sus coeficientes y de qué archivos se puede disponer para la recogida de los datos que BADA proporciona.

### 4.1.2. Sub-fase de aprendizaje de Java y POO

Con la primera sub-fase bastante asentada, empezamos con la segunda; empezábamos de 0 en Java y en la programación orientada a objetos. Empezamos con tutoriales de YouTube, buscando información de diversas páginas con ejemplos de todo tipo, desde lo más básico (tipos de datos, declaración de variables, bucles, etc.) hasta algo más avanzado (creación de clases, métodos, declaración de atributos, instanciación, concepto de herencia, polimorfismo, encapsulamiento, etc.). Una vez obtenida una visión global de lo que es la programación orientada a objetos en Java, y con una gran variedad de ejemplos hechos, nos vimos capaces de pasar a la siguiente fase.

## 4.2. Fase de lectura de código

En esta fase nuestro objetivo era saber más concretamente desde donde partíamos a nivel de código, pero también tener una visión, aunque aún globalizada, más concreta del interior del programa. Leímos unas cuatro veces el código de arriba abajo, analizando y entendiendo cada clase, los métodos de los que se componía cada clase y viendo qué hacía cada una de ellas.

Además del código, a este software lo acompañaban archivos de texto, que hacían de bases de datos que posteriormente se utilizaban en el programa para simular las dinámicas de cualquier aeronave y su ruta. Estos archivos son los archivos de extensión ya comentados con anterioridad: .APF, .OPF, .PTF, GPF, .sct y .txt. A pesar de no ser código, su lectura se incluyó dentro de esta fase.

La gran ventaja de la programación (y en ocasiones como esta, inconveniente), es que una misma funcionalidad es posible implementarla de mil maneras distintas. Es por ello que esta fase fue una de las más complicadas al principio, ya que cada persona tiene un estilo al programar, una filosofía y una forma de hacer las cosas y organizarlas totalmente diferente a otra. Esto lleva a que entender el código y saber por ejemplo qué quiere decir X variable, a veces se convierta en un verdadero quebradero de cabeza, porque tal vez otra persona o uno mismo lo hubiera hecho totalmente diferente. Entender y conocer el funcionamiento de cada clase y reconocer sus variables y su utilidad fue un verdadero reto.

## 4.3. Fase de planificación y análisis

En este punto ya habíamos "manoseado" bastante el código y conceptualmente estaba claro. Era el momento de reunirse con el tutor y proponer diferentes mejoras y opciones a implementar en el programa.

El objetivo de esta fase simplemente era tener claro qué es lo que queremos que hagan las nuevas opciones del programa. Finalmente, con ayuda del tutor, se establecieron las bases de los principales objetivos. Ahora había que analizar cada uno de ellos, es decir, saber con exactitud qué es lo que va a hacer dicha parte del programa, qué es lo que se necesita y cuáles son los requerimientos del sistema (las características que debe poseer) en cada una de las mejoras que se querían realizar.

## 4.4. Fase de diseño

En esta fase teníamos que estudiar qué recursos y qué caminos íbamos a tomar para llevar a cabo cada una de las funcionalidades. Debíamos descubrir y decidir las características que el sistema debía poseer para llegar a la solución que queríamos obtener.

Por ejemplo, como ya es bien sabido, uno de los requerimientos era la creación de un sistema que representara el perfil vertical de una aeronave a tiempo real. Para ello sabíamos que necesitábamos una gráfica que representara, al fin y al cabo, puntos. Se consideraron diferentes opciones; una de ellas fue utilizar la librería *OpenMaps*, una conocida librería que facilita bastante la creación de aplicaciones que requieren representación geográfica. Durante un tiempo se replanteó todo para poder utilizar *OpenMaps*, (se estudió su viabilidad y muchas de sus funciones) incluso se llegó a crear una interfaz inicial con *OpenMaps*, pero llegó un punto en el que nos dimos cuenta que dicha librería sólo computaba datos geográficos en términos de longitud y latitud; en nuestro caso iba a ser distancia y altitud, y el formato de las funciones dentro de dicha librería nos impedía el uso de la misma. Por tanto, después de estudiar otras opciones optamos por la librería *JFreeChart*.

En otros casos, como el tema de la opción del indicador de despliegue de spoilers *SPEED BRAKE ARMED*, la solución inicial era la adecuada, pero incluso con una buena solución desde el principio, habría más adelante que refinar la idea, y en la fase de diseño se tuvo que pensar bien sobre todo a nivel interfaz, dónde situar el indicador de aviso, para que no pareciera que está muy desplazado de la parte de control de modos de vuelo, pero con cuidado de no juntarlo demasiado para no incluirlo involuntariamente en dentro de algún modo de vuelo. Finalmente se optó por dejarlo en un sitio tal que a la vista tuviera fácil acceso pero que a la vez no se confundiera con ningún modo de vuelo.

A nivel de código se pensó también en qué clases habría que modificar, cuántas clases habría que crear y si habría que borrar o reescribir algo del código original. Por ejemplo, para la parte de la creación del perfil vertical, se creó una clase sólo dedicada a ello, llamada "*VerticalRoute*", en la que se implementa y se configura todo lo relacionado con la gráfica *JFreeChart*.

Por otra parte, una de las principales clases que se tuvo que modificar fue la clase *enMovimiento*, principalmente para remodelar el movimiento del avión en el descenso, de forma que descendiera acorde a las restricciones de la ruta. Al final lo que se buscó dentro de dicha clase fue mejorar la dinámica de descenso, ya que la algoritmia que describe dicha fase era desde un principio muy simple, y no tenía en cuenta el correcto cumplimiento de todas las restricciones. Además, tampoco tenía en cuenta el movimiento que debía llevar el avión en caso de tener un descenso escalonado. En el programa inicial no se contemplaba el caso en el que la ruta tuviera dos TODs, por lo que las restricciones dentro de todo el tramo tampoco las tenía en cuenta correctamente. Nuestro principal objetivo dentro de dicha clase fue rediseñar la dinámica para que el avión tuviera en cuenta ese tipo de situaciones y que fuera capaz de actuar en consecuencia.

En este tipo de modificaciones también han sido importantes clases como *SIMU, Trafico o Calculos.*

## 4.5. Fase de implementación y testeo

Sin duda la fase más larga. En esta fase, se empezó a realizar lo establecido en la fase de diseño; se intentó realizar de la forma más limpia y correcta posible. Es decir, se hizo con el objetivo de que el código no fuera indescifrable, comentando lo máximo posible cada método, pero manteniendo a su vez, la lógica y filosofía de Davinia. Se intenta también obtener una buena interpretación visual del código, estando lo mejor estructurado posible. Obedeciendo también a los estándares de programación no necesarios, elegimos hacer uso tanto de la notación "*Lower Camel Case*" para métodos (un ejemplo puede ser el método llamado *updatePlaneMovement()),* como de la notación "*Upper Camel Case*" para las clases (ejemplo de ello es la clase llamada "*VerticalRoute*"). Por otra parte, para las variables usamos generalmente la notación snake_case (es el caso de por ejemplo la variable declarada como *double dist_invert[]).*

Además, cabe destacar que, en esta fase, con tal de saber que íbamos "por el buen camino", desarrollamos casos de prueba o puntos de análisis, ejecutando y testeando que el programa funcionaba adecuadamente hasta donde se había llegado. Esto es una buena praxis sobre todo en programas de gran tamaño para asegurarse de que el software funciona correctamente al

menos de forma parcial. En todo caso cuando posteriormente se testea el programa en conjunto, es más fácil detectar los fallos si es que los hay.

## 4.6. Fase de puesta a punto

Finalmente, una vez hechas todas las pruebas necesarias, se dieron las últimas pinceladas al *software*, como últimos comentarios en el código, revisión global del mismo, correcto funcionamiento de la ejecución, etc. Una vez se comprobó que todo estaba en orden y funcionaba perfectamente, se dio por concluido el desarrollo del *software*.

Cabe tener en cuenta que aun que se hayan explicado todas las fases por separado, no todas se iniciaban cuando terminaba la anterior. En muchos casos una fase podía empezarse, aunque la anterior no se hubiera acabado ya que, a pesar de no poder completarse, sí que se intentaba avanzar lo máximo posible con lo que ya se tenía de la fase previa. Aun así, al empezar una fase nueva siempre se intentaba tener lo máximo acabado de las fases anteriores.

# 5. Estudio económico

## 5.1. Diagrama de Gantt

Uno de los puntos requeridos en la memoria, es el estudio económico. Pero, puesto que este proyecto es meramente académico y no responde a ningún tipo de contrato ni especificaciones por parte de ninguna empresa, vamos a suponer para este punto, que tenemos que dar un presupuesto a una empresa la cual quiere realizar un proyecto exactamente igual a este. Nosotros, tenemos que darle un presupuesto a dicha empresa, para que ellos posteriormente decidan si aceptan nuestra propuesta económica o no.

Para ello, hemos realizado, previo al análisis, un diagrama de Gantt, donde representamos y justificamos en forma de tabla el tiempo que hemos invertido en cada fase del desarrollo. De esta forma plasmamos el tiempo de trabajo por tarea realizada en una única tabla. Para este punto nos es útil el punto cuatro, donde explicamos con todo detalle cada una de las fases que contiene el proyecto. El diagrama de Gantt es el siguiente:

| Diagrama de Gantt | | | | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| ACTIVIDAD | | 2021 | | | | | | | | | | | | 2022 | | | | | |
| | | ENERO | FEBRERO | MARZO | ABRIL | MAYO | JUNIO | JULIO | AGOSTO | SEPTIEMBRE | OCTUBRE | NOVIEMBRE | DICIEMBRE | ENERO | FEBRERO | MARZO | ABRIL | MAYO | JUNIO |
| Fase preliminar y de aprendizaje | Sub-fase de aprendizaje sobre terminología aeroespacial | | | | | | | | | | | | | | | | | | |
| | Sub-fase de aprendizaje de Java y la POO | | | | | | | | | | | | | | | | | | |
| Fase de lectura de código | | | | | | | | | | | | | | | | | | | |
| Fase de planificación y análisis | | | | | | | | | | | | | | | | | | | |
| Fase de diseño | | | | | | | | | | | | | | | | | | | |
| Fase de implementación y testeo | | | | | | | | | | | | | | | | | | | |
| Fase de puesta a punto | | | | | | | | | | | | | | | | | | | |
| Realización de la memoria | | | | | | | | | | | | | | | | | | | |

*Tabla 3: Diagrama de Gantt del proyecto*

## 5.2. Presupuesto

Vamos a considerar por simplicidad a lo largo de todo el estudio económico, ciertas condiciones constantes a lo largo de todo el cálculo. En primer lugar, consideraremos que un mes son 30 días, de los cuales laborables son 22 y que se trabaja de lunes a viernes 6 horas diarias. Además, supondremos que nuestro analista desarrollador cobra 14,62 €/h. Por otra parte, los servicios también van a tener un valor constante; supondremos que todos los meses gastamos lo mismo de luz y que todos los meses la electricidad está al mismo precio. Por otra parte, el plan de internet va a ser siempre el mismo y en cuanto a la gasolina (transporte), todos los meses va a valer lo mismo. Dicho esto, hemos adjudicado que el coste por hora del material hardware, es de unos 0.75€, que el coste unitario por kW de energía es de 0.20€, que el plan de internet es de 15€/mes y que el precio por litro de combustible es de 1.68€. Para contabilizar el

precio total del transporte, asumiremos también que cada desplazamiento le cuesta al analista unos 2’8 litros. Esto último implica que, si un mes el analista ha tenido que desplazarse 4 días el primer mes, en total ha gastado unos 11 litros de gasolina. Con estos datos ya podemos realizar nuestro presupuesto:

Presupuesto

| COMPONENTE | 2021 | | | | | | | | | | | | 2022 | | | | | | CANTIDAD | COSTO UNITARIO | COSTO TOTAL |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | ENERO | FEBRERO | MARZO | ABRIL | MAYO | JUNIO | JULIO | AGOSTO | SEPTIEMBRE | OCTUBRE | NOVIEMBRE | DICIEMBRE | ENERO | FEBRERO | MARZO | ABRIL | MAYO | JUNIO | | | |
| MANO DE OBRA | | | | | | | | | | | | | | | | | | | | | |
| Analista desarrollador de software | 132 h | 132 h | 132 h | 132 h | 132 h | 132 h | 132 h | 132 h | 132 h | 132 h | 132 h | 132 h | 132 h | 132 h | 132 h | 132 h | 132 h | 132 h | 2376 h | 14.62 € | 34,737.12 € |
| HARDWARE | | | | | | | | | | | | | | | | | | | | | |
| Uso informático de computadora más complementos (ratón y teclado) | 132 h | 132 h | 132 h | 132 h | 132 h | 132 h | 132 h | 132 h | 132 h | 132 h | 132 h | 132 h | 132 h | 132 h | 132 h | 132 h | 132 h | 132 h | 2376 h | 0.75 € | 1,782.00 € |
| SOFTWARE | | | | | | | | | | | | | | | | | | | | | |
| NetBeans 8.2. | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 |
| DropBox | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 |
| Microsoft Teams | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 |
| PoliformaT | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 |
| Lucidchart | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 |
| GASTOS INDIRECTOS | | | | | | | | | | | | | | | | | | | | | |
| Energía Eléctrica | 15 kW/h | 15 kW/h | 15 kW/h | 15 kW/h | 15 kW/h | 15 kW/h | 15 kW/h | 15 kW/h | 15 kW/h | 15 kW/h | 15 kW/h | 15 kW/h | 15 kW/h | 15 kW/h | 15 kW/h | 15 kW/h | 15 kW/h | 15 kW/h | 273.60 kW/h | 0.20 € | 54.72 € |
| Internet | 1 plan | 1 plan | 1 plan | 1 plan | 1 plan | 1 plan | 1 plan | 1 plan | 1 plan | 1 plan | 1 plan | 1 plan | 1 plan | 1 plan | 1 plan | 1 plan | 1 plan | 1 plan | 18 | 15.00 € | 270.00 € |
| Transporte | 11 L | 6 L | 3 L | 0 L | 0 L | 0 L | 3 L | 6 L | 0 L | 0 L | 0 L | 3 L | 0 L | 3 L | 0 L | 0 L | 0 L | 0 L | 34 L | 1.68 € | 56.45 € |
| | | | | | | | | | | | | | | | | | | | | Sub Total | 36,900.29 € |
| | | | | | | | | | | | | | | | | | | | | 10% imprevistos | 3,690.03 € |
| | | | | | | | | | | | | | | | | | | | | 25% ganancia | 9,225.07 € |
| | | | | | | | | | | | | | | | | | | | | 21% IVA | 7,749.06 € |
| | | | | | | | | | | | | | | | | | | | | Total | 57,564.45 € |

*Tabla 4: Presupuesto del proyecto*

Si observamos los resultados finales, aplicándole al subtotal un 10% de imprevistos (problemas técnicos, de desplazamiento, recambios de material, etc.), un 25% como ganancia y el respectivo IVA, el total del presupuesto asciende a 57.564’45€.

# 6. Conclusión

A modo de cierre de este proyecto, podemos señalar que este trabajo fin de grado, tenía como principales objetivos el desarrollo y la mejora de un software que funciona como simulador de tráfico aéreo. Para cumplir con dicho desarrollo, hizo falta una primera etapa de aprendizaje y comprensión de campos como la programación en Java, y el sector aeroespacial; de los cuáles hasta el momento desconocíamos por completo. Finalmente conseguimos desarrollar las mejoras suficientes como para hacer del programa una herramienta mucho más útil, mostrando una gran cantidad de información acerca del tráfico o tráficos que estemos analizando en el momento.

El resultado más relevante en este proyecto, ha sido sin duda la elaboración de los algoritmos necesarios para hacer posible la correcta simulación dentro de la gráfica *JFreeChart*. Dichos algoritmos, tenían que ver principalmente con el rediseño de la dinámica del avión sobre todo en una de las fases más críticas: el descenso. El programa inicial no contaba con la capacidad de detectar en qué momento la ruta no cumple con las restricciones y tomar acción. El descenso era un modelo mucho más simple, y nosotros hemos sido capaces de mejorarlo, haciendo que, de forma automática, el programa rehaga la ruta de descenso en caso de no cumplir con alguna restricción, modificando también el comportamiento del avión. Un objetivo así requería de un nivel de comprensión del programa previo bastante alto, puesto que era imposible implantar algo así sin antes entender la estructura del programa y la filosofía de trabajo que se ha ido llevando dentro de este.

En cuanto a la elaboración de este proyecto, creemos que desde un principio supimos llevar una buena organización de las tareas a llevar a cabo. Podría decirse pues, que la metodología de trabajo fue la acertada, ya que finalmente conseguimos los objetivos marcados haciendo caso de nuestra planificación inicial. La piedra angular en la organización de las tareas, fue sin duda repartir estas de la mejor forma posible, de tal manera que una tarea estuviera lo más terminada posible, así luego no tendríamos que volver atrás a rehacer trabajo de tareas anteriores, lo que hubiera dificultado y atrasado en gran medida la evolución del proyecto.

No obstante, personalmente, el resultado más relevante podemos decir que ha sido la gran experiencia y conocimiento sobre todo en el campo de la programación que este proyecto nos ha brindado. Gracias a la cantidad de tiempo invertido en programar, conforme van pasando las semanas uno mismo puede darse cuenta de lo mucho que acaba aprendiendo de una disciplina como esta. Una disciplina compleja, la cual requiere un gran nivel de abstracción, pero que a la vez aporta una capacidad inmensa a la hora de abordar cualquier problema de una

forma mucho más eficiente. Además, ha sido una experiencia muy divertida y enriquecedora, sobre todo por el hecho de poder aprender y a la vez aplicar en el programa conceptos sobre el sector aeroespacial. Aspectos como las ecuaciones que rigen los movimientos de una aeronave, aprender sobre algo de aerodinámica, componentes y comportamiento del avión, y también cómo funciona en gran media el espacio aéreo y su control.

# 7. Ampliaciones y desarrollo futuro

Este proyecto deja las puertas abiertas a una gran cantidad de ideas de mejora. Ya en su primera versión, podían verse muchos caminos por lo que poder investigar mejoras de la aplicación. Sin embargo, se nombrarán únicamente algunos de los posibles proyectos futuros a partir del camino que se eligió para desarrollar este proyecto.

Por ejemplo, en cuanto al archivo .PTF, se podría estudiar la manera de poder leerlo y almacenar los datos como nos interese. De esa forma, se podría reducir el error que pueda haber entre la ruta teórica y el movimiento real del avión, ya que la ruta teórica se crearía a partir del mismo modelo de avión que el introducido en el plan de vuelo.

Una de las grandes mejoras que podrían implantarse podría ser, diseñar el programa, para que fuera capaz de mostrar los datos de velocidad (VCAS) y altitud (VNAV) que el avión va a tomar en cada *waypoint*, tal y como se muestra en la siguiente figura.

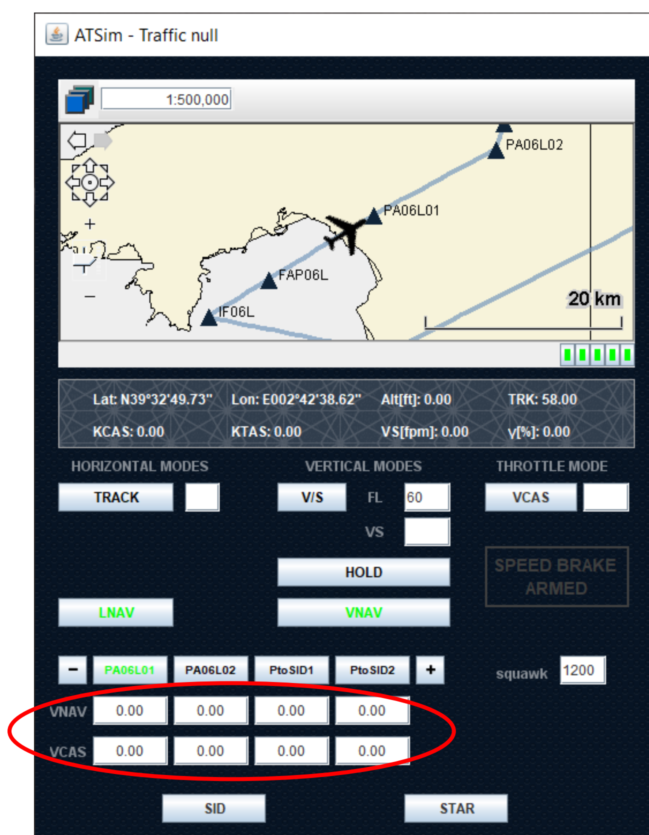


*Figura 93: Ejemplo de mejora del programa*

El programa podría ser capaz de mostrar inicialmente los datos que teóricamente va a llevar el avión, con la posibilidad de introducir nuevos valores

de VCAS y/o VNAV en un *waypoint* en concreto. De esa forma toda la ruta junto con la propia dinámica del avión se debería modificar automáticamente, para cumplir con la/las condición/es introducida/s en dicho *waypoint*.

En cuanto a la gráfica *JFreeChart*, incorporar algún elemento gráfico que permitiese desplazar el contenido del gráfico horizontalmente, como un *ScrollBar*, aumentaría la comodidad del usuario a la hora de usar dicha gráfica. Otra opción podría ser, hacer que el avión siempre se mantuviera visible en el panel, de esa forma el usuario nunca tendría que estar pendiente de si el avión ya no es visible y verse obligado en ese momento a hacer zoom para volver a visualizarlo.

Otra pequeña mejora podría ser, mejorar el aspecto general de la interfaz, dándole un aspecto más actualizado, o incluso una mejor distribución de los elementos dentro de la misma.

Todas estas ideas, dejan abiertas las puertas a futuros desarrollos dentro del alumnado interesado en este tipo de proyectos de la Escuela Técnica Superior de la Ingeniería del Diseño.

# 8. Bibliografía


1. https://stackoverflow.com/
2. https://www.youtube.com/
3. https://www.jfree.org/
4. https://codereview.stackexchange.com/questions/251735/great-circle-distance-in-java
5. https://forums.x-plane.org/index.php?/forums/topic/154948-speed-brake-do-not-armed-is-missing/
6. http://dar10comyr.blogspot.com/
7. https://www.sociedadaeronautica.org/
8. https://www.enaire.es/home
9. http://www.movable-type.co.uk/scripts/latlong.html
10. https://www.lucidchart.com/pages/es
11. http://ingluisfransv.blogspot.com/2015/07/elaborar-el-presupuesto-para-un.html
12. http://www.edwilliams.org/avform147.htm#Intermediate
13. http://puntocomnoesunlenguaje.blogspot.com/p/teoria_7.html
14. https://www.figma.com/?fuid=
15. https://en.wikipedia.org/wiki/Main_Page
16. https://www.tutiempo.net/calcular-distancias.html
17. https://codereview.stackexchange.com/questions/251735/great-circle-distance-in-java
18. https://docs.oracle.com/javase/7/docs/api/
19. https://www.manualvuelo.es/index.html

20. https://www.ifiseducacion.com/blog/6-errores-la-hora-de-redactar-nuestro-tfg-o-tfm

21. https://es.talent.com/salary?job=programador+.net

22. Recursos de la asignatura de grado Oficina Técnica.

23. Tutorías con el tutor Pedro Yuste Pérez.