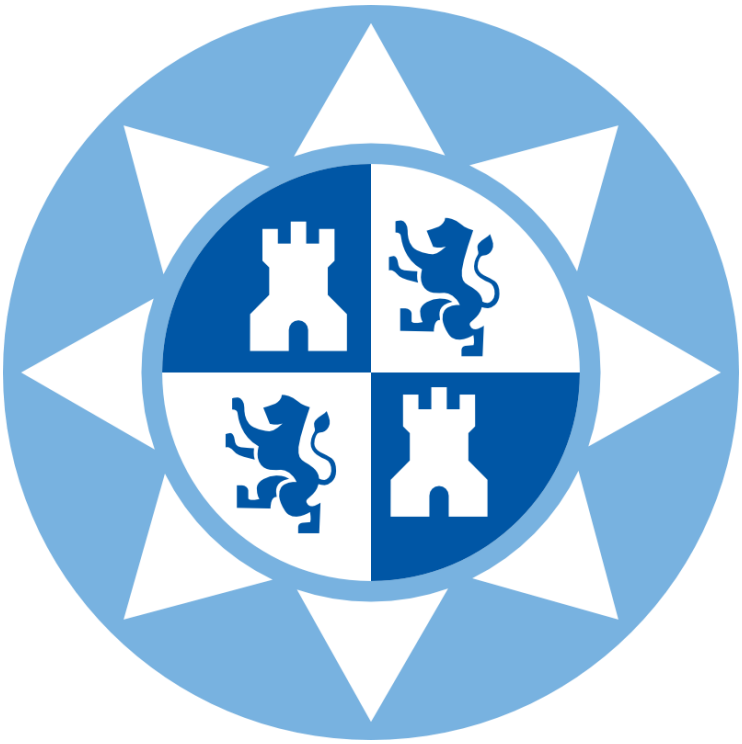
Universidad Politécnica de Cartagena

Escuela Técnica Superior de Ingeniería de Telecomunicaciones



Proyecto Final de Carrera

Aplicación Java para controlar y supervisar el simulador de vuelo FlightGear

Alumno: Víctor Pérez García
D.N.I.:23027389-L
Directores: D. Diego Alonso Cáceres
D. Antonio Manuel Padilla Urrea
09 / 2014

Universidad Politécnica de Cartagena

Autor	**Víctor Pérez García**
E-mail del Autor	**Victor_p.83@hotmail.com**
Director(es)	**D. Diego Alonso Cáceres, D. Antonio Manuel Padilla Urrea**
E-mail del Director	**diego.alonso@upct.es, antonio.padilla@upct.es**
Codirector(es)	
Título del PFC	**Aplicación Java para supervisar y controlar el simulador de vuelo FlightGear**
Descriptores	

Resumen

En el presente proyecto se pretende crear una aplicación en lenguaje de programación Java para controlar y supervisar el simulador de vuelo FlightGear de forma eficiente.

Esta aplicación se encargará de recibir los datos de vuelo del simulador, así como los controladores o actuadores de dichos datos, para así poder modificarlos manualmente o mediante una serie de controladores PID. Una vez modificados, la aplicación los enviará de nuevo al simulador para que éste realice la maniobra requerida o siga en vuelo recto y nivelado.

La supervisión se realiza de forma que los datos recibidos que consideramos más importantes son imprimidos en la interfaz gráfica de usuario y utilizados para representar o pintar varias gráficas.

Titulación	**Ingeniería Técnica de Telecomunicaciones, Especialidad en Telemática**
Intensificación	
Departamento	**Tecnologías de la Información y Comunicaciones**
Fecha de Presentación	**09 - 2014**

ÍNDICE

Capítulo 1. Introducción 7

1.1. Motivación 7

1.2. Objetivos 8

1.3. Estructura de la memoria 8

1.4. Otras iniciativas similares con FlightGear 9

Capítulo 2. FlightGear, maniobras básicas de aviación y teoría de control 12

2.1. FlightGear 12

2.1.1 Modelos Dinámicos de Vuelo 13

2.1.2 Otras características 19

2.2. Maniobras fundamentales en aviación 19

2.2.1. Potencia y actitud 22

2.2.2. Ascensos 23

2.2.3. Descensos 24

2.2.4. Giros 26

2.2.5. Vuelo recto y nivelado 27

2.3. Teoría de Control 28

2.3.1. Historia 29

2.3.2. Componentes básicos de un sistema de control 31

2.3.3. El regulador PID 33

2.3.4. Ajustando el regulador PID 37

2.3.5. Discretización de controladores PID 38

Capítulo 3. Descripción de la solución 42

3.1. Protocolo de comunicación con FlightGear 42

3.1.1. Implementación del protocolo de comunicación 43

3.2. Tipos de comunicación 47

3.3. Estructura de los datagramas 51

3.4. Estructura de clases junto con diagramas UML 54

3.5. Interfaz gráfica 60

3.5.1. Implementación de la GUI 60

3.5.2. Implementación de las gráficas 63

3.6. Diseño de los controladores 65

3.6.1. Implementación de los reguladores PID 66

3.6.2. Ajuste de los PID 68

3.7. Problemas encontrados 68

Capítulo 4. Conclusiones y líneas futuras 71

4.1. Conclusiones y experiencia 71

4.2. Trabajos futuros 71

ANEXO I. CÓDIGO FUENTE.....73

ANEXO II. ESPECIFICACIONES DEL AVIÓN ELEGIDO 102

ANEXO III. INSTALACIÓN DE SOFTWARE 104

BIBLIOGRAFÍA.....111

ÍNDICE DE FIGURAS

Figura 1. Esquema del proyecto de la UPC.....10
Figura 2. Álvaro López (TCPSI) y Patricio Gómez (GISD-UPM).....10
Figura 3. Arquitectura del Sistema SISCANT.....11
Figura 4. Imagen del simulador de vuelo FlightGear. 12
Figura 5. Logo de FlightGear.....12
Figura 6. Parte de un archivo XML para la configuración JSBSim de un F16.....15
Figura 7. Parte del cuestionario Aeromatic.....16
Figura 8. Editor JSBSim Commander.....17
Figura 9. Parte del archivo XML de configuración de YASim.....18
Figura 10. En blanco los cambios realizados en el fichero de configuración XML.....18
Figura 11. Los tres ejes de un avión. 20
Figura 12. Movimiento de cabeceo. 20
Figura 13. Movimiento de alabeo. 21
Figura 14. Movimiento de guiñada. 21
Figura 15. Actitudes de morro. 23
Figura 16. Actitudes de alabeo. 23
Figura 17. Potencia y velocidad en el ascenso. 24
Figura 18. Potencia y velocidad en el descenso. 25
Figura 19. Fuerza de sustentación. 26
Figura 20. Descomposición de la fuerza de sustentación al alabear el avión. 27
Figura 21. Relación entre ángulo de ataque y velocidad en vuelo recto y nivelado 28
Figura 22. Regulador de velocidad de James Watt. 29
Figura 23. Componentes básicos de un sistema de control. 31
Figura 24. Sistema de control en lazo abierto. 31
Figura 25. Sistema de control en lazo cerrado. 32
Figura 26. Sistema de control con regulador PID. 33
Figura 27. Constante integral y derivativa. 35
Figura 28. Controlador proporcional. 36
Figura 29. Controlador Proporcional-Integral (PI). 36
Figura 30. Controlador Proporcional-Derivativo (PD). 37
Figura 31. Controlador PID. 37
Figura 32. Ajuste de Ziegler y Nichols (dos primeras) y del método del relé. 38
Figura 33. Sistema de control digital o control por computador. 39
Figura 34. Muestreo cada T segundos. 39
Figura 35. Bloque retenedor.....39
Figura 36. Comunicación entre FG y la aplicación. 42
Figura 37. Opciones de Entrada/Salida de FG. 43
Figura 38. Opciones de Entrada/Salida de FG eligiendo "file" en Medio de comunicación.....47
Figura 39. Opciones de Entrada/Salida de FG eligiendo la opción "serial" en Medio. 48
Figura 40. Protocolo OpenGC en FlighGear.....50
Figura 41. Silla de movimiento sincronizada con un simulador de vuelo. 51
Figura 42. Captura de tramas, la primera contiene a native-fdm. 52
Figura 43. Datagrama net_fdm.....53

Figura 44. Captura de la trama native-ctrls. 53
Figura 45. Datagrama que contiene a net_ctrls. 54
Figura 46. Captura de trama native-ctrls que envía la aplicación al simulador de vuelo. 54
Figura 47. Diagrama UML de la aplicación. 55
Figura 48. Hilo flightGearController y su interfaz Iflightgear. 56
Figura 49. Diagrama UML con las clases encargadas de la entrada/salida de datos. 57
Figura 50. Diagrama UML de la interfaz gráfica de usuario. 58
Figura 51. Diagrama UML con los PID encargados de controlar el avión. 59
Figura 52. Interfaz Gráfica de Usuario. 61
Figura 53. Gráficas FlightGear. 65
Figura 54. Piper J3 Cub. 102
Figura 55. Página principal de FG 104
Figura 56. Elección del S.O. 105
Figura 57. Icono de acceso directo a FlightGear. 105
Figura 58. Elección del avión. 106
Figura 59. Pantalla de opciones. 107
Figura 60. Descargando Wireshark. 108
Figura 61. Opciones de captura de Wireshark. 109
Figura 62. Descarga de Eclipse 110

Capítulo 1. Introducción

1.1. Motivación

Más que una motivación, han sido varias las motivaciones encontradas para que finalmente me decidiera por la elección de controlar el simulador de vuelo FlightGear mediante una aplicación basada en lenguaje de programación Java, a continuación las explico una por una.

Uno de los motivos por el cual he elegido realizar este proyecto ha sido principalmente en que éste parecía ser divertido y entretenido, y de hecho, así ha sido. Encontré varios proyectos ofertados pero finalmente me quedé con la última propuesta, la de controlar remotamente un simulador de vuelo. Me pareció que era algo diferente, sin muchas fórmulas matemáticas, ni estadísticas, sin teoría de teletráfico y colas de espera y sin mucho o nada a tener que ver con tratamiento o teoría sobre señales. Además, este proyecto está basado en el lenguaje de programación de Java, que probablemente haya sido una de mis asignaturas favoritas impartidas en mi carrera, por tanto, un plus añadido, y otra motivación.

Otra motivación muy importante para mí fue que uno de mis directores del proyecto iba a ser Diego Alonso Cáceres, al que ya conocía anteriormente, debido a que fue uno de mis profesores de prácticas de la asignatura de Fundamentos De Programación, y me pareció muy claro en sus explicaciones y muy accesible a la hora de ayudarnos a resolver los problemas que nos iban surgiendo en dicha asignatura.

No voy a decir que antes de elegir este proyecto me encantaran los aviones y que ésta fue la causa de haberlo elegido, la verdad es que no tenía ni idea sobre el mundo de la aviación y el hecho de adentrarse y explorar algo desconocido significa un reto, por aprender cosas nuevas y por tanto otra motivación.

En realidad, no puedo comparar directamente éste proyecto con otros, ya que es el único que he realizado, pero creo que pocos o ninguno hubiesen sido más entretenidos y llevaderos. Las únicas comparaciones que puedo hacer son con los comentarios sobre proyectos de final de carrera de otros compañeros que ellos mismos me han hecho, todos me han dicho que parecía ser un trabajo bastante difícil pero también bastante mas divertido que sus propios proyectos, yo también lo creo.

Todo lo dicho anteriormente me motivaría aún mas para trabajar en el proyecto final de carrera, ayudándome a dedicarle el suficiente tiempo y que las horas empleadas en él no se me hicieran muy pesadas o aburridas.

Ahora que he terminado este trabajo y que he adquirido unos pocos conocimientos sobre aviones, puedo decir que me ha gustado bastante, mucho más de lo que yo me esperaba y que no creo que lo abandone del todo. Seguiré aprendiendo a pilotar con FlightGear diferentes aviones e incluso cuando tenga medios, mejoraré el software de mi ordenador y añadiré algún accesorio para divertirme más y para que la experiencia de pilotar un avión mediante un simulador sea más realista.

1.2. Objetivos

El objetivo de este PFC es diseñar un sistema básico de control para el simulador de vuelo de libre distribución FlightGear. Este sistema deberá ser capaz de efectuar automáticamente las maniobras básicas de vuelo, como son vuelo recto y nivelado, ascenso y descenso a una determinada altitud y giro hasta un determinado rumbo, todo ello programado en el lenguaje de programación Java. El sistema tendrá además una interfaz gráfica en la que se mostrará la evolución en el tiempo de diversos parámetros del vuelo, como altitud, velocidad lineal, posición de los mandos del avión, etc.

1. Instalación y puesta en marcha del simulador FlightGear y del entorno de programación Eclipse para Java.
2. Estudio de los protocolos de comunicación con el simulador y elección del más adecuado.
3. Programación del protocolo de comunicación y verificación del correcto funcionamiento del mismo.
4. Programación y ajuste del controlador de un eje del avión.
5. Programación y ajuste de los controladores de todas las variables del avión.
6. Desarrollo de una herramienta gráfica extensible.

1.3. Estructura de la memoria

La presente memoria se divide en los siguientes capítulos:

Capítulo 1: Introducción.

Capítulo dedicado a ofrecer una visión general del proyecto.

Capítulo 2: FlightGear, maniobras fundamentales en aviación y Teoría de Control.

Este capítulo aborda los tres temas fundamentales de este proyecto: el simulador de vuelo FlightGear, las maniobras que realizamos en el simulador como son ascensos, descensos, giros y vuelo recto y nivelado; y Teoría de Control, que es la herramienta que nos permite hacer estas maniobras de forma automática.

Capítulo 3: Descripción de la solución.

Explica paso a paso como se ha realizado el proyecto, desde la elección del protocolo de comunicación hasta la interfaz gráfica, incluyendo los problemas encontrados en cada caso.

Capítulo 4: Conclusiones y experiencia.

Valoración del trabajo realizado, posibilidades de continuación, aprendizaje y posibles aplicaciones futuras.

1.4. Otras iniciativas similares con FlightGear

Destacar dos proyectos basados en UAS (Unmanned Aerial System), es decir, aviones no tripulados, que usan FlightGear. En uno se usa FlightGear como sustituto del avión real, como es el caso de la UPC (Universidad Politécnica de Cataluña) y en el otro se usa para desarrollar el modelo dinámico de vuelo, es decir, para simular el comportamiento que tendrá el avión real, como es el caso de la UPM (Universidad Politécnica de Madrid).

La Universidad Politécnica de Cataluña ha desarrollado un "Sistema PARROT de posicionamiento audible por radio orientado a territorio", en el cual el grupo de investigación ICARUS intenta crear un conjunto de sistemas necesarios para equipar a un avión no tripulado (simulado con FlighGear) para que éste sea capaz de sobrevolar una zona con el objetivo de detectar incendios.

El objetivo de este proyecto es poder equipar al UAS con un mecanismo de radiocomunicación, más exactamente implementar una aplicación capaz de emitir por radio la posición del avión, con el objeto de evitar posibles accidentes con aviones en cobertura, ya que la señal de radio podrá ser recibida por una emisora en concreto por cualquier receptor. Otro objetivo es crear un mecanismo capaz de trazar una ruta de manera automática para el UAS. De esta manera, se puede controlar automáticamente la dirección y rumbo del UAS sin necesidad de tocar los mandos manuales del simulador. Así cuando el avión real está volando, la única opción es modificar el trayecto de manera remota a través de un ordenador sin necesidad de utilizar un mando a distancia.

Dicha aplicación trabaja dentro del perímetro de Cataluña, y para ello ubica al avión dentro de la zona gracias a un simulador de vuelo que simulará un vuelo real. Una vez se conoce la posición emitimos por radio unos mensajes indicando el rumbo y origen del avión. El resultado completo pues, consiste en introducir un plan de vuelo al simulador e implementar un sistema de posicionamiento por radio, para indicar las intenciones del avión hacia otros aviones o estaciones base, siguiendo el vocablo propio de la aeronavegación, con el objetivo de mantener localizado al UAS y poder trazar un perímetro de acción en caso de localizar fuego, y a su vez poder evitar accidentes aéreos dentro de un perímetro de vuelo concreto.

En la figura 1 podemos ver el esquema de la comunicación entre el avión y la recepción de sus coordenadas.

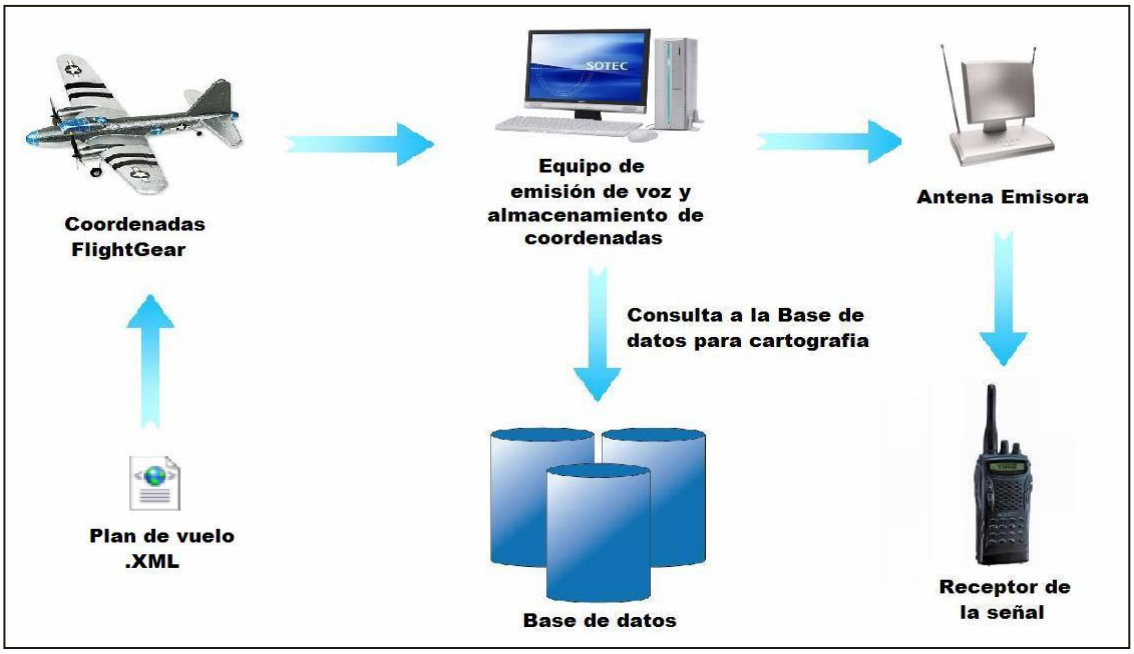


Figura 1. Esquema del proyecto de la UPC.

En resumen, FlightGear interviene de forma que simula el UAS, el simulador recibe el plan de vuelo (altitudes, latitudes y velocidades) en formato .XML mediante socket, así una vez recibidos los parámetros, el avión (FlightGear) realizará la ruta de forma automática. Para la monitorización del avión, el simulador envía su posición actual al equipo de almacenamiento de coordenadas en todo momento, también mediante socket.

El otro proyecto se denomina SISCANT (SIStema de Control Adaptativo para Aviones No Tripulados), ver [1] y se trata de un sistema avanzado de control de vuelo fly-by-wire desarrollado en España. En la figura 2 los responsables del proyecto.

Figura 2. Álvaro López (TCPSI) y Patricio Gómez (GISD-UPM).

El Grupo de Investigación en Sistemas Dinámicos de la Universidad Politécnica de Madrid está participando en el desarrollo del proyecto SISCANT (SIStema de Control Adaptativo para Aviones No Tripulados), centrándose en el diseño de las leyes de control y de los estimadores de estado para la determinación de posiciones, velocidades y actitud.

El proyecto, que está liderado por la empresa española TCP Sistemas e Ingeniería, cuenta con la participación, además del Grupo de Sistemas Dinámicos de la UPM, del Instituto de Investigación Tecnológica de la Universidad Pontificia de Comillas y de la empresa UAV Navigation.

SISCANT aborda el desarrollo completo y la certificación de un sistema de control de vuelo (Flight Control System, FCS) avanzado que permita el control de un avión no tripulado (Unmanned Aerial Vehicle, UAV) adaptándose automáticamente a fallos catastróficos que afecten a las actuaciones de la aeronave, por daños en la estructura, fallos de motor, actuadores o sensores, así como a condiciones ambientales adversas, decidiendo de forma autónoma proseguir la misión o abortar, devolviendo la aeronave a su base, de forma segura. En la figura 3 se muestran los componentes y su relación.

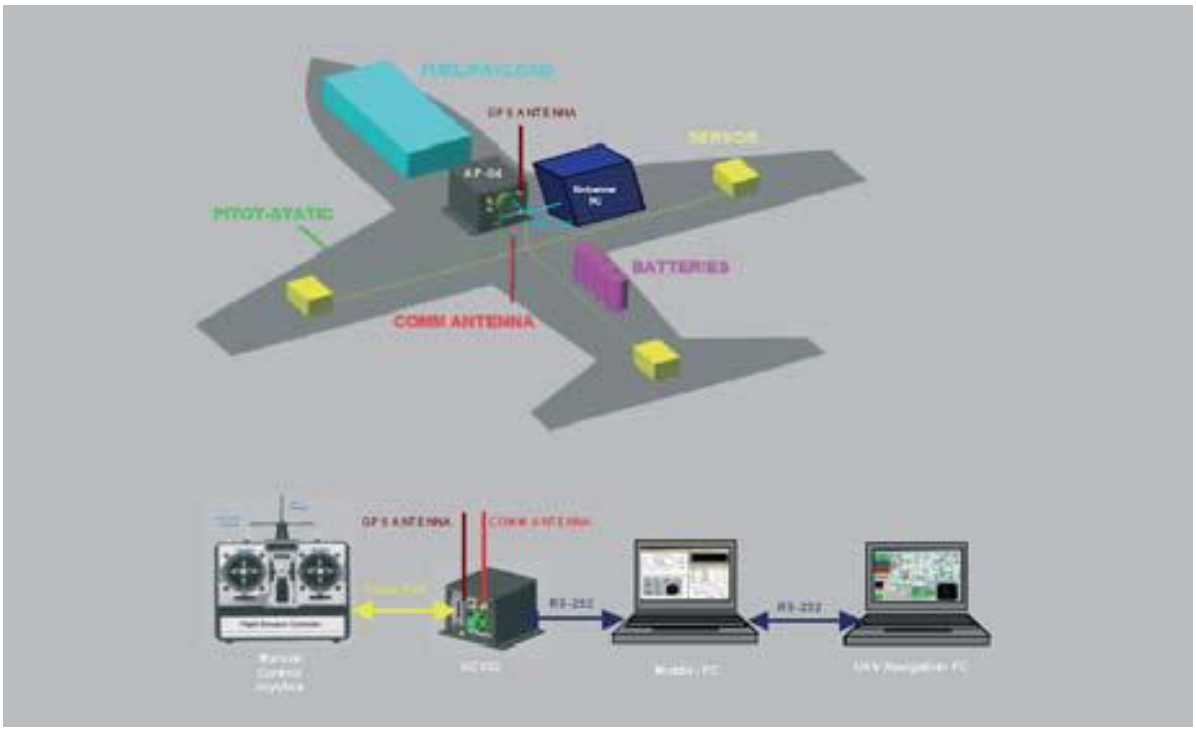


Figura 3. Arquitectura del Sistema SISCANT.

El modelo del avión se ha volado en el simulador de vuelo Flight Gear. En primer lugar se realizaron vuelos en lazo abierto y, posteriormente, con los distintos controladores. Esto permitió someter a distintas maniobras típicas de pruebas en vuelo para excitar los modos propios (corto período, fugoide, balanceo del holandés, convergencia de balance y espiral), comprobando la respuesta del sistema desde el punto de vista de un hipotético piloto.

Capítulo 2. FlightGear, maniobras básicas de aviación y teoría de control

2.1. FlightGear

Figura 4. Imagen del simulador de vuelo FlightGear.

El simulador de vuelo elegido para la realización de este proyecto es FlightGear Flight Simulator (FGFS), un simulador de código abierto, libre, que además soporta varias plataformas como Windows (32 bits), Linux, Mac OS-X, FreeBSD, Solaris e IRIX.

FlightGear no es sólo un simulador de vuelo, es también un proyecto para el desarrollo del mismo de forma cooperativa, ya que cualquiera que tenga suficientes conocimientos e ideas para participar en el desarrollo de FGFS puede hacerlo debido a que el código fuente de todo el proyecto está enteramente disponible bajo la licencia de GNU General Public License. La figura 5 muestra el logotipo oficial de FlightGear.

Figura 5. Logo de FlightGear.

La idea de FlightGear surgió fruto del descontento que generaban los simuladores de vuelo comerciales de aquella época (el proyecto FGFS comenzó en 1.996). El problema

principal era que al ser de propietarios no podían ser modificados al gusto del usuario, careciendo así de extensibilidad y flexibilidad.

En la actualidad FGFS es una alternativa importante frente a los simuladores de vuelo comerciales. Es probablemente el único programa de este tipo cuyo código es libre y sin intención de esconder cómo funciona internamente, lo que lo hace muy extensible. Hay usuarios que consideran que no consigue superar el nivel gráfico de los mejores productos comerciales, pero el modelo físico del vuelo y el realismo de los controles están al mismo o mayor nivel que los mejores simuladores. Esto se debe a que FlightGear fue desarrollado desde un comienzo con un alto perfil técnico y científico. Se apoya en OpenGL[1] y requiere hardware de aceleración 3D. FlightGear está principalmente escrito en C++. Además ofrece un tutorial extenso, ver [2].

Los objetivos que persigue el proyecto son:

- Crear un sofisticado marco de trabajo de simulador de vuelo para su uso en investigación o ambientes académicos, de esta manera se favorece el desarrollo y la búsqueda de interesantes ideas para el vuelo por simulación.

- Ser una aplicación de usuario final.

2.1.1 Modelos Dinámicos de Vuelo

La razón principal de por qué los usuarios de FlightGear consideran que está a la altura de los mejores simuladores de vuelo comerciales es por el realismo del comportamiento del vuelo de los muchos aviones que podemos pilotar en éste simulador. Y dicho realismo es gracias a los Modelos Dinámicos de Vuelo o FDM que simulan el vuelo del aparato.

El FDM es el cómo se comporta el avión, el código que simula las características de vuelo o comportamiento de un aeronave, mediante funciones formadas por fórmulas y cálculos matemáticos que simulan por ejemplo las fuerzas de aceleración del avión, la gravedad, la aerodinámica o también la presión de la atmósfera, la altitud, latitud, etc… FlightGear usa una variedad de proyectos de FDM internamente escritos. Toda aeronave ha de ser programada para usar alguno de estos modelos. En la actualidad FlightGear es el único simulador de vuelo gráfico que usa todos los Modelos Dinámicos de Vuelo. FlightGear también puede ser configurado para usar FDM externos, como por ejemplo desde Matlab. También han sido escritos otros FDM personalizados para aeronaves específicas, como las aeronaves que son más ligeras que el aire.

La versión temprana usaba un FDM de la NASA basado en LaRCsim, el cual fue reemplazado por un FDM más flexible.

[1] Open Graphics **L**ibrary. Desarrollada por Silicon Graphics Inc. en 1.992, es una librería estándar que define una API multilenguaje y multiplataforma de más de 250 funciones para crear aplicaciones gráficas en 2D y 3D.

Lejos de empezar enteramente de la nada. Los desarrolladores de FlightGear hicieron uso del modelo de vuelo LaRCsim de la NASA, con OpenGL para el código de gráficos 3D, y datos de elevación libremente disponibles. Los primeros trabajos binarios salieron en 1997, progresivamente más estable y con programas más avanzados, resultado de una intensa actualización de nuevas versiones durante años.

A continuación pasamos a describir brevemente cada FDM:

LaRCsim: Desarrollado en La Nasa y utilizado en una amplia gama de proyectos, fue el FDM por defecto de FlightGear hasta el año 2.000. Sus principales funciones son:

- *ls_accel()* que suma las fuerzas y momentos de la aerodinámica, motor, engranaje y las transfiere al centro de gravedad para calcular las aceleraciones resultantes.
- *ls_step()* Integra aceleraciones para obtener las velocidades y éstas para obtener la posición de la aeronave.
- *ls_aux()* a partir de las velocidades y las posiciones calcula otros parámetros como ángulos de ataque (alpha) y deslizamiento lateral (beta), las presiones y temperaturas, etc…
- *atmos_62()* búsquedas de tabla de atmósfera estándar de 1.962.
- *ls_geoc_to_geod(lat_geoc, radius, lat_geod, alt, sea_level_r)* y *ls_geod_to_geoc(lat_geod, alt, sl_radius, lat_geoc)* para calcular posiciones geocéntricas y geodésicas.
- *ls_gravity(SCALAR radius, SCALAR lat, SCALAR *gravity)* modela la gravedad local de LaRCsim basándose en la altitud y latitud.

También cuenta con rutinas específicas para Navión, un avión monomotor y cuatro plazas norteamericano construido en 1.940.

JSBSim: Escrito en C++, usa ficheros de configuración XML, desarrollado en 1996 y también usado por OpenEaagles[2]. Este modelo de vuelo es totalmente configurable desde ficheros XML: aerodinámica, propulsión, hasta incluso el arreglo del tren de aterrizaje, pasando por la simulación de los efectos rotacionales de La Tierra.

[2] OpenEaagles: es un framework de simulación, multiplataforma, dirigido a ingenieros y desarrolladores de software, para construir aplicaciones de simulación.

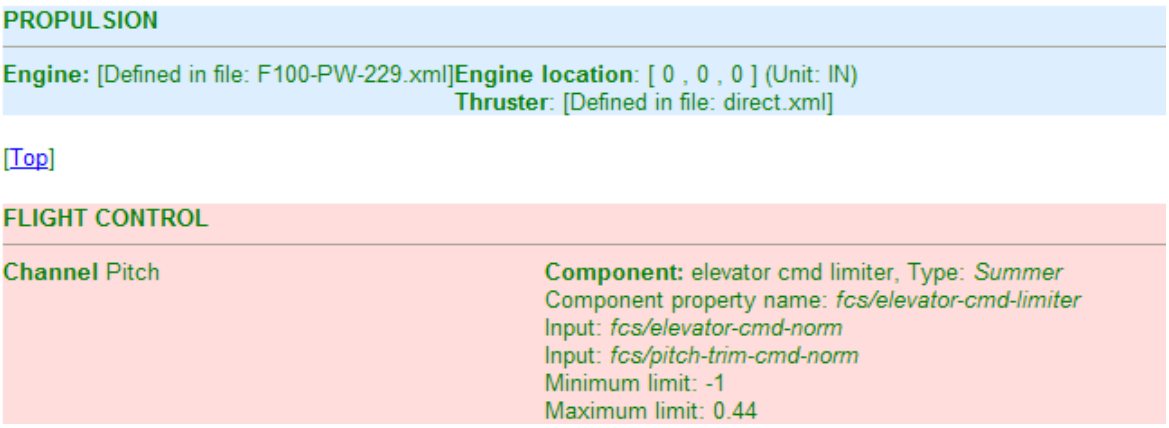


Figura 6. Parte de un archivo XML para la configuración JSBSim de un F16.

En la figura 6 podemos ver parte del archivo XML anteriormente citado. JSBSim es el modelo de vuelo por defecto en FlightGear desde el año 2.000.

Existe una herramienta llamada Aeromatic, un cuestionario online como el de la figura 7, que genera archivos basados en JSBSim, aunque es muy fácil generar un modelo de vuelo no fiable ya que, dependiendo de la simetría y del número de motores del avión habría que realizar algunos cambios manualmente. No es una herramienta independiente, necesita dos archivos, uno que contenga información sobre las características del avión (peso, aerodinámica…) y el otro sobre el motor o motores del mismo.

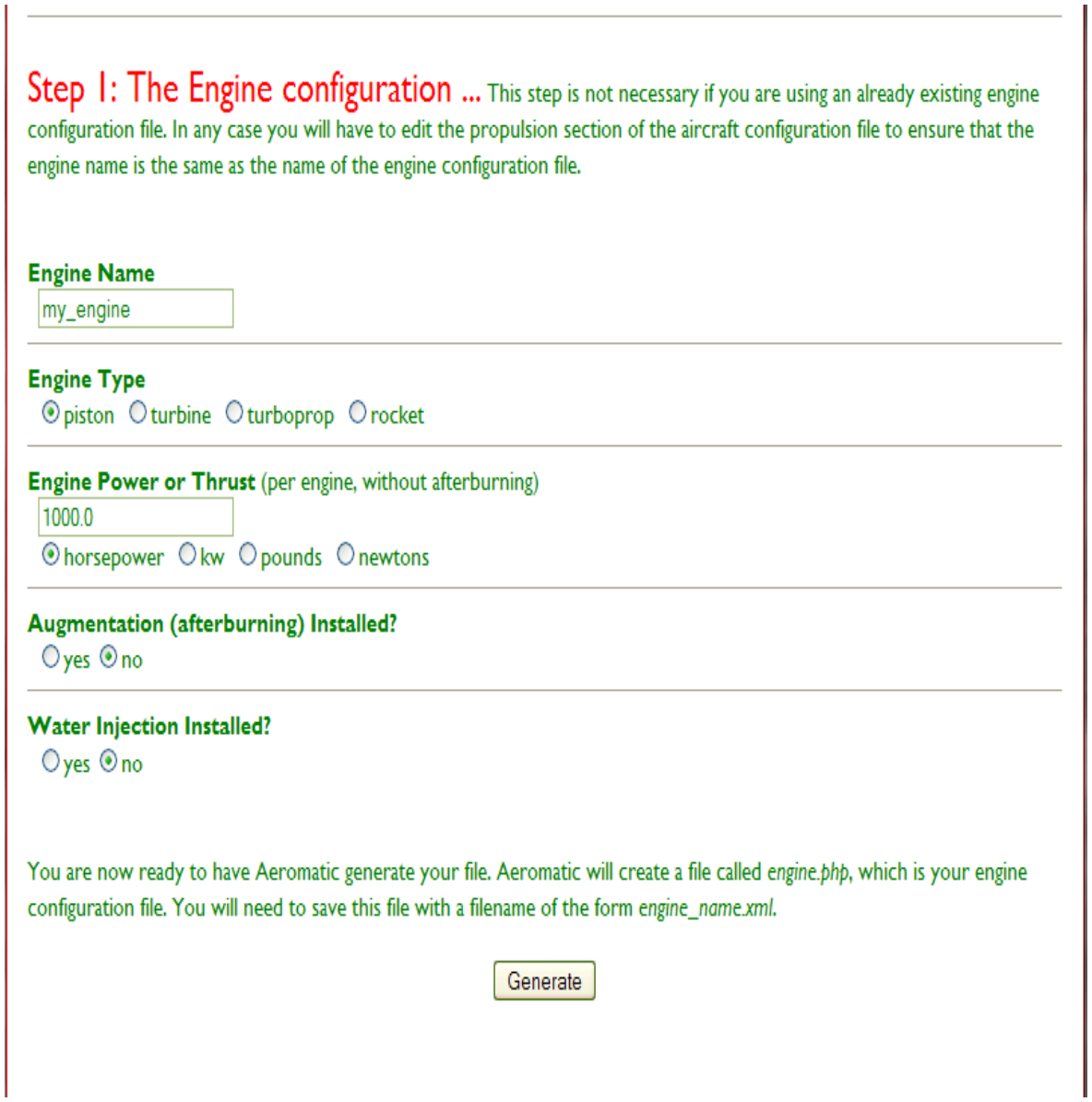


Figura 7. Parte del cuestionario Aeromatic.

También existe un editor en modo gráfico llamado JSBSim Commander, figura 8, el cual crea y edita FDM basados en JSBSim.

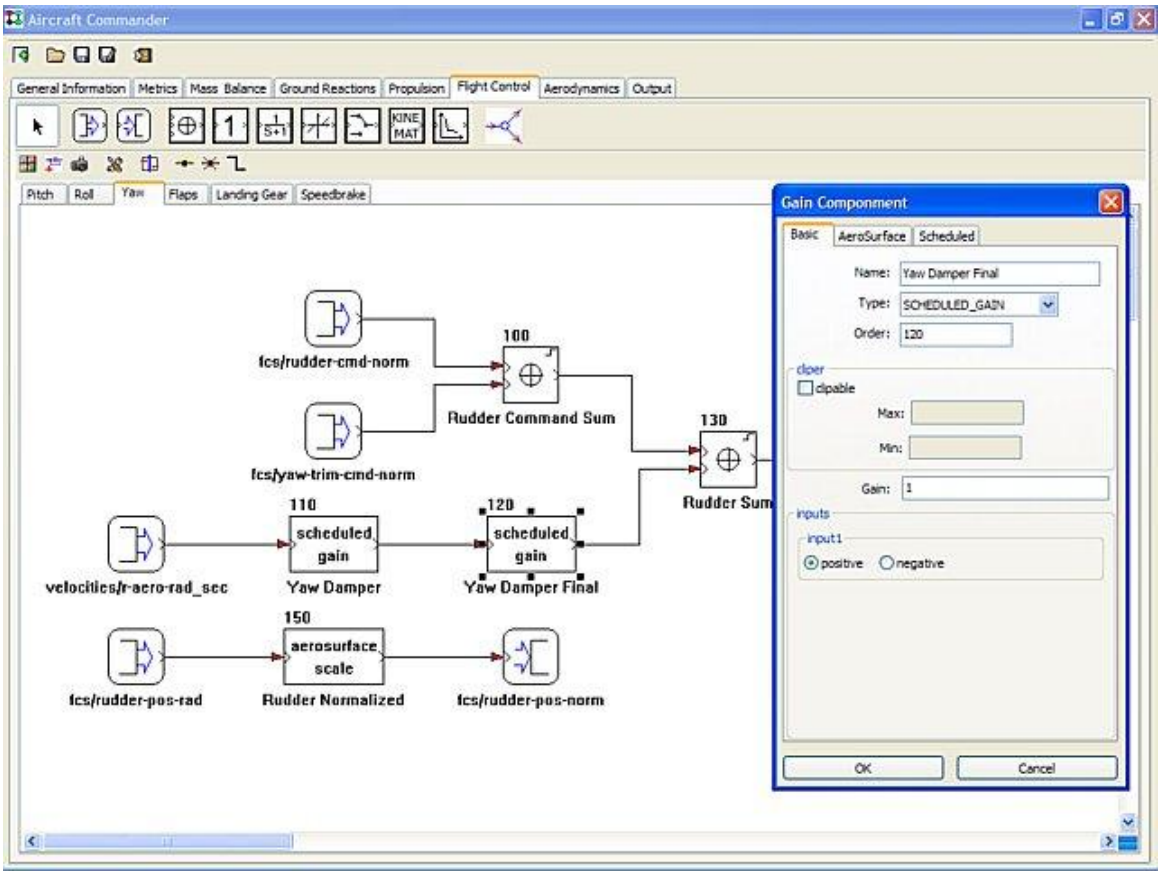


Figura 8. Editor JSBSim Commander.

YASim: Escrito en 2.002, este FDM simula la corriente del aire en las distintas partes del avión. Está basado en las notas del sistema de coordenadas, esto es, poniendo la mano derecha como si fuera una pistola, pero con el dedo corazón apuntando hacia la izquierda, el eje X (dedo índice) sería el morro del avión, el eje Y (dedo corazón) serían las alas y el eje Z (dedo pulgar) sería la cola del aparato. Si giramos cada uno de los ejes sobre sí mismos, estaríamos simulando los movimientos de alabeo, cabeceo y guiñada respectivamente.

La ventaja de este enfoque es que es posible realizar la simulación con la combinación de las informaciones de geometría y masa del avión, esto permite implementar más rápidamente un comportamiento muy real de un avión sin necesidad de tener los datos de la tradicional prueba aerodinámica.

YASim también se configura mediante un archivo XML como muestra la figura 9.

Existe un programa llamado Blender[3], figura 10, con el que se puede visualizar en 3D la geometría del avión configurada con YASim e incluso compararla con su modelo de avión original.

[3] Blender es una suite de creación de contenido 3D publicada bajo la licencia GNU General Public License, usada por muchos usuarios de FlightGear.

```
CODE: SELECT ALL

<airplane mass="1620">

  <!-- Approach configuration -->
  <approach speed="45" aoa="14">
    <control-setting axis="/controls/engines/engine[0]/throttle" value="0.3"/>
    <control-setting axis="/controls/engines/engine[0]/mixture" value="1"/>
    <control-setting axis="/controls/flight/flaps" value="1"/>
  </approach>

  <!-- Cruise configuration -->
  <cruise speed="122" alt="6000">
    <control-setting axis="/controls/engines/engine[0]/throttle" value="1"/>
    <control-setting axis="/controls/engines/engine[0]/mixture" value="1"/>
```

Figura 9. Parte del archivo XML de configuración de YASim.

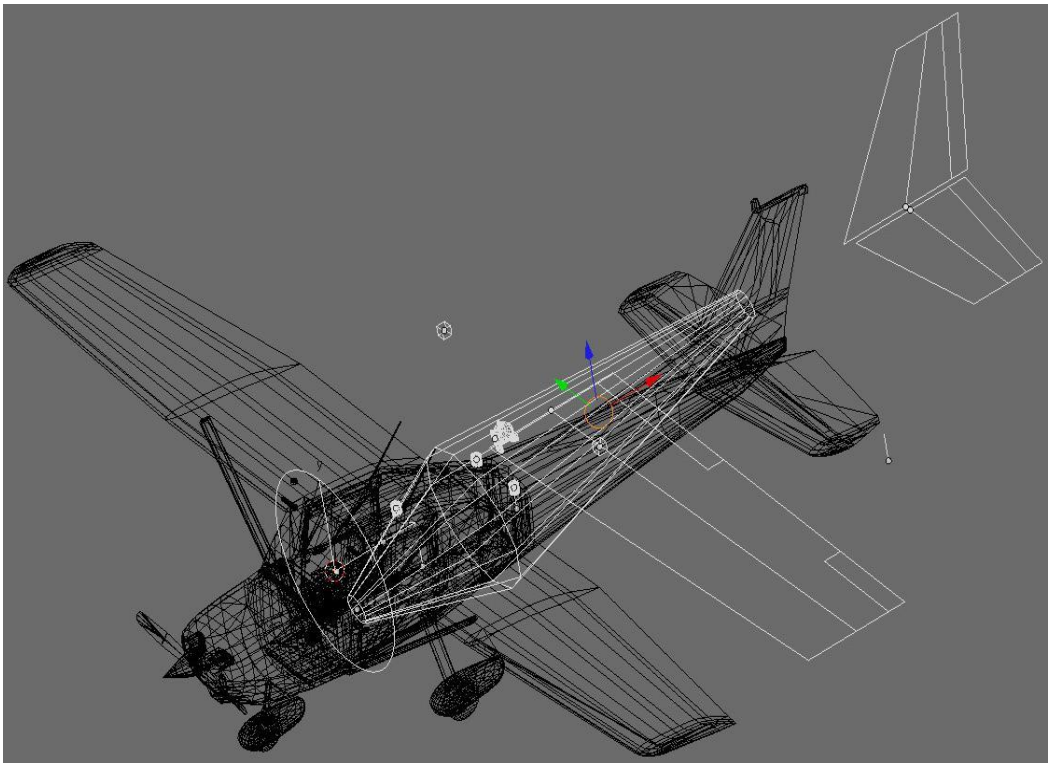


Figura 10. En blanco los cambios realizados en el fichero de configuración XML.

UIUC: Este FDM está basado en LaRCsim y fue desarrollado a principios del año 2.000 por UIUC Applied Aerodynamics Group de la Universidad de Ilinois, Chicago. UIUC extiende su código por permitir archivos de configuración de avión y por la adición de código para la simulación de aviones bajo condiciones de hielo[4]. Actualmente es usado por muchos modelos de avión en FlightGear. UIUC, al igual que JSBSim, utiliza tablas de

[4] Icing conditions: en aviación, son las condiciones atmosféricas bajo las cuales podría aparecer hielo en la superficie del avión o incluso en el motor. No todos los aviones pueden volar bajo condiciones de hielo.

búsqueda para recuperar los componentes de fuerza aerodinámicos y los coeficientes del momento (cabeceo, alabeo y guiñada) del avión, y después, utiliza estos coeficientes para calcular la suma de las fuerzas y momentos que actúan en el avión.

2.1.2 Otras características

Otras características que convierten a FlightGear en uno de los simuladores más realistas del mundo son:

- Terreno preciso de todo el mundo, basado en la publicación más reciente de los datos de terreno SRTM[5]. El escenario incluye todos los lagos, ríos, carreteras, ferrocarriles, ciudades, pueblos, terrenos, etc.
- Base de datos del escenario mundial precisa y extensa.
- Alrededor de 20.000 aeropuertos reales.
- Modelo del cielo detallado y preciso, con ubicaciones correctas del sol, la luna, las estrellas y los planetas para la fecha y hora especificadas.
- Sistema de modelado de aviones abierto y flexible, amplia variedad de naves.
- Animación instrumental extremadamente fluida y suave. Modela de una forma realista el comportamiento de los instrumentos del mundo real. Incluso reproduce de forma precisa los fallos de muchos sistemas e instrumentos.
- Modo multijugador.
- Simulación de tráfico real mediante Inteligencia Artificial.
- Opción de tiempo real que incluye tanto la iluminación del sol, el viento, la lluvia, niebla, humo, etc.

2.2. Maniobras fundamentales en aviación

En este apartado sólo nos centraremos en las maniobras básicas que se realizan con un avión, en lugar de hablar de la aviación en general, ya que son las cuatro maniobras que se realizan en este proyecto. Ver [3].

Desde que el avión despega hasta que aterriza, es decir durante el vuelo, seguramente será necesario realizar alguna de estas cosas: respecto a la altura, ascender, descender, o mantener una altitud constante; en cuanto a dirección, girar a la derecha, a la izquierda, o mantener la dirección de vuelo, y en cuanto a velocidad, acelerar, decelerar o mantener una velocidad constante. Estas tareas reciben el nombre de maniobras fundamentales, pues cualquier maniobra que se realice en vuelo requerirá el empleo de alguna de ellas, individualmente o combinadas entre sí: ascensos, descensos, giros, y vuelo recto y nivelado.

Pero antes de pasar a explicar cada una de éstas maniobras, primero es conveniente hablar de los tres ejes que gobiernan cualquier movimiento de un avión y posteriormente de la potencia y la actitud.

[5] Misión Topográfica Radar Shuttle (SRTM) es un sistema de radar modificado para obtener un modelo digital de elevación del globo terráqueo entre 56 ˚S a 60 ˚N que genera una base de cartas topográfica completa de La Tierra.

Cualquier avión es capaz de realizar 3 giros posibles alrededor de 3 ejes perpendiculares entre sí cuyo punto de intersección está situado sobre el centro de gravedad del avión. Estos 3 ejes, como muestra la figura 11 son: eje transversal (o lateral), longitudinal y vertical.

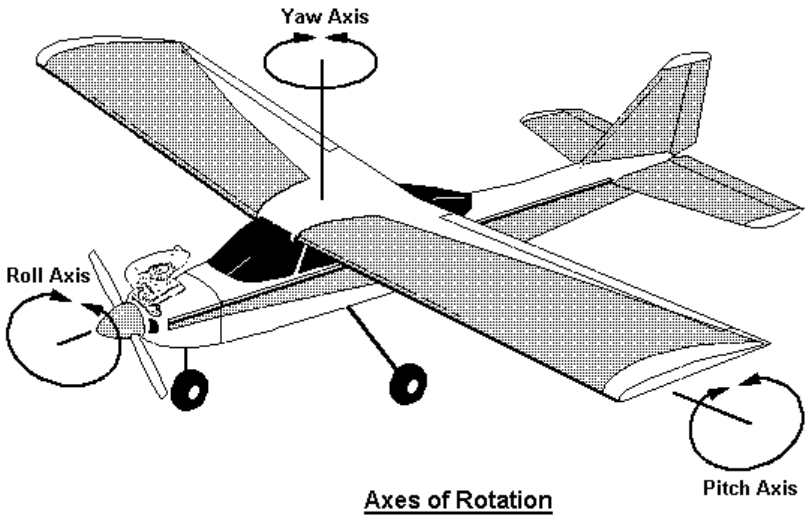


Figura 11. Los tres ejes de un avión.

El eje **lateral** o **transversal** (pitch axis) es un eje imaginario que se extiende de punta a punta de las alas del avión. El movimiento que realiza el avión alrededor de este eje se denomina **cabeceo**.

Figura 12. Movimiento de cabeceo.

El piloto, es capaz de modificar la orientación respecto a este eje a través del timón de profundidad. Al tirar hacia atrás (hacia el piloto) se produce una elevación del morro del avión, y al empujarlo adelante se produce una bajada del morro del avión.

El **eje longitudinal** (roll axis) es un eje imaginario que se extiende desde el morro a la cola del avión. El movimiento que realiza el avión alrededor de este eje se denomina **alabeo**.

Figura 13. Movimiento de alabeo.

Las superficies de mando del alabeo son los alerones. Al girar el bastón de mando se produce la deflexión diferencial de los alerones: al tiempo que el alerón de una de las alas sube, el alerón de la otra ala baja, siendo el ángulo de deflexión proporcional al grado de giro de los cuernos de mando. El alerón que se ha flexionado hacia abajo, produce un aumento de sustentación en su ala correspondiente, provocando el ascenso de la misma, mientras que el alerón que es flexionado hacia arriba, produce en su ala una disminución de sustentación, motivando el descenso de la misma. El piloto, en caso de querer inclinarse hacia la izquierda, giraría el cuerno de mando hacia la izquierda, haciendo que el alerón derecho descendiera elevando así el ala derecha, y simultáneamente, el alerón izquierdo se flexionaría hacia arriba produciendo una pérdida de la sustentación en el ala izquierda y por tanto su descenso.

El **eje vertical** (yaw axis) es un eje imaginario que, pasando por el centro de gravedad del avión, es perpendicular a los ejes transversal y longitudinal. Este eje es perpendicular al eje de cabeceo y al de balanceo, está contenido en un plano que pasa por el morro y la cola del aparato y que normalmente divide a este en dos partes simétricas). El movimiento que realiza el avión alrededor de este eje se denomina *guiñada*.

Figura 14. Movimiento de guiñada.

La superficie de mando de la guiñada es el timón de cola o timón de dirección. El control sobre el timón de dirección se realiza mediante los pedales. Para conseguir un movimiento de guiñada hacia la derecha, el piloto presiona el pedal derecho, generando así un giro de la superficie del timón de dirección hacia la derecha. Al ofrecer más resistencia al avance por este lado, el aparato tiende a retrasar su parte derecha y avanzar la izquierda y por tratarse de una estructura rígida el resultado es un giro a la derecha sobre el eje vertical mencionado. La guiñada puede ocurrir de forma involuntaria en vuelo o en tierra. En vuelo puede ser causada por una ráfaga de viento lateral o por irregularidades

aerodinámicas debidas al pilotaje. En casos extremos se puede llegar a la autorrotación, que origina la barrena. La guiñada en tierra puede ser provocada, además de las causas citadas, por diferente resistencia al avance entre una y otra rueda debida a la superficie del terreno o a una frenada irregular que puede provocar un "caballito", incidente en el que el aparato sufre una guiñada rápida de 90º o más, con peligro de rotura de un ala.

2.2.1. Potencia y actitud

La potencia y la actitud no son maniobras fundamentales, pero si que son aspectos importantísimos para entender el comportamiento de un avión, además todas las maniobras que se realizan en un avión están basadas en estos dos aspectos.

Se puede ascender o descender con distintas tasas de ascenso o descenso, girar con mayor o menor tasa de giro y con diferentes grados de alabeo, todo ello con diferentes velocidades, igual que se puede volar recto y nivelado a distintas velocidades. Pero hay un hecho característico: el control del aeroplano depende de la potencia aplicada y de la actitud[6] de morro y alabeo del avión. La velocidad vertical (tasa de ascenso/descenso) y horizontal en un ascenso o descenso dependerá de la potencia puesta y de la actitud del avión; en vuelo recto y nivelado la velocidad desarrollada dependerá igualmente de la potencia y la actitud; en giros lo mismo, en cualquier maniobra combinada también. Por tanto, al aplicar una potencia específica y poner al avión con una actitud determinada el piloto está "seleccionando" los parámetros de vuelo.

El comportamiento a grandes rasgos de los aviones es el siguiente:

- Diferentes combinaciones de potencia y actitud producen distintas velocidades y tasas (de ascenso, descenso o giro) por lo que una misma maniobra puede efectuarse bajo distintos parámetros de velocidad y tasas dependiendo de la potencia y actitud adoptadas.
- Con la misma potencia aplicada, una actitud de más morro arriba supone menor velocidad del avión y más morro abajo mayor velocidad, es decir, el control primario de la velocidad se ejerce con la actitud (volante de control).
- Manteniendo la velocidad, si se reduce potencia el avión descenderá, si se aumenta ascenderá, es decir, el control primario de la altura reside en la potencia (palanca de gases).
- Si se cambia de potencia sin cambiar de actitud, o de actitud sin cambiar de potencia, los parámetros de vuelo (velocidad, altura, tasas) cambiarán.
- Si se quieren mantener dichos parámetros, un cambio de potencia implica también un cambio de actitud y viceversa.

La actitud básica de un avión es la actitud de crucero, esto es, un vuelo nivelado, con una velocidad y altitud constante, con una potencia adecuada y las alas paralelas al horizonte.

[6] Denominación que recibe la posición del avión respecto al horizonte.

Figura 15. Actitudes de morro.

Figura 16. Actitudes de alabeo.

La potencia se puede utilizar para superar la resistencia, para acelerar, o para ascender.

Con el avión compensado en vuelo recto y nivelado y una potencia moderada, si aumentamos la potencia sucede una cosa muy curiosa: el avión no acelera (en muchos aviones incluso decelera ligeramente) sino que levanta ligeramente el morro y comienza a ascender, esto es debido a que un avión puede transformar este aumento de energía en sentido horizontal y vertical, y debido a la aerodinámica, la transforma en sentido vertical (ascendiendo). Obviamente, disminuir la potencia, no hace que el avión decelere (la velocidad incluso aumenta ligeramente) sino que baje algo el morro y descienda.

2.2.2. Ascensos

El ascenso es una maniobra básica durante la cual una combinación adecuada de potencia y actitud hace ganar altura al avión. Aunque también podríamos hablar de potencia y velocidad, ya que la actitud es un controlador del ángulo de ataque, y la mejor información sobre éste nos la indica la velocidad (ya que no hay ningún indicador del ángulo de ataque), pero éste sería el punto de vista del piloto.

La potencia es necesaria en primer lugar para vencer la resistencia al avance del avión. La cantidad de resistencia a vencer depende de la velocidad de una forma cuya expresión gráfica se muestra en la figura 17.

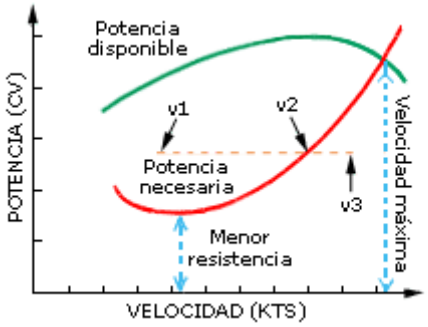


Figura 17. Potencia y velocidad en el ascenso.

Para una misma potencia aplicada, representada por la línea discontinua de la figura, con velocidad *v1* el avión está en ascenso; con *v2* vuela nivelado y con *v3* está en descenso. Podemos decir que el ascenso se produce cuando aplicamos mas potencia de la necesaria para vencer la resistencia a la que es sometido el avión a cierta velocidad. Por ejemplo si volamos al 70% de potencia con una velocidad de 90 nudos en vuelo recto y nivelado, con la misma potencia pero a 70 nudos el avión ascenderá, ya que la resistencia es menor. Menor velocidad no implica menor resistencia, a velocidades muy bajas también necesitamos una potencia muy alta para vencer la resistencia, también podemos conseguir la misma tasa de ascenso con diferentes velocidades, una muy alta y otra muy baja, esto sería por ejemplo el caso de adoptar un ángulo de ataque muy elevado, por tanto necesitaríamos mucha potencia solo para vencer la resistencia que implica ese ángulo, resultando una velocidad muy baja. Resumiendo, las claves del ascenso son:

- Ascender requiere más potencia que el vuelo nivelado.
- Aumentar la potencia manteniendo el ángulo de ataque (la velocidad) hace que el avión ascienda.
- Con una misma potencia, de todas las velocidades posibles la mejor tasa de ascenso se obtiene con una específica. Esta se corresponde con un ángulo de ataque determinado.
- La mejor tasa de ascenso no se obtiene con un mayor ángulo de ataque (actitud de ascenso muy pronunciada) sino con una combinación adecuada de potencia y velocidad.

2.2.3. Descensos

El descenso es la pérdida de altura del avión, de una forma controlada, mediante una combinación de actitud de morro (véase figura 15) y potencia. Hay dos tipos de descensos: asistidos por el motor o sin motor, a éstos últimos también se les puede denominar planeo.

Dependiendo de la tasa de descenso y la distancia se puede utilizar una u otra maniobra, por ejemplo, los descensos asistidos por el motor se realizan cuando es necesario un control preciso de la tasa de descenso y la distancia recorrida durante el mismo. La mayoría de los aviones comerciales realizan este tipo de descenso habitualmente, tanto en crucero como en aproximación para aterrizar, procurando así un mejor confort al pasaje y para cumplir con los requerimientos de velocidad y espacio entre aeronaves demandados por el control de tráfico. El descenso en planeo requiere un mayor control de la trayectoria

de vuelo (senda de planeo), pues al no aportar potencia el motor, solo se cuenta con la actitud para controlar el aeroplano y ello no proporciona muchas variantes sobre la tasa de descenso, la velocidad o la distancia recorrida. Salvo descensos para aterrizajes, o casos muy especiales, lo habitual es descender con motor pues ello proporciona mayores posibilidades de control del avión y su trayectoria.

Como se ha mencionado antes, las claves del descenso son potencia y actitud o potencia y velocidad (desde el punto de vista del piloto). Si para pasar de vuelo nivelado a ascenso es necesario un excedente de potencia (abrir gases) para la maniobra inversa, descenso, será preciso un déficit de potencia (cortar gases). Sabemos que abrir gases (manteniendo la velocidad) produce que el avión ascienda, así que el efecto contrario, cortar gases, no resulta novedoso: el aeroplano mantendrá la velocidad para la cual estaba compensado o en algunos incluso acelerará ligeramente, y entrará en un descenso con velocidad constante.

En el gráfico de la figura 18, vemos un ejemplo de como con una potencia del 70% y una velocidad de 90 nudos el avión vuela nivelado mientras que con esa misma velocidad pero solo el 40% de potencia el aeroplano desciende con una tasa concreta. La misma figura nos introduce también en la segunda clave del descenso: la velocidad. Observamos claramente que para una misma potencia aplicada (40%), la tasa de descenso es distinta para diferentes velocidades.

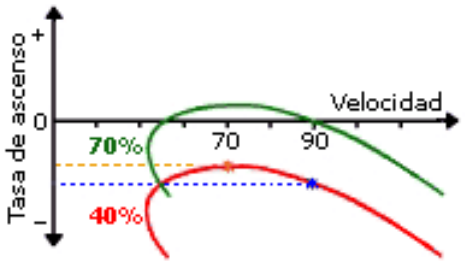


Figura 18. Potencia y velocidad en el descenso.

También se ve que, lo mismo que en ascenso, de todas las velocidades posibles hay un rango que proporciona la mejor tasa de descenso, rango que corresponde a velocidades próximas al punto más alto de la curva de potencia. En el ejemplo de la figura el rango estaría alrededor de los 70 nudos, resultando que con esta velocidad la tasa de descenso es menor por ejemplo que con 50 o 90 nudos.

Las claves del descenso son:

- Descender requiere menos potencia que volar nivelado.
- Disminuir la potencia manteniendo la velocidad (ángulo de ataque) hace que el avión descienda.
- Con una misma potencia, de todas las velocidades posibles la mejor tasa de descenso (el menor hundimiento) se obtiene con una específica, la cual corresponde a un ángulo de ataque concreto.

- La mejor tasa de descenso se obtiene (lo mismo que la de ascenso) con una combinación adecuada de potencia y velocidad.

2.2.4. Giros

El giro se utiliza para cambiar la dirección de vuelo del avión. Un giro preciso y nivelado consiste en un cambio de dirección, manteniendo el ángulo de alabeo deseado, sin derrapar ni resbalar, mientras se mantiene la altitud de vuelo. Aerodinámicamente, el giro es probablemente la maniobra básica más compleja e implica la utilización coordinada de todos los controles primarios: alerones, timón de profundidad, y timón de dirección, además del control de potencia.

La clave del giro está en alabear el avión, esto es, girando el timón de dirección hacia el lado que queramos hacer el giro. Al realizar el giro también se usa el timón de profundidad (controlador de la guiñada), es decir, se pisa el pedal del lado hacia el que giramos. Pero en realidad es el alabeo lo que hace girar el avión, debido a que al inclinarse el avión, la fuerza de sustentación (figura 19) del mismo (fuerza vertical en sentido contrario a la fuerza gravitatoria) pasaría a ser inclinada, es decir, se descompone en una fuerza vertical y en otra horizontal, ver figura 20, pues ésta última es la que hace que el avión gire, la guiñada es solamente una ayuda o complemento. Es más, algunos modelos de aviones solo necesitan alabear para girar, sin ayuda de la guiñada, pero al contrario es imposible ya que la fuerza con la que incide el aire sobre el avión es muy pequeña y en vez de girar derraparíamos.

Figura 19. Fuerza de sustentación.

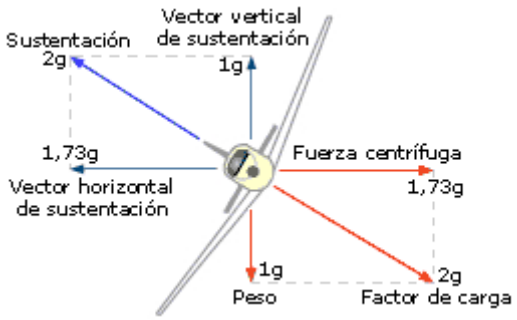


Figura 20. Descomposición de la fuerza de sustentación al alabear el avión.

Los giros se clasifican dependiendo del número de grados de alabeo. Lo importante no es conocer los grados exactos sino la respuesta del avión a cada uno de estos giros.

- Suave (hasta 25º). El amortiguamiento[7] al alabeo del avión tiende a sacarlo del viraje y retornarlo a una posición de nivelado, por lo que para mantenerlo en el giro es necesario mantener el volante de control girado hacia ese lado.
- Medio (hasta 45º). Se compensa la inestabilidad del giro con el amortiguamiento al alabeo y el avión tiende a permanecer en esa posición de viraje sin necesidad de mantener el volante girado, es decir con los alerones nivelados.
- Pronunciado (más de 45º). La inestabilidad del giro es mayor que el amortiguamiento al alabeo y el avión tiende a incrementar el alabeo lo que hace necesario mantener el volante girado al lado contrario del giro para neutralizar la tendencia al "sobrealabeo".

Otras claves para la maniobra del giro serían:

- A mayor alabeo mayor ratio o tasa de giro (cantidad de grados de giro por unidad de tiempo) y menor radio de giro, es decir, cuanto mas inclinemos el avión más cerrado será el giro.
- Para un alabeo constante, si aumentamos la velocidad del avión, la tasa de giro se reduce y el radio aumenta. La cantidad de alabeo debe de ser directamente proporcional a la velocidad del avión.
- Para girar con la máxima tasa de giro posible y con el radio más corto, el avión ha de volar con la velocidad más baja para ese ángulo de alabeo.

2.2.5. Vuelo recto y nivelado

Volar recto y nivelado significa mantener el avión en la dirección establecida mientras se mantiene una altitud constante. No es que sea una maniobra difícil pero tampoco puede considerarse fácil, ya que requiere muchas aunque pequeñas correcciones

[7] El amortiguamiento al alabeo tiende a retornar el avión a su posición de alas niveladas. El trabajo de los alerones consiste precisamente en neutralizar y rebasar ese amortiguamiento. Si en vuelo recto y nivelado el avión alabea ligeramente y soltamos el volante de control el avión vuelve a una posición de alas niveladas.

tanto en la potencia como en las diferentes actitudes del avión (cabeceo y alabeo) para mantener una dirección rectilínea y una altura y velocidad constantes. Por ejemplo, si queremos aumentar la velocidad tenemos que aumentar la potencia, esto implica que la potencia que nos sobra de vencer a la resistencia de avance nos haría ascender manteniendo la velocidad que llevábamos anteriormente, esto se soluciona corrigiendo la actitud de morro, es decir, bajando el morro del avión hasta nivelarlo con la línea del horizonte, obteniendo así una velocidad más alta a la anterior y manteniendo el vuelo recto y nivelado. Para reducir la velocidad bastaría con hacer justo lo contrario, reducir potencia y aumentar ángulo de ataque.

En este ejemplo se ha visto claramente como un avión no puede controlar su velocidad mediante un único controlador o acelerador, como por ejemplo en un coche. La velocidad en un avión es el resultado de la potencia y de la actitud.

Queda claro que en vuelo recto y nivelado podemos volar a diferentes velocidades, dependiendo de las combinaciones de potencia y actitud, la figura 21 muestra las tres categorías, aunque los grados son sólo un ejemplo.

Figura 21. Relación entre ángulo de ataque y velocidad en vuelo recto y nivelado

Para mantener el avión a una altura constante hay que revisar frecuentemente el altímetro, si ganamos altura bajamos el morro del avión y si perdemos altura lo subimos. Pero también podemos usar como referencia por ejemplo uno o varios puntos en el cristal y mantenerlos en línea constante entre el horizonte y nuestros ojos. El horizonte artificial muestra la actitud de morro del avión respecto al horizonte real, pero esta información no dice (al menos directamente) si mantenemos el nivel de vuelo, puesto que como hemos visto se puede volar nivelado con diferentes combinaciones de actitud y potencia.

Mantener el avión en la dirección deseada es más fácil que mantenerlo a una altura y velocidad constantes. Basta con poner el avión en la dirección deseada y mantener las alas niveladas con respecto al horizonte, es decir, 0 º de alabeo. Cualquier grado de alabeo hará que el avión entre en una trayectoria curvilínea. El mejor instrumento para mantener un vuelo recto es el indicador de dirección; cualquier pequeño alabeo provocará en este instrumento un cambio de rumbo.

2.3. Teoría de Control

El control automático ha jugado un papel vital en la evolución de la ingeniería y la ciencia. Además de tener una extrema importancia en vehículos espaciales, guiado de

proyectiles y sistemas de pilotaje de aviones, el control automático también se ha convertido en una parte importante e integral de los procesos industriales modernos, como controlar la presión, temperatura, viscosidad, etc.

La Teoría de Control o regulación automática es una rama de la ingeniería que estudia el comportamiento de los sistemas dinámicos (sistemas que evolucionan en el tiempo) y se ocupa del control de los procesos (pueden ser biológicos, económicos, de ingeniería, etc.) que incluyen dichos sistemas.

2.3.1. Historia

Las primeras aplicaciones de control con realimentación datan del tiempo de los griegos, como se dice en [4], en el que se realizaron mecanismos regulados por flotador. Herón de Alejandría (siglo I D.C.) publicó un libro llamado "Pneumática", en el que mostraba varias formas de mecanismos de agua mediante reguladores con flotador. Los árabes recogieron estos conocimientos y perfeccionaron la construcción de relojes de agua hasta 1.258 (toma de Bagdad por los mongoles). Pero el primer regulador con realimentación automática usado en un proceso industrial y que para muchos representa el primer sistema de control automático y punto de partida de esta ciencia, fue el regulador centrífugo de James Watt como el de la figura 22, desarrollado en 1.770 para controlar la velocidad de una máquina de vapor. Este sistema regulador era propenso a las oscilaciones y gracias a esto fue objeto de muchas investigaciones.

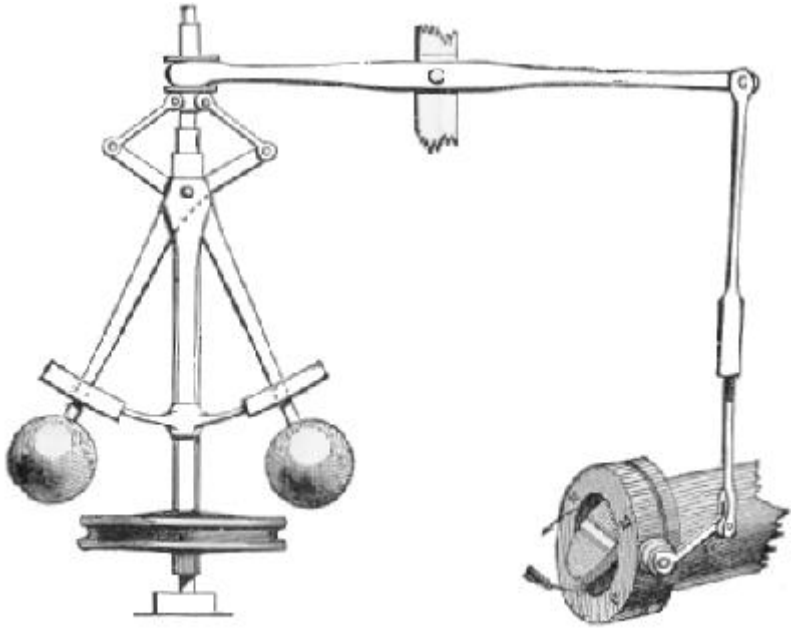


Figura 22. Regulador de velocidad de James Watt.

En 1.868, J.C. Maxwell publicó "On governors" donde propuso una solución al problema de los reguladores centrífugos utilizando una ecuación diferencial y analizando las condiciones de estabilidad para un sistema de hasta tercer orden. Edward John Routh, generalizó los resultados de Maxwell para los sistemas lineales en general. Este resultado

se conoce con el nombre de Teorema de Routh-Hurwitz (1.895). En 1.927, Harold Black inventó el amplificador con realimentación negativa. Harry Nyquist, compañero de trabajo de Black, analizó la estabilidad del amplificador realimentado en el dominio de la frecuencia, utilizando el concepto de anchura de banda y otras variables dependientes de la frecuencia.

Una fecha importante para la ingeniería de control fue el año 1.934, en que Hazen publicó el primer trabajo analítico sobre diseño de sistemas en lazo cerrado. Durante la II Guerra Mundial se realizó un gran avance, ya que fue necesario diseñar y construir pilotos automáticos para aeroplanos, sistemas de dirección de tiro para cañones y otros sistemas militares basados en los métodos de control por realimentación. En la década de los cuarenta se desarrollaron nuevos métodos analíticos y matemáticos, como el uso de las funciones de transferencia y los diagramas de bloque, nuevos procedimientos gráficos para el estudio de la estabilidad en el dominio de la frecuencia, síntesis de redes y el método del lugar de las raíces.

Entre 1.950 y 1.960 se hizo extensivo el uso de la Transformada de Laplace y el plano de la frecuencia compleja, también fue posible la utilización de los ordenadores analógicos y digitales como componentes de control. Estos nuevos elementos proporcionaron una capacidad para calcular con rapidez y exactitud que no existían antes para el ingeniero de control. También en esta época aparece el nuevo concepto de variables de estado.

La Era espacial dio otro impulso a la ingeniería de control. Se diseñaron sistemas de control muy complejos y altamente precisos para proyectiles y pruebas espaciales. Se amplían los métodos en el dominio del tiempo, espacios de estado y sistemas no lineales. Se amplían los trabajos de Liapunov sobre la estabilidad, se empiezan a estudiar los sistemas de control óptimo y adaptativo.

La teoría del control sigue incorporando nuevos métodos que se aplican a nuevas ramas del saber. Una nueva área de la ingeniería de control es la "robótica industrial", donde ordenadores, sensores y mecanismos se integran de un modo inteligente para producir autómatas[8] cada vez más sofisticados. Aunque no cabe duda de que en la actualidad, las técnicas de Inteligencia Artificial son las que más atención están acaparando en el mundo de la ingeniería de control. Los espectaculares avances en la Electrónica Integrada crean saltos cualitativos y generan nuevas expectativas: el futuro de la nanotecnología, computación molecular y cuántica, son algunas de las puertas que la ciencia está intentando abrir y que a buen seguro supondrán nuevos saltos cualitativos y cuantitativos en el desarrollo de la ingeniería de control.

[8] Autómatas: máquinas que controlan procesos industriales como fabricación, producción, etc.

2.3.2. Componentes básicos de un sistema de control

Los componentes básicos de un sistema de control son los objetivos de control, el sistema de control y los resultados o salidas. La figura 23 muestra la relación básica entre estos tres componentes.

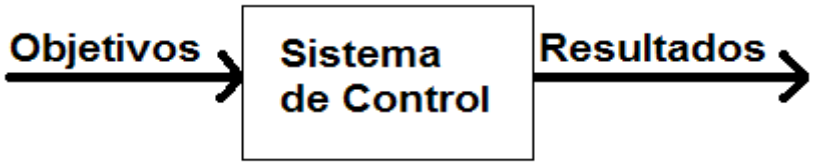


Figura 23. Componentes básicos de un sistema de control.

Los objetivos se pueden identificar como entradas o señales actuantes, denominadas como *u*, y los resultados se llaman salidas o variables controladas *y*. En general, el objetivo de un sistema de control es controlar las salidas mediante las entradas a través de los elementos del sistema de control. Los sistemas de control en lazo abierto constituyen el tipo más sencillo y económico de los sistemas de control. Consta de dos partes: el controlador y el proceso controlado o planta. Una señal de entrada *r* se aplica al controlador, cuya salida actúa como señal actuante *u*, ésta señal controla el proceso de tal forma que la variable controlada *y*, se desempeñe de acuerdo a lo preestablecido. En los casos simples el controlador puede ser un amplificador, filtro, etc. En los casos más complejos puede ser una computadora. Debido a la economía y simplicidad de los sistemas en lazo abierto, se les encuentra en aplicaciones no críticas.

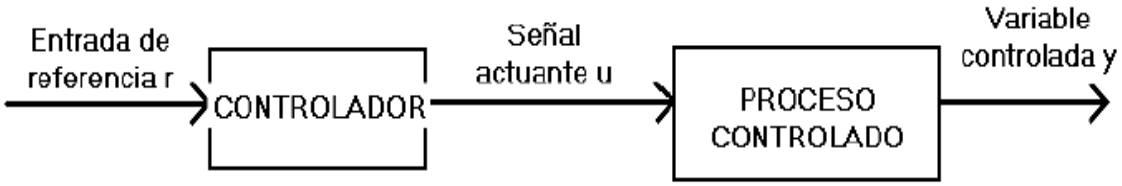


Figura 24. Sistema de control en lazo abierto.

Para que un sistema de control en lazo abierto como el de la figura 24, sea más exacto y adaptable es necesaria una conexión o realimentación desde la salida hacia la entrada del sistema. Para obtener un control más exacto, la señal controlada debe de ser realimentada y comparada con la señal de referencia *r*, y se debe enviar una señal actuante proporcional a la diferencia de la entrada y la salida a través del sistema para corregir el error. Un sistema con una trayectoria o más de realimentación se denomina sistema en lazo cerrado.

En la figura 25 se muestra un sistema simple en lazo cerrado, donde se puede ver un comparador o detector de error, así como la realimentación de la señal de salida para poder compararla con la señal de referencia. Cualquier diferencia entre estas dos señales constituye una señal de error o de actuación. El controlador recibe esta señal y modifica la

variable manipulada de tal forma que obliga al proceso o planta a reducir el error original. El control realizará su acción correctora hasta que el error sea nulo. De este modo se obtiene un sistema de control con un comportamiento completamente automático, ya que no es precisa la actuación humana para adaptar la salida a la entrada, de ahí el nombre con el que también se conoce a los sistemas realimentados o en lazo cerrado: sistemas de control automático.

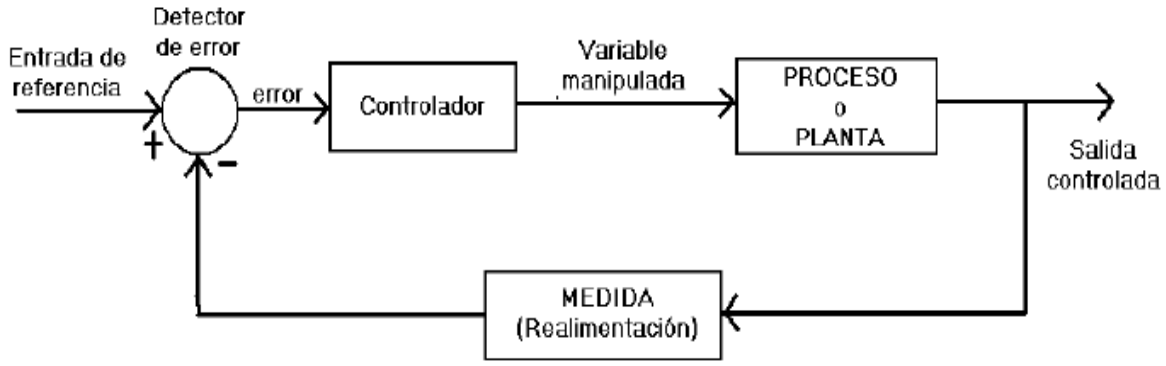


Figura 25. Sistema de control en lazo cerrado.

Algunas ventajas del control realimentado frente al de lazo abierto podrían ser:

- Incremento de la exactitud: el sistema de ciclo cerrado puede configurarse para llevar a cero la diferencia de error entre las señales de referencia y salida.
- Pequeña sensibilidad a los cambios en los componentes: el sistema puede diseñarse para tratar de obtener un error cero, a pesar de los cambios de la planta o proceso.
- Reducidos efectos de las perturbaciones: se pueden atenuar notablemente los efectos de las perturbaciones al sistema.
- Incremento en la rapidez de respuesta y en la anchura de banda: la retroalimentación puede emplearse para aumentar la gama de frecuencias sobre la cual el sistema responderá, y hacer que responda más satisfactoriamente.

En cuanto al lazo abierto, presenta las siguientes ventajas frente al cerrado:

- Montaje simple y facilidad de mantenimiento.
- Mayor economía que un sistema de lazo cerrado equivalente.
- Habitualmente no presenta problemas de estabilidad.
- Es conveniente cuando es difícil o más caro medir la salida.

Por el contrario, este sistema también presenta las siguientes desventajas frente a la retroalimentación:

- Las perturbaciones y modificaciones en la calibración introducen errores y la salida puede diferir de la deseada.

- Para mantener la calidad necesaria a la salida, periódicamente hay que efectuar una re calibración.

2.3.3. El regulador PID

Una vez vistos los tipos de sistemas de control, vamos a ver los tipos de controladores o reguladores, centrándonos solamente en el regulador PID (figura 26), ya que es el controlador que hemos utilizado en este proyecto.

Mirar [5]. Cada sistema de control debe garantizar en primer lugar la estabilidad[9] del comportamiento en lazo cerrado. En función de las condiciones que deseamos imponer a la salida, se debe escoger una estrategia de control (regulador) u otra. Algunos de los sistemas que más se utilizan en la actualidad son por ejemplo el regulador ON-OFF, los autómatas programables, el regulador PID, el control multivariable, el control difuso, los sistemas SCADA, los sistemas distribuidos de control (SDC), asignación del lugar de los polos, el control óptimo, etc. Vamos a centrarnos en el regulador PID.

El algoritmo de control más ampliamente extendido es el PID, pero existen muchos otros métodos que pueden dar un control de mayor calidad en ciertas situaciones donde el PID no responde a la perfección. El PID da buenos resultados en la inmensa mayoría de casos y tal vez es por esta razón que goza de tanta popularidad frente a otros reguladores teóricamente mejores. Los diseñadores de software de regulación pretenden que programar los nuevos sistemas de control sea tan fácil y familiar como el PID, lo que posibilitaría una transición sin dificultades. Sea cual sea la tecnología de control, el error de regulación es la base a partir de la cual actúa el PID y cuanto más precisa sea la medida, mejor se podrá controlar la variable en cuestión. Esta es la razón por la que el sensor es el elemento crítico del sistema. Un regulador proporcional-integral-derivativo o PID tiene en cuenta el error, la integral del error y la derivada del error. La acción de control se calcula multiplicando los tres valores por una constante y sumando los resultados. Los valores de las constantes, que reciben el nombre de constante proporcional, integral y derivativa, definen el comportamiento del regulador.

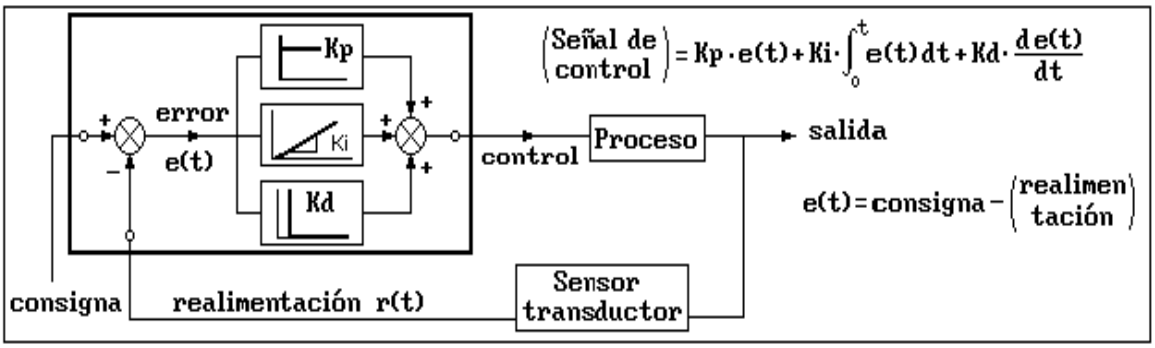


Figura 26. Sistema de control con regulador PID.

[9]Estabilidad de un sistema en lazo cerrado: un sistema es estable si, ante cualquier entrada acotada en un intervalo cualquiera de tiempo, la salida también está acotada.

La acción proporcional hace que el PID responda enérgicamente cuando el error es grande, condición que aparentemente es necesaria y suficiente, pero no es así en la mayoría de los casos por dos razones fundamentales:

1. Muchas veces la variable regulada aumenta o disminuye si no existe una acción que la mantenga invariable, por ejemplo un cuerpo desciende por gravedad, un fluido disminuye su nivel o presión si existe una vía de salida, un resorte tiende a adoptar la posición de mínima energía, etc. Cuando la variable se acerca al punto de consigna la acción proporcional se debilita y no vence la tendencia de la variable, alcanzando un reposo antes de lo previsto y por lo tanto manteniendo un error permanente.

2. Aunque el error disminuye al aumentar la constante proporcional, no es correcto aumentar dicha acción todo lo necesario para conseguir un error muy pequeño, porque toda magnitud tiene cierta inercia a permanecer en su estado de reposo o de variación constante, responde desde el primer momento a la acción de control pero con cierto retraso o pereza, por ejemplo no podemos detener un móvil de forma instantánea, un motor no alcanza inmediatamente su velocidad nominal, etc. Si la acción proporcional es grande, la variable regulada se acercará al punto de consigna demasiado deprisa y será inevitable un sobrepasamiento.

Por la primera razón expuesta se deduce la conveniencia de añadir otra acción que responda si el error se mantiene a lo largo del tiempo, algo parecido a una memoria histórica que tenga en cuenta la evolución del error. Así actúa la acción integral, que se encarga de mantener una respuesta cuando el error se anula, gracias al error que existió en el tiempo ya pasado. Esta respuesta mantenida contrarresta la tendencia natural de la variable.

Por la segunda razón expuesta, se comprende la necesidad de añadir otra acción que contrarreste la inercia del proceso, frenándolo cuando evoluciona demasiado rápido y acelerándolo en caso contrario, algo parecido a una visión de futuro que se anticipa a lo que previsiblemente ocurrirá. Así actúa la acción derivativa, conocida también como anticipativa por ese motivo.

La parte problemática es la sintonización, es decir, dar valores a las constantes que representan las intensidades con las que actúan las tres acciones. La solución a este problema no es trivial puesto que depende de cómo responde el proceso a los esfuerzos que realiza el regulador para corregir el error.

Si se considera un proceso con un retraso considerable y el error varía rápidamente por un cambio en consigna o en carga (perturbaciones), el regulador reaccionará de inmediato, pero como el sistema responde lentamente, la acción integral empezará a tomar mucha importancia y cuando llegue al punto de consigna mantendrá una acción muy intensa basada en el error existente durante el tiempo de retraso y produciendo un

rebasamiento. En los procesos con mucho retraso, la acción integral debe ser pequeña según esta consideración. Si el proceso presenta poco retraso, el término integral tendrá poco peso respecto a las otras dos acciones porque los errores existen poco tiempo. En cambio, el término derivativo será de importancia porque el error varía con rapidez, debiendo utilizar una constante derivativa pequeña para evitar reacciones exageradas.

$Ki = \frac{Kp}{Ti} \qquad Kd = Kp \cdot Td$	$\left(\begin{matrix}\text{Señal de} \\ \text{control}\end{matrix}\right) = Kp \cdot e(t) + \frac{Kp}{Ti} \cdot \int_0^t e(t)\,dt + Kp \cdot Td \cdot \frac{d\,e(t)}{dt}$

Figura 27. Constante integral y derivativa.

En la figura 27 vemos la representación matemática de la señal de control, que es la suma de las tres acciones ya conocidas, es decir, un valor proporcional al error, mas la constante integral Ki por la integral del error, mas la constante derivativa Kd por la derivada del error. Las acciones integral y derivativa no se ajustan generalmente por sus constantes sino por un tiempo integral Ti y un tiempo derivativo Td que dependen de la constante proporcional Kp. Queda claro entonces que los tres parámetros de un regulador PID son Kp, Ti y Td; y que los tres componentes de un controlador PID son:

Acción proporcional:

$$Kp \cdot e(t) \qquad (1)$$

Acción integral:

$$\frac{Kp}{Ti} \int_0^t e(t)dt \qquad (2)$$

Acción derivativa:

$$Kp \cdot Td \frac{de(t)}{dt} \qquad (3)$$

Y que el peso de la influencia que tiene cada una de estas partes en la suma final, viene dado por la constante proporcional Kp, el tiempo integral Ti y el tiempo derivativo Td. También queda claro que la acción proporcional por sí sola, es una estrategia de control rápida de calcular, debido a que sólo hay que ajustar la constante Kp, y además genera una respuesta instantánea, pero que, sin embargo posee una característica indeseable como es el error en estado estacionario u offset. En la figura 28 se muestra un ejemplo de un regulador PID en el que sólo se ha activado la acción proporcional, por tanto, se denomina controlador o regulador proporcional. La consigna o referencia es la línea azul, vemos que al principio existen varias oscilaciones y cuando pasamos al estado

estacionario se puede observar el offset, es decir, la señal se queda en valores muy cercanos a la referencia pero nunca va a llegar a su objetivo.

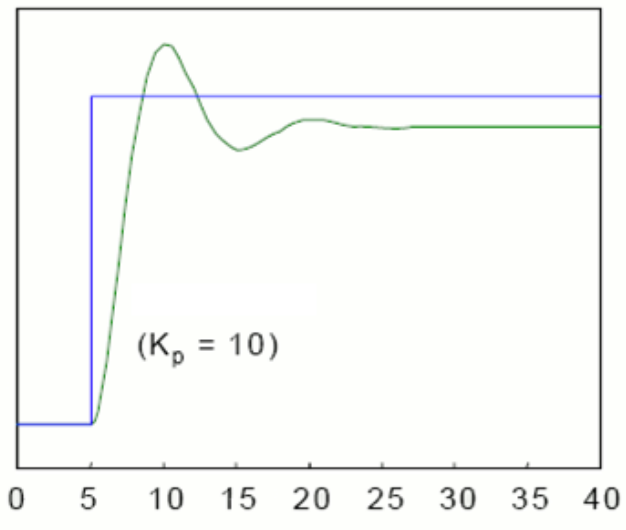


Figura 28. Controlador proporcional.

La acción integral o integrativa elimina el offset, pero se obtiene una mayor desviación del setpoint o referencia, la respuesta es más lenta y el periodo de oscilación es mayor que en la acción proporcional. En la figura 29 se observan perfectamente estas afirmaciones.

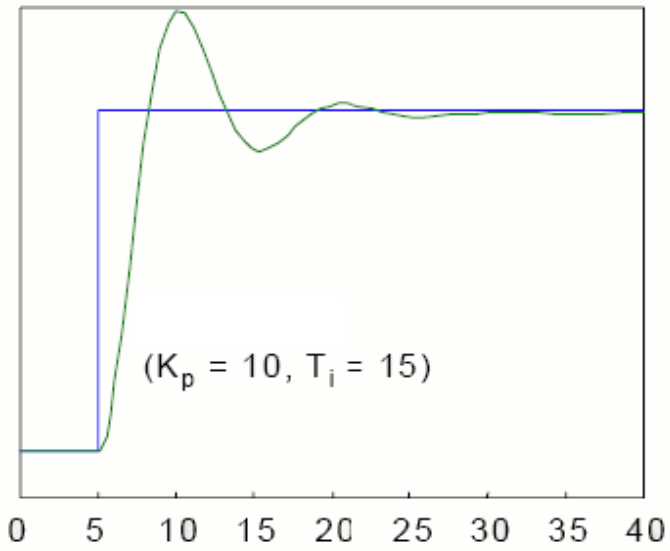


Figura 29. Controlador Proporcional-Integral (PI).

La acción derivativa da una respuesta proporcional a la derivada del error, es decir, la velocidad de cambio del error. Esta acción elimina el exceso de oscilaciones, aunque no elimina el offset. Sólo se manifiesta cuando hay un cambio en el valor absoluto del error, o sea, cuando cambia de signo. Si el error es constante no actúa. Ver figura 30.

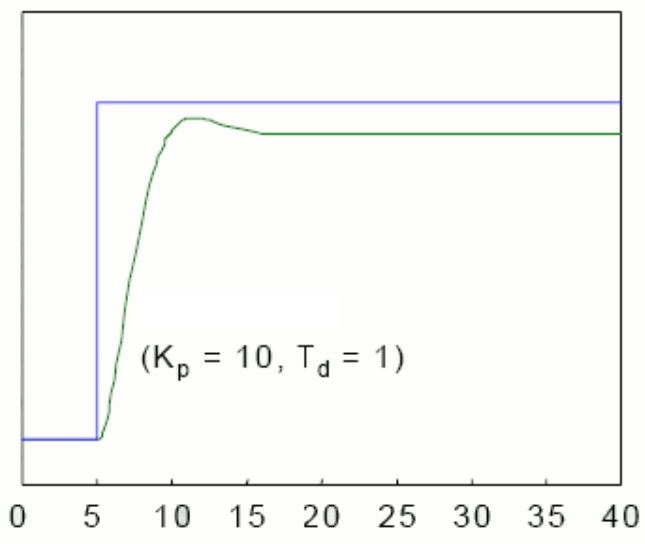


Figura 30. Controlador Proporcional-Derivativo (PD).

La acción de control proporcional integral derivativa (PID) reúne las ventajas de cada una de las tres acciones de control individuales (figura 31). Esto es posible sumando las tres acciones, como se ha visto anteriormente. Este sistema no es eficiente para sistemas con retardos muy grandes.

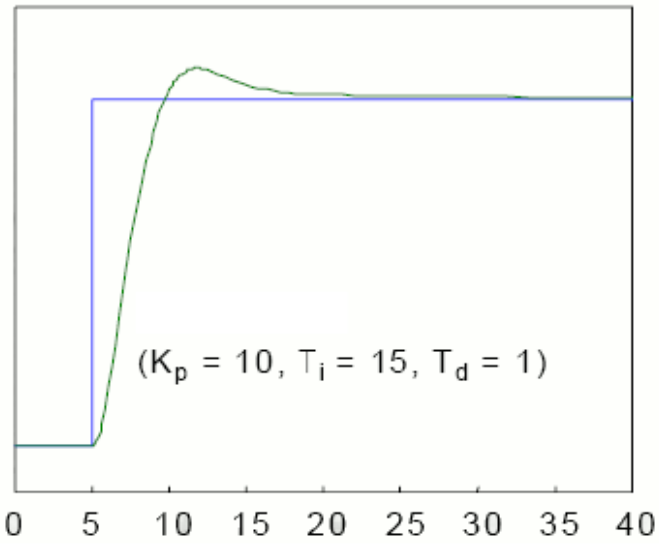


Figura 31. Controlador PID.

2.3.4. Ajustando el regulador PID

Vamos a ver cómo se ajusta un controlador PID. El objetivo de los ajustes de los parámetros PID es lograr que el bucle de control corrija eficazmente y en el mínimo tiempo los efectos de las perturbaciones; se tiene que lograr la mínima integral de error. Si los parámetros del controlador PID (la ganancia del proporcional, integral y derivativo) se eligen incorrectamente, el proceso a controlar puede ser inestable, por ejemplo, que la salida de este varíe, con o sin oscilación. El método más efectivo generalmente requiere del desarrollo de alguna forma del modelo del proceso, luego elegir P, I y D basándose en los

parámetros del modelo dinámico. Los métodos de ajuste manual pueden ser muy ineficientes. La elección de un método dependerá de si el lazo puede ser "desconectado" para ajustarlo, y del tiempo de respuesta del sistema. Si el sistema puede desconectarse, el mejor método de ajuste a menudo es el de ajustar la entrada, midiendo la salida en función del tiempo, y usando esta respuesta para determinar los parámetros de control. Ziegler y Nichols también describieron una técnica de ajuste en lazo cerrado pero con la parte integral y la parte derivativa anuladas. La constante proporcional (Kp) es incrementada hasta que una perturbación causa una oscilación mantenida (que no se anula). El valor más pequeño de la constante proporcional que causa tal oscilación se denomina constante proporcional crítica (Kcrit). El período de esas oscilaciones es el llamado período de oscilación crítico (Tc). Véase figura 32. En el segundo recuadro de la figura tenemos una tabla con las fórmulas de cálculo que se ajustan al criterio de amortiguamiento 1/4. Es por tanto apropiado cuando deban prevenirse grandes desviaciones frente a cambios en consigna y en carga y los tiempos de respuesta y estabilización son aceptables al cambiar la consigna. Este método da resultados precisos pero puede suponer mucho tiempo de prueba y error hasta conseguir la oscilación mantenida, existiendo además el peligro de desestabilizar el sistema. Se recomienda generalmente la estimación en lazo abierto (a pesar de ser un método aproximado) porque es más fácil y porque abarca un mayor número de criterios. El método de ajuste en lazo abierto no se va a explicar en esta memoria.

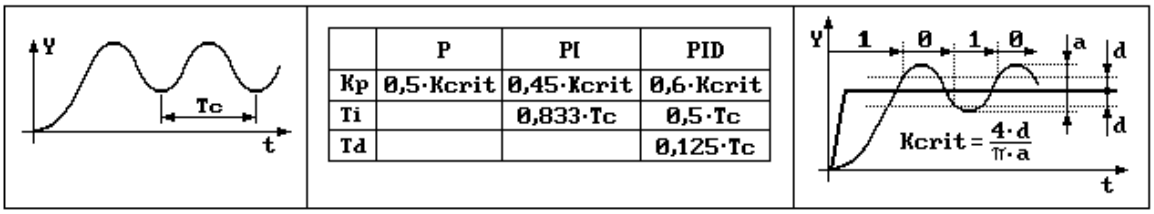


Figura 32. Ajuste de Ziegler y Nichols (dos primeras) y del método del relé.

Un método más sencillo, pero solo aproximado, es el método del relé (último recuadro de la figura anterior). Consiste en sustituir el controlador proporcional por un control todo o nada, que satura o anula la acción de control sobre el proceso. Aplicando una consigna constante, se utiliza la señal de error para decidir los momentos de conexión y desconexión (+d y -d en la figura) y cuando la salida alcanza una frecuencia de oscilación estable, el período coincide aproximadamente con el valor Tc ya explicado. El valor Kcrit se calcula aproximadamente con la fórmula que tenemos en la figura y finalmente podemos aplicar las mismas fórmulas de la tabla central.

2.3.5. Discretización de controladores PID

Visto en [6]. Hoy en día prácticamente el 100% de los controladores automáticos que funcionan en la industria son digitales. La figura 33 representa un sistema de control digital, también llamado control por computador:

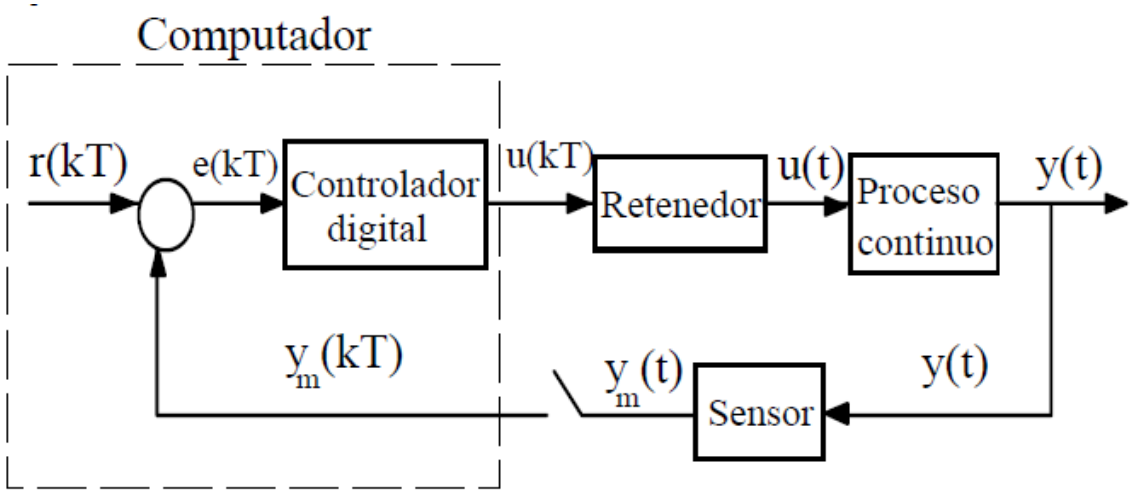


Figura 33. Sistema de control digital o control por computador.

El computador obtiene una medida del sensor cada *T* segundos, y calcula con ella el valor de la acción de control (*u(kT)*). Figura 34.

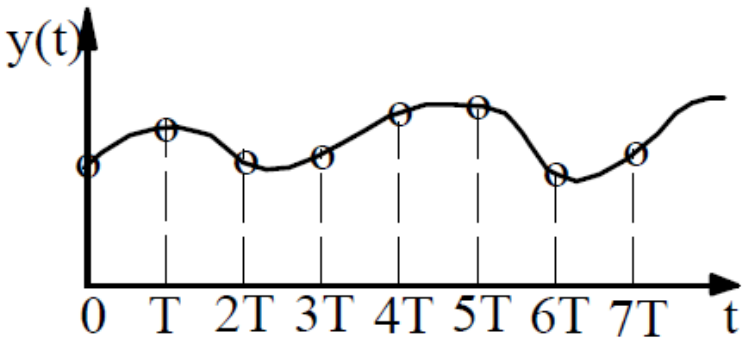


Figura 34. Muestreo cada T segundos.

El bloque retenedor no es más que un convertidor digital analógico, que da como salida una tensión *u(t)* constante hasta el siguiente periodo. Figura 35.

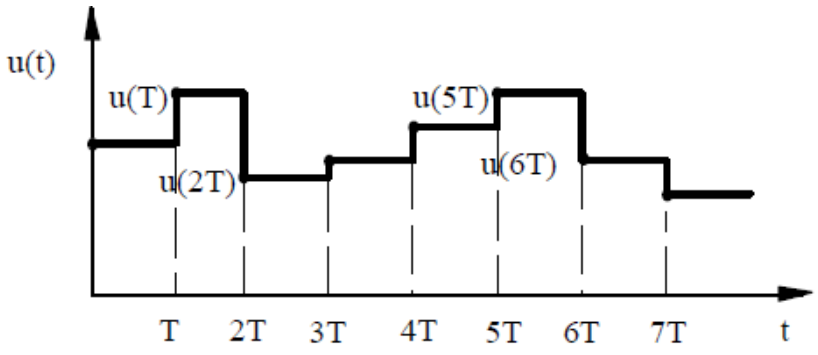


Figura 35. Bloque retenedor.

Las ventajas del control por computador respecto del control analógico son:

- Permite comunicación en red (control centralizado).
- Incorpora otras funciones

- Monitorización
- Almacenamiento de variables (generación de informes)
- Tratamiento de alarmas
- Supervisión
- Es mucho más flexible
- Permite controladores más avanzados que requieren cálculos complejos:
  - Controladores adaptativos.
  - Control basado en lógica borrosa.
  - Sistemas expertos.
  - Controladores no lineales.
- Son equipos comerciales estándar
- Cada vez son más baratos y más potentes.

Para poder implementar en un computador un controlador PID continuo (diseñado por ejemplo mediante el lugar de las raíces), es necesario obtener una ecuación en diferencias discreta a partir de la ecuación diferencial continua que define el controlador.

Se llama discretización a la acción de obtener una ecuación en diferencias (controlador discreto) que aproxime el comportamiento de una ecuación diferencial (controlador continuo). La ecuación diferencial de un regulador PID continuo es:

$$u(t) = Kp\left(e(t) + Td\frac{de}{dt} + \frac{1}{Ti}\int_0^t e(\tau)d\tau\right) \qquad (4)$$

La aproximación discreta consiste en aproximar la ecuación diferencial anterior obteniendo *u(kT)* a partir de los valores de *e(t)* en los periodos de muestreo (es decir, a partir de *e(T), e(2T), etc.*). La aproximación más fácil de la derivada es:

$$\frac{de(kT)}{dt} \approx \frac{e(kT) - e((k-1)T)}{T} \qquad (5)$$

Mientras que la integral se puede aproximar como:

$$\int_0^{kT} e(t)dt \approx \sum_{j=0}^{k-1} e(jT)\cdot T \qquad (6)$$

De esta forma se tendría:

$$u(kT) \approx Kp\cdot e(kT) + Kp\cdot Td\frac{e(kT) - e((k-1)T)}{T} + \frac{Kp}{Ti}\sum_{j=0}^{k-1} e(jT)\cdot T \qquad (7)$$

Es decir:

$$u_k \approx Kp \cdot e_k + Kp \cdot Td\frac{e_k - e_{k-1}}{T} + \frac{Kp \cdot T}{Ti}\sum_{j=0}^{k-1} e(jT) \qquad (8)$$

Capítulo 3. Descripción de la solución

3.1. Protocolo de comunicación con FlightGear

El envío y la recepción de los datos de interés entre la aplicación y el simulador de vuelo lo hacemos mediante el uso de dos de los protocolos que vienen incorporados en el simulador FlightGear (FG). Estos protocolos nos dicen el orden que ocupan los parámetros de cada variable en la trama y el tipo de variable de la que se trata, conocido esto no existe problema alguno en la identificación, ni en la manipulación de las variables de interés.

Para empezar, la aplicación recibe dos clases de estructuras o de datos del simulador de vuelo y envía una de ellas previamente modificada a partir de los datos de la otra estructura. Se puede ver en la figura 36.

Una clase sería la información de vuelo, es decir, los parámetros de las variables del avión durante el vuelo como son la altura, rumbo, velocidad, etc., esta clase la usamos no sólo para dar información útil a la aplicación, sino también para imprimir dicha información por pantalla, tanto en la interfaz gráfica de usuario (GUI) como en una serie de gráficas. La otra clase de datos que recibimos son las variables o controladores que gobiernan los parámetros de las variables anteriormente citadas, por ejemplo la variable *altura* estaría gobernada directamente por la variable *elevator* o la *aceleración* se modificaría por medio de la variable *throttle* (acelerador). Esta clase es la que envía la aplicación al simulador una vez que ha modificado los parámetros de las variables controladoras que nos interesa modificar, haciendo que el avión maniobre de forma automática.

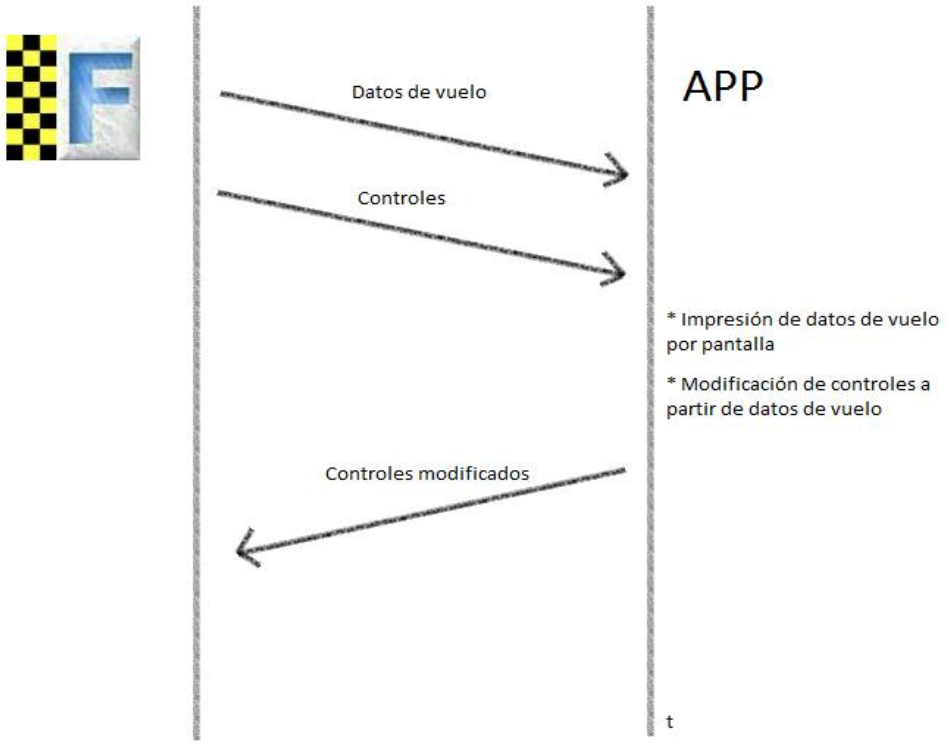


Figura 36. Comunicación entre FG y la aplicación.

3.1.1. Implementación del protocolo de comunicación

Hay que comenzar diciendo que la comunicación entre la aplicación y FG se realiza mediante sockets, que es una de las opciones de Entrada/Salida con las que cuenta el simulador. También nos da a elegir entre sockets TCP o UDP, en nuestro caso nos quedamos con los UDP ya que se trata de una aplicación en tiempo real y no tenemos como principal objetivo que no se pierda ningún paquete o que lleguen en perfecto orden.

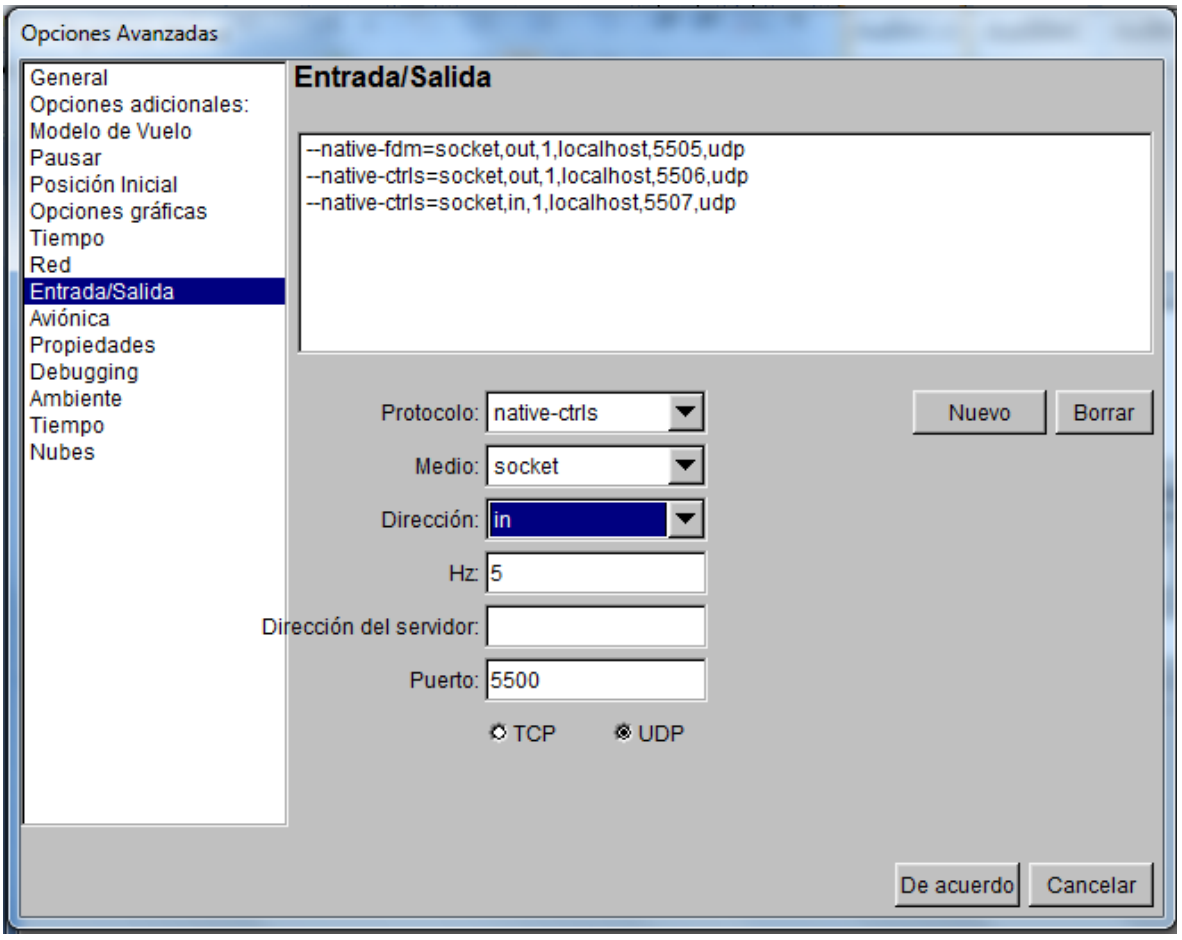


Figura 37. Opciones de Entrada/Salida de FG.

Además, como vemos en la figura 37, también podemos elegir el protocolo. Esta parte la veremos más detalladamente en el punto 3.2. Para saber qué protocolo o protocolos debíamos elegir, descargamos el código fuente de FG y tras ver en qué consistía cada uno nos quedamos con los protocolos *native_fdm* y *native_ctrls*. Estos protocolos operan sobre unas estructuras de datos llamadas *net_fdm* y *net_ctrls* respectivamente, es decir, FG actualiza su *árbol de propiedades internas*[10] desde estas estructuras (en caso de ser recibidas) o actualiza estas estructuras desde su "*property tree*" (justo antes de ser enviadas). Para ver el contenido de estas estructuras ir al Anexo I o descargar el código fuente de FG desde su propia web.

[10] Property tree en inglés, es considerado el sistema nervioso central del simulador, contiene los parámetros de todas las variables existentes en FG, pudiéndose modificar en tiempo de ejecución.

La casilla "dirección" se refiere a si el socket es *in* de entrada (para recibir), salida *out* (para enviar) o bidireccional *bi* (enviar y recibir).

En la casilla "Hz" introducimos el número de emisiones y/o recepciones que se van a hacer cada segundo y por último, en "Dirección del servidor" escribimos la dirección ip del pc transmisor o receptor. En caso de dejar esta casilla sin rellenar, la dirección por defecto será "localhost", que también puede ser escrita.

En nuestra aplicación tenemos la clase *gestorSocket*, en la cual creamos tres *DatagramPacket*, dos son para la recepción y uno para la transmisión,. El espacio de memoria que debíamos reservar en los búferes de tipo bytes lo sabíamos de antemano debido a que, realizamos varias capturas de las tramas con un programa sniffer llamado Wireshark. Mirar Anexo III.

```
dpFdm=new DatagramPacket(buff_fdm,408);
dpCtrls=new DatagramPacket(buff_ctrls,744);
try {
dpSend=new DatagramPacket(buff_send,744,InetAddress.getLocalHost(),5507);

}
```

flightGearController es el hilo que tiene como función más importante la de recibir y enviar datos constantemente. Mediante la función *abrirSockets()* de este hilo, invocada desde la clase *flightGearGUI*, llamamos a las funciones que crean los sockets:

```
public void abrirSockets(){
	gs.abrirSocketCtrls();
	gs.abrirSocketFdm();
	gs.abrirSocketSend();
}
```

Como la función para crear los sockets delega en las 3 funciones de apertura de la clase *gestorSocket*, es en ésta donde se crean los 3 sockets de esta manera:

```
public void abrirSocketFdm(){

	try {
	socketFdm = new DatagramSocket(5505);
		} catch (SocketException e) {
			e.printStackTrace();
		}/* catch (UnknownHostException e) {
			e.printStackTrace();
		}*/
		System.out.println("Abriendo socket fdm...");
}//abrirSocket
```

Asignamos a cada socket un número de puerto distinto para tener una recepción más *sencilla* y cómoda de las dos tramas, recibiéndolas ya por separado. Una vez abiertos los tres sockets, e invocado el método *run()* de *flightGearController* mediante la orden

start() en la GUI, se reciben los datos y se envían en el bucle infinito que hemos hecho en *fligh*GearController.

```
while(true){

		fdm=gs.recibirFdm();
		gs.recibirCtrls(); //gs es gestorSocket
```

Después imprimimos datos, activamos los controladores PID, etc. Y enviamos:

```
		gs.enviarCtrls();
```

Como se puede ver en el código, las funciones para enviar y recibir las estructuras también delegan en la clase de *gestorSocket:*

```
public void recibirCtrls(){
	try {
		socketCtrls.receive(dpCtrls);
	} catch (IOException e) {
		e.printStackTrace();
	}//catch
	System.out.println("recibiendo datos ctrls...");
	ctrls.deserialize(dpCtrls.getData()); //datos en dp_ctrls
(buff_ctrls)

}//recibirCtrls
```

Creo que es interesante comentar la última línea de esta función, ya que a *deserialize()* (de la cual hablaremos en la página siguiente) se le pasa como entrada un array de bytes. Esto lo obtenemos llamando a la función *getData()* del objeto de la clase *DatagramPacket*. Para el envío de los controles serializamos los datos, los volcamos en el buffer asociado a *DatagramPacket* y los mandamos:

```
public void enviarCtrls(){
	dpSend.setData(ctrls.serialize());//volcamos los datos en buff_send
	try {
		socketSend.send(dpSend);//lo enviamos por la red
		} catch (IOException e) {
		e.printStackTrace();
	}//catch*/
	System.out.println("enviando controles...");
}//enviarCtrls
```

Podríamos haber evitado delegar la apertura de sockets en otras funciones, pero esto es debido a que primero se creó la clase *gestorSocket* y después *flightGearController*, además de esta forma el orden es más lógico ya que todo lo relacionado con la gestión de los sockets es mejor que esté en esta clase.

Vamos a hablar ahora de los datos que se intercambian la aplicación y el simulador. *FGnetFDM* es una clase que es idéntica a la estructura *net_fdm.* Esta clase contiene los datos de vuelo que usaremos para imprimirlos por pantalla y para modificar las variables de control. Cuando recibimos la estructura *net_fdm* del protocolo *native-fdm,* la guardamos en un array de bytes, el cual le pasamos al constructor de la clase FGnetFDM para realizar

la asignación de todas las variables de instancia, obteniendo de forma relativa los parámetros de cada variable. Todo este proceso se realiza en el propio constructor y para poder obtener (*getInt() o getDouble()*) los datos del array de bytes hay que transformarlo primero en un objeto de la clase ByteBuffer mediante la función *wrap(byte[])*.

```
public FGnetFDM(byte [] msg) {

       ByteBuffer data = ByteBuffer.wrap(msg);
        version = data.getInt();
        padding = data.getInt();
        longitude = data.getDouble();
       .........
```

y así hasta asignar todas las variables de instancia de la clase.

La clase *FGnetCtrls* es igual a la estructura *net_ctrls*, que recibimos del protocolo *native-ctrls* y por tanto contiene los parámetros de las variables de control. Debido a que esta estructura la usamos también para enviarla, su implementación es diferente a la de *FGnetFDM*. La primera diferencia es que la asignación de las variables de instancia de *FGnetCtrls* no se realiza en el constructor, sino en la función *deserialize([]byte)*. Esta función no devuelve nada y como argumento requiere un array de bytes (al igual que el constructor de *FGnetFDM*). También hay que usar un objeto *ByteBuffer* para usar el método *getInt()* y *getDouble()*.

```
public void deserialize(byte[] msg) {

       ByteBuffer data = ByteBuffer.wrap(msg);
             //asignacion relativa de las variables.
       version=data.getInt();// increment when data values change
       data.getInt();//para que coincidan datos
                  // Aero controls
       aileron=data.getDouble();                  // -1 ... 1
       elevator=data.getDouble();                  // -1 ... 1
       rudder=data.getDouble();                  // -1 ... 1
       aileron_trim=data.getDouble();             // -1 ... 1
             ..........
```

Si la función *deserialize(byte[])* se usa al recibir la trama, lógicamente al enviarla habrá que usar una función que haga justo lo contrario, es decir, en vez de obtener valores tendrá que ponerlos también de forma relativa. Este sería el proceso de modificación de los datos. Para ello creamos la función *serialize()*, no requiere ningún argumento de entrada y devuelve un array de bytes. Se usa justo antes del envío de la trama, es decir, con esta función rellenamos o asignamos los parámetros a las variables de control y las enviamos por la red. *Serialize()* también se encarga de transformar el array de bytes a ByteBuffer y una vez todas sus variables han sido asignadas se transforma nuevamente en un array de bytes mediante la función *array()* para ser devuelto por *serialize()* y poder ser enviado al simulador por el socket correspondiente en la clase *gestorSocket*, como ya hemos visto en la página 48.

```
public byte[] serialize() {
       ByteBuffer  bb = ByteBuffer.wrap(ctrls);
       //bb.putDouble(alerions);
```

```
        bb.putInt(version);// increment when data values change
        bb.putInt(0);//para que coincidan datos

    // Aero controls
        bb.putDouble(aileron);                  // -1 ... 1

...
```

3.2. Tipos de comunicación

Como se puede comprobar en el menú de FlightGear, concretamente en la pantalla de Entrada/Salida (fig. 38), existen diversos protocolos y varios medios que nos permiten comunicarnos con el simulador.

En el menú desplegable "Medio" podemos elegir el camino o mejor dicho la interfaz de entrada o salida por donde se recibirán o enviarán los datos o "Protocolo" que queramos, aunque hay que decir, que no todos los protocolos funcionan ni en todos los medios, ni en todas las direcciones. Podemos elegir tres medios diferentes: "socket" visto anteriormente en el apartado 3.1.1., "file" o "serial".

Si elegimos "file" estamos diciendo a FlightGear que nos comunicaremos con él por medio de un archivo. Si la dirección es de salida (out), flightgear escribirá y guardará los datos del protocolo que elijamos en dicho archivo. En cambio, si la dirección es de entrada (in), el simulador leerá los datos desde el archivo tantas veces como le señalemos en la casilla Repeat. Cuando usamos el medio file, nos dará flightgear un error si le indicamos que la dirección sea de entrada y salida, es decir, bidireccional. Ya que en el código fuente, al abrir un archivo solo se puede especificar si es para leerlo, modificarlo o bien crearlo y modificarlo.

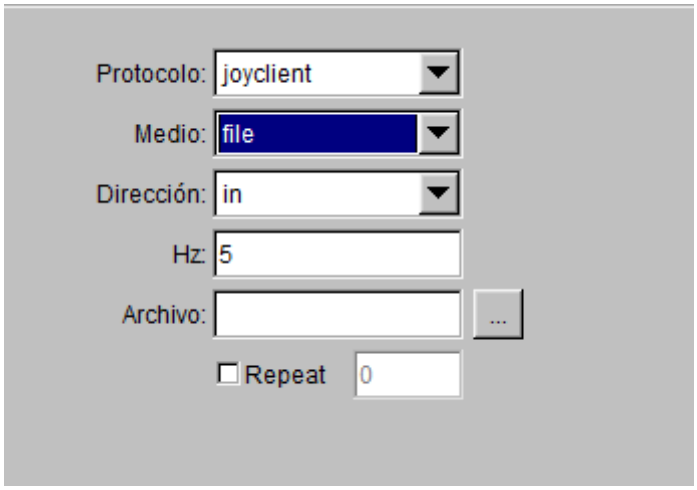


Figura 38. Opciones de Entrada/Salida de FG eligiendo "file" en Medio de comunicación.

Si el medio elegido es "serial", figura 39, la comunicación será mediante el puerto serie del ordenador. Donde "Puerto" es el nombre con el que nuestro sistema operativo identifica al puerto serie, normalmente COMX (siendo X un número del 1 hasta el número de puertos serie externos que tenga el ordenador, normalmente COM1 ó COM2) y "Ratio

de Baudios[11]" es la velocidad, en baudios por segundo, a la que vamos a hacer la comunicación.

Figura 39. Opciones de Entrada/Salida de FG eligiendo la opción "serial" en Medio.

Los protocolos que podemos usar son:

- **atcsim:** Es el simulador de control de tráfico aéreo atc610x, escrito en un archivo .xml. Si lo ponemos como entrada, el árbol de propiedades internas se actualizará a partir de dicho archivo y por tanto el avión solamente se controlará desde el archivo del atc. Y si lo ponemos como salida será el atc610x el que se actualice a partir de las propiedades de FG.
- **atlas:** Nos proporciona datos de un gps como altitud, latitud, velocidad, hora, etc.
- **garmin:** Protocolo que simula un gps Garmin.
- **AV400:** Protocolo que simula un gps Garmin 400 Series.
- **AV400Sim:** Genera el conjunto de comandos que un gps Garmin 400 Series emite o espera recibir.
- **generic:** Protocolo basado en un archivo .xml que puede ser de entrada, de salida o ambos, según esté configurado el archivo .xml. Dicho archivo puede ser creado fácilmente por los usuarios o bien elegir uno de los que vienen en la carpeta FlightGear/data/Protocol. En el archivo .xml escribimos los parámetros que deseamos ver o modificar del avión. Por ejemplo, si queremos obtener los datos *airspeed-kt, heading-deg y pitch-deg* debemos crear un archivo .xml como éste:

[11] Baudio: Es el conjunto de bits, también llamado símbolo, que determina el número de veces que cambia el estado de una señal. Al dividir el ratio de bits entre el ratio de baudios nos dará el número de bits que codifican un símbolo. $n = rb/rs$.

```
         <?xml version="1.0"?>
<PropertyList>
  <generic>

   <output><!-- al ser output, solamente se escribirán las
variables, para leerlas habría que poner un input-->
     <line_separator>newline</line_separator>
     <var_separator>newline</var_separator> <!--queremos que las
variables estén separadas por un salto de línea-->


 <chunk>
       <name>speed</name><!--nombre que le damos a la variable-->
       <format>V=%d</format>
       <node>/velocities/airspeed-kt</node><!--Nombre real y path--
>
     </chunk>

     <chunk>
       <name>heading (rad)</name>
       <format>H=%.6f</format>
       <type>float</type>
       <node>/orientation/heading-deg</node>
       <factor>0.0174532925199433</factor><!--para conversión a
radianes>
     </chunk>

     <chunk>
       <name>pitch angle (deg)</name>
       <format>P=%03.2f</format>
       <node>/orientation/pitch-deg</node>
     </chunk>
  </output>

</generic>
</PropertyList>
```

Al abrir el archivo donde le hayamos indicado al simulador que escriba estos datos, nos encontraremos esto:

```
V=16
H=3.590505
P=3.59
V=12
H=3.589020
```

- **joyclient:** Protocolo para conectar y controlar FG desde un joystick.
- **jsclient:** Conexión con un joystick remoto usando el protocolo de transporte UDP.
- **native**: Protocolo que lee o actualiza los parámetros de las variables de vuelo como aceleraciones, posiciones, velocidades, ángulos, etc. directamente de las funciones matemáticas escritas en el correspondiente FDM.
- **native-ctrls:** Es uno de los dos protocolos usados por nuestra aplicación. Se trata de un protocolo que manda o recibe una estructura de datos llamada *net_ctrls*, que contiene los parámetros de todos los controladores del avión

(alerón, acelerador, pedal...). Si FG recibe esta estructura, el protocolo native-ctrls acturalizará el árbol de propiedades internas. Si la transmite, el protocolo actualiza la estructura del árbol de propiedades justo antes de ser enviada. Nuestra aplicación la envía para poder controlar el avión y la recibe para poder modificar los parámetros. Estructura completa en Anexo I.

- **native-fdm:** Ídem que el protocolo anterior, salvo en que la estructura con la que opera es *net_fdm*, que contiene información de vuelo (altura, rumbo, velocidad, etc.). Éste es el otro protocolo que usamos, pero a diferencia del anterior sólo recibimos la estructura para poder leer los datos y a partir de la lectura modificamos los datos de la estructura *net_ctrls*. También leemos los datos *net_fdm* para imprimir los que nos interesan por pantalla, en la interfaz gráfica y en las gráficas. La estructura completa se puede ver en el anexo...
- **native-gui:** Protocolo que opera de igual forma que los dos anteriores sobre la estructura *net_gui* para simular una gui externa.
- **nmea[12]:** Protocolo que simula el protocolo de navegación marítima nmea. Un ejemplo para este protocolo y para los protocolos simuladores de un gps sería mandar los datos de estos protocolos al mismo puerto donde estemos ejecutando la aplicación atlas de FG, esto hará que en el mapa móvil de atlas se dibujen nuestras coordenadas.
- **opengc:** Interfaz de red para enviar datos a una pantalla a través de LAN. Este protocolo sirve para comunicarnos con un simulador de displays de cabinas escrito con OpenGC[13], es decir, envía los datos para que salgan en los displays de información desarrollados por OpenGC, como se muestra en la figura 40.

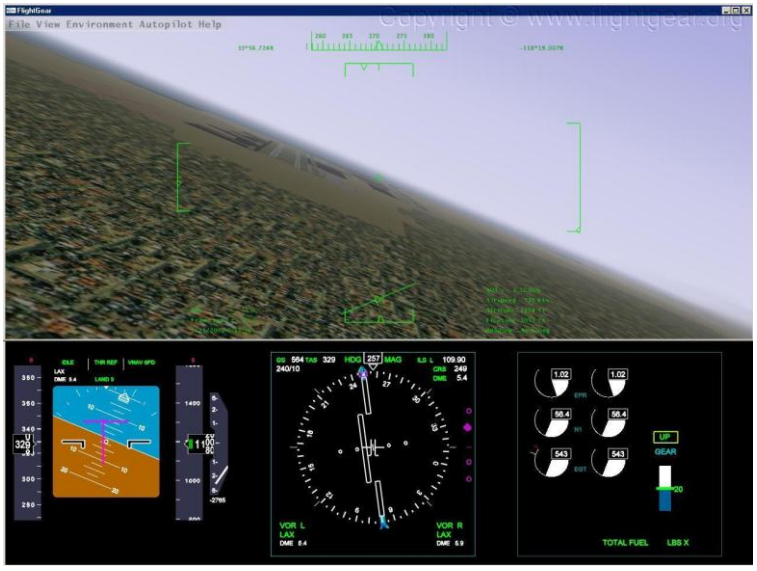


Figura 40. Protocolo OpenGC en FlighGear.

[12] National Marine Electronics Association (nmea) : es un protocolo por el cual los instrumentos marítimos y la mayoría de receptores gps pueden comunicarse.
[13] OpenGC: The Open Source Glass Cockpit Project, es una herramienta de software escrita en C++ para implementar displays de cabina de cristal de alta calidad.

- **props:** Crea un servidor telnet para poder acceder al árbol de propiedades internas de FG. Con las últimas actualizaciones este protocolo ya no se llama props sino telnet.
- **pve:** Protocolo usado para conexión con plataformas de movimiento. Se envían un byte codificado en ASCII del carácter 'p' y 3 bytes más que corresponderían a las variables *roll, pitch y heave* (aceleración vertical).
- **ray:** Este protocolo está desarrollado para que FlightGear envíe (y sólo envíe) algunos datos de vuelo a la silla de movimiento, como por ejemplo sobre la fuerza gravitatoria, los ángulos del avión, alabeo, cabeceo, aceleraciones, etc. Al estar el simulador FG sincronizado con la silla, ésta reproducirá los movimientos del avión en tiempo real, dando así un mayor realismo en la simulación del vuelo. La silla que vemos en la figura 41 ha sido desarrollada por Cobra Technologies, cuyo presidente es Ray WoodWorth, nombre que toma FlightGear para el protocolo. Figura 41.

Figura 41. Silla de movimiento sincronizada con un simulador de vuelo.

- **rul:** Mismo protocolo que el citado anteriormente pve.
- **telnet:** Mismo protocolo que props, ahora se llama telnet.

3.3. Estructura de los datagramas

En la comunicación entre FlightGear y la aplicación se intercambian dos tipos de datagramas: los que contienen la estructura de datos *net_fdm*, con la que opera el protocolo *native-fdm*. Y los que contienen la estructura *net_ctrls*, con la que opera el protocolo *native-ctrls*. Una de las primeras cosas realizadas en este proyecto fue la de capturar las

tramas que intercambian FG y la aplicación. El software utilizado para esto es una herramienta tipo sniffer llamada Wireshark, anteriormente conocido como Ethereal. Las tramas presentan este aspecto:

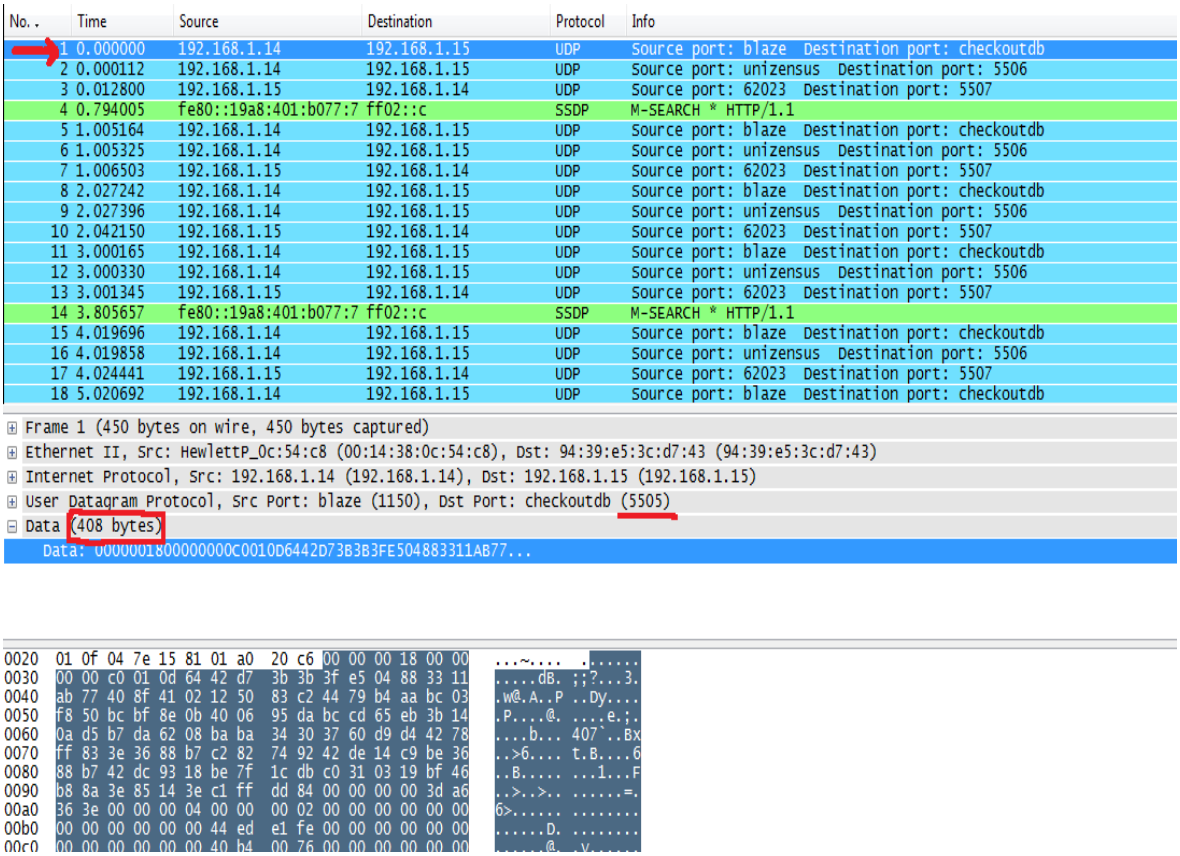


Figura 42. Captura de tramas, la primera contiene a native-fdm.

Antes de explicar el contenido de la figura 42 hay que decir que en el momento de hacer la captura de la figuras 42, 44 y 46 teníamos en funcionamiento una red LAN compuesta por dos ordenadores. El ordenador con la ip 192.168.1.14 ejecutaba el simulador de vuelo FG mientras que el pc con ip 192.168.1.15 ejecutaba nuestra aplicación. En las figuras 43, 45 y 47 se puede observar el intercambio de los datagramas entre simulador y aplicación, las dos primeras tramas son las del simulador que envía a la aplicación las estructuras *net_fdm* y *net_ctrls* por este orden, después nuestra aplicación envía al simulador la trama con la estructura *net_ctrls* ya modificada para que el avión la reciba y actualice sus controladores. Y así sucesivamente. Hay que decir que las tramas de color verde no forman parte del intercambio de datos. En la figura 42 concretamente se puede observar como seleccionamos arriba de la imagen la primera trama, que es la que contiene los datos fdm. Como podemos observar dichos datos tienen un tamaño de 408 bytes de los 450 bytes de la trama total habiendo sumado los 8 bytes de la cabecera del protocolo UDP, los 20 bytes de la cabecera del protocolo IP y finalmente los 14 bytes del protocolo Ethernet (Figura 43). Esta trama siempre tiene como puerto de destino el número 5505. La visualización de los datos capturados por Wireshark se hace en hexadecimal y ascii. Se pueden ver en la parte inferior de la imagen, subrayados en azul oscuro.

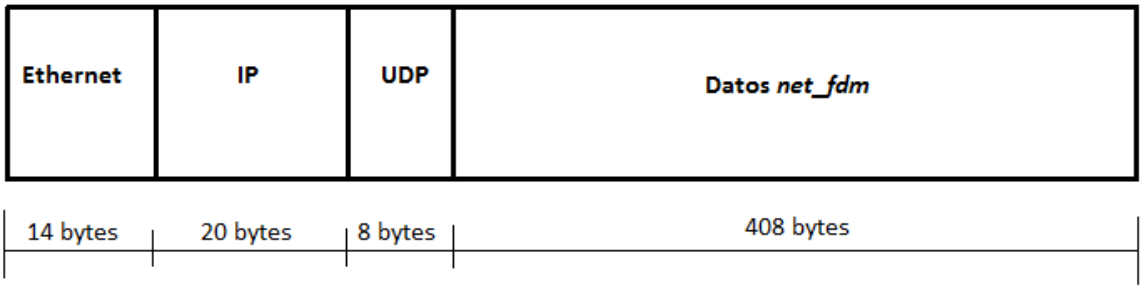


Figura 43. Datagrama net_fdm.

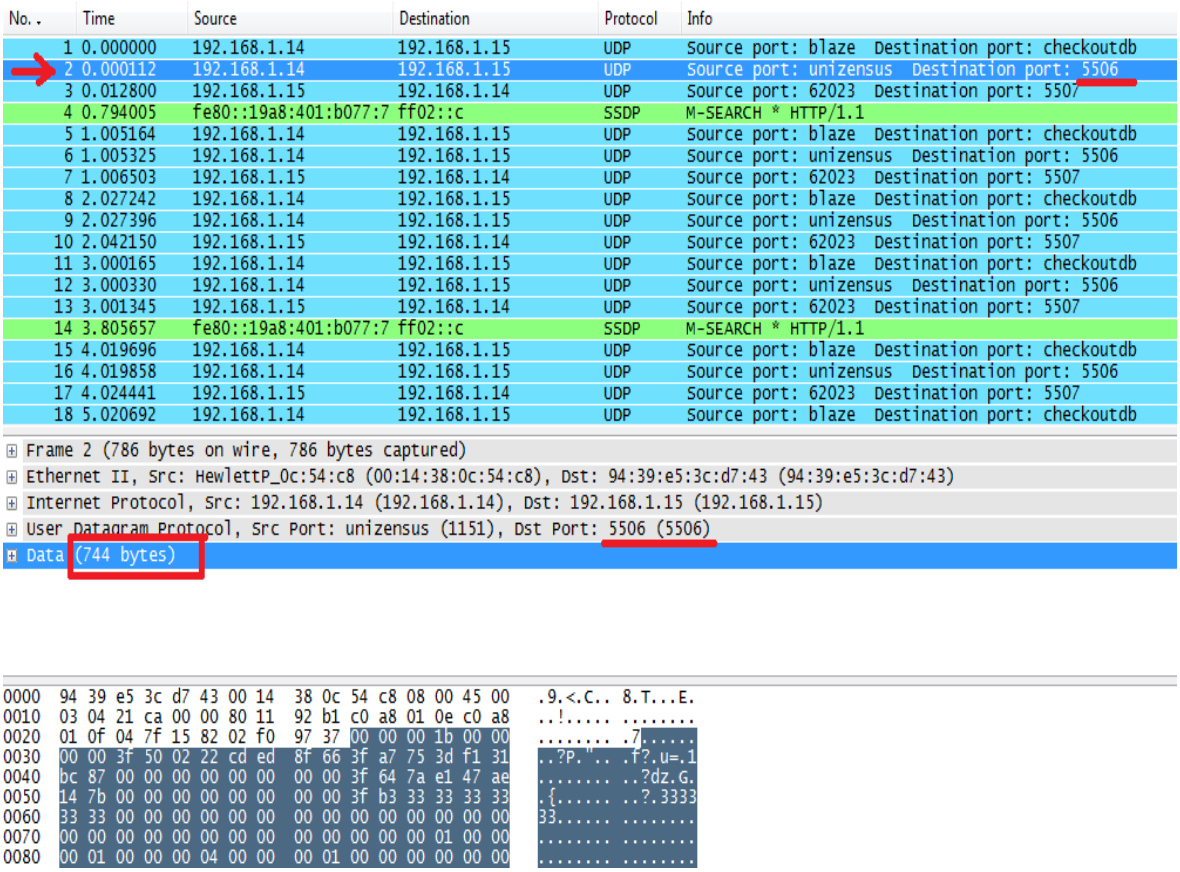


Figura 44. Captura de la trama native-ctrls.

Como se puede ver en las figuras 44 y 46 el tamaño de la tramas seleccionadas varían con respecto a las que contienen la estructura *net_fdm* de la figura 42, eso es porque la estructura *net_ctrls* es más grande y su tamaño es de 744 bytes. Al sumarle las cabeceras de los protocolos UDP, IP y Ethernet (figura 45) el tamaño total de la trama es de 786 bytes.

La diferencia entre la figura 44 y la 46 es el emisor y el receptor. La trama en sí no varía. Solamente varían los parámetros de las variables de la estructura, lógicamente, al ser modificados por nuestra aplicación.

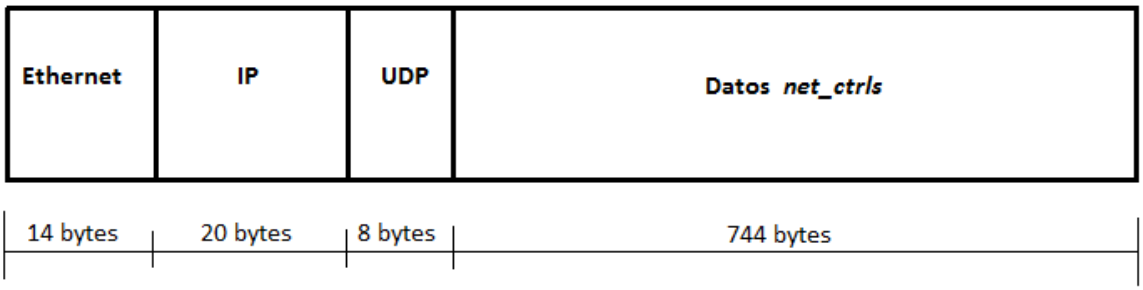


Figura 45. Datagrama que contiene a net_ctrls.

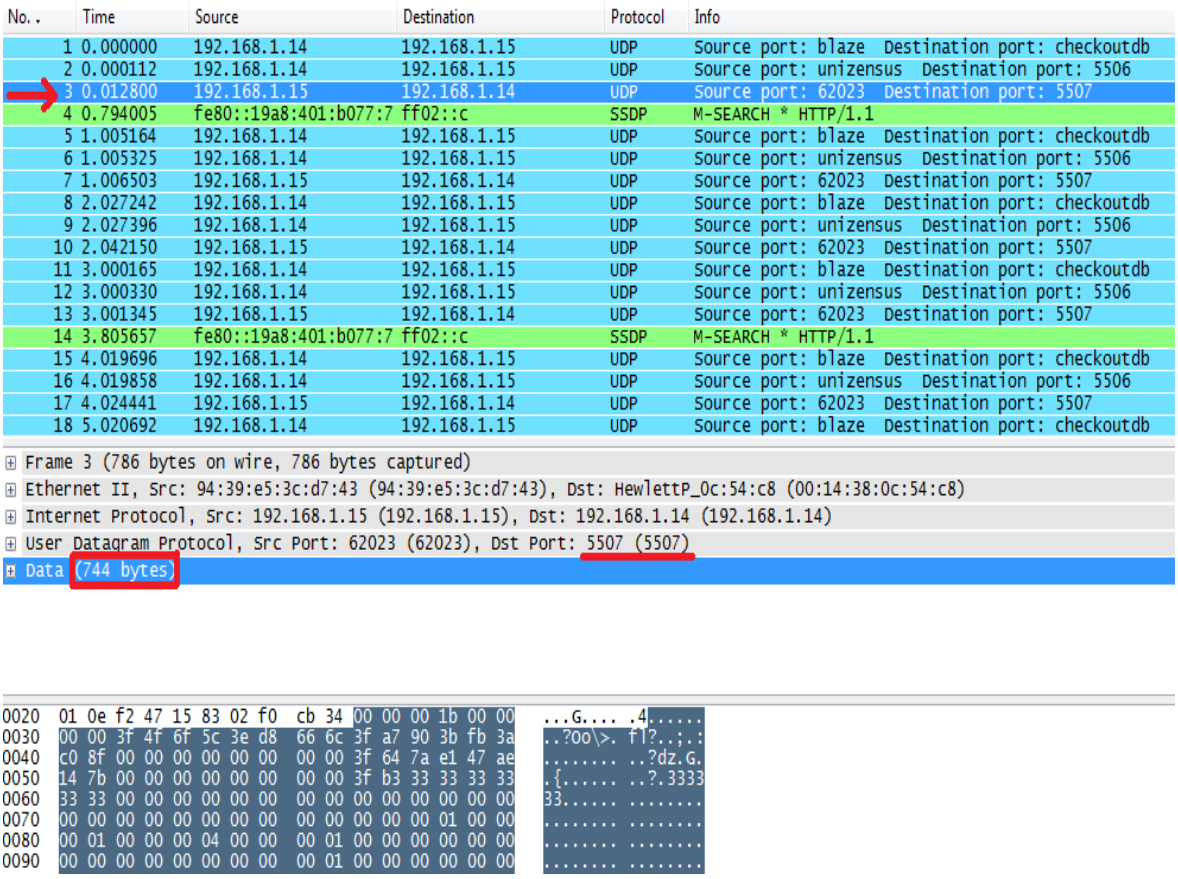


Figura 46. Captura de trama native-ctrls que envía la aplicación al simulador de vuelo.

3.4. Estructura de clases junto con diagramas UML

En este apartado vamos a mostrar el diagrama UML de nuestra aplicación, además de explicar las relaciones entre cada una de las clases también veremos sus funciones.

Comenzamos con el diagrama global de la aplicación, tal y como se puede ver en la figura 47.

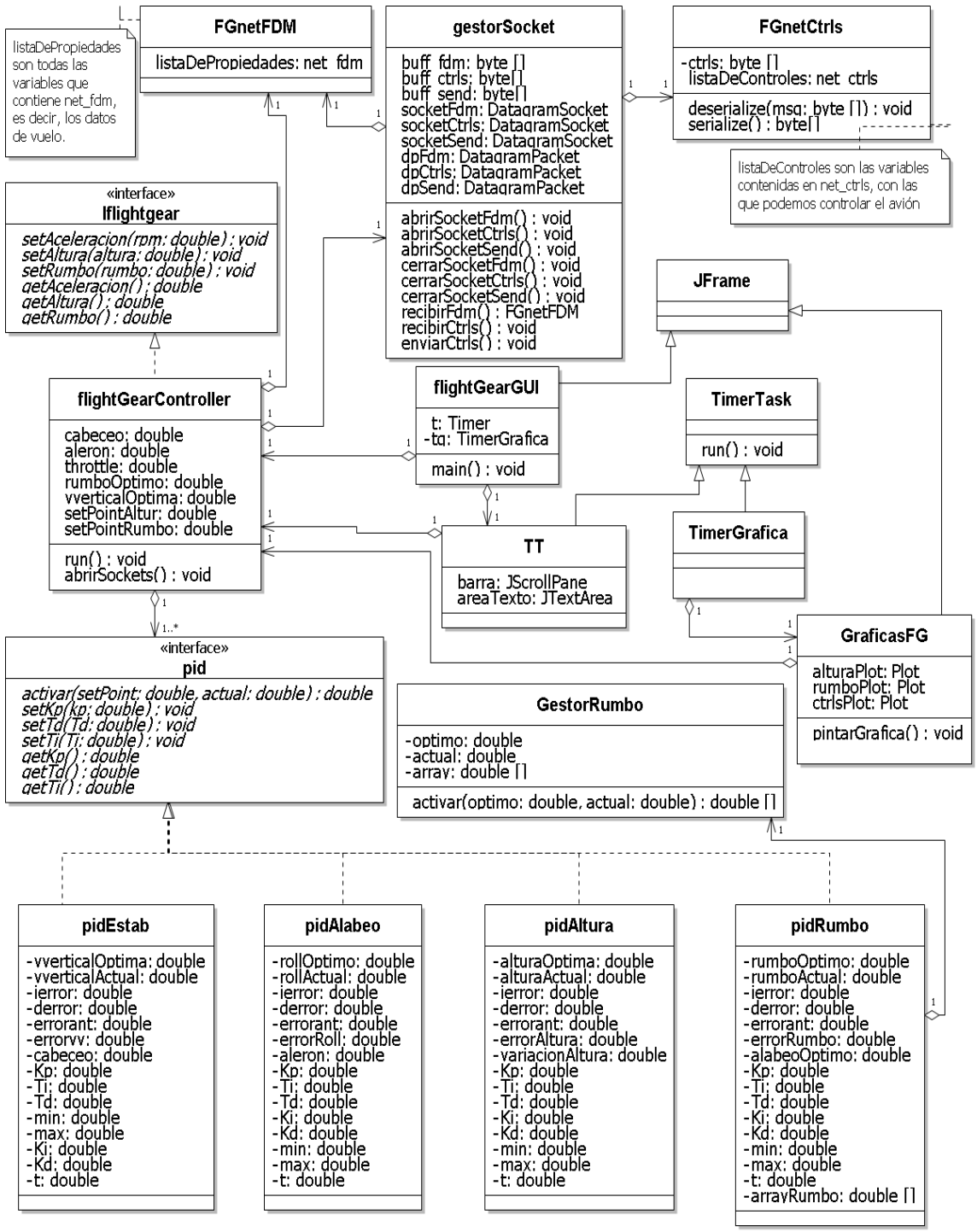


Figura 47. Diagrama UML de la aplicación.

Pero para explicarlo más claramente es mejor ir desmenuzando este diagrama en pequeños diagramas UML más sencillos. Para hacer esto hay que tener en cuenta que el factor común es la clase llamada *flightGearController* ya que es un hilo y por tanto implementa la función *run()*. En esta función lo más importante que se hace es la recepción de los datos de vuelo y de las variables de control, como hemos visto anteriormente.

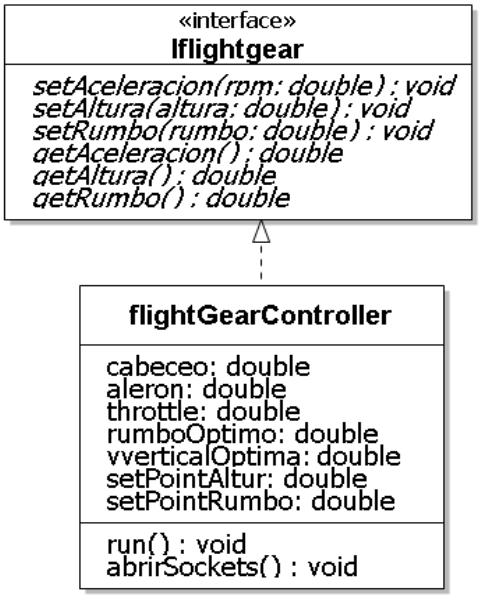


Figura 48. Hilo flightGearController y su interfaz Iflightgear.

Como se ve en la figura 48, ésta clase implementa una interfaz, *Iflightgear*, que es la que nos va hacer que podamos obtener y modificar los datos que nos interesan mediante las funciones *get/set*. Los datos que nos interesa obtener y modifcar son la altura, el rumbo y el controlador de la aceleración, *throttle*. Estas variables son las que nos aparece en la interfaz gráfica de usuario de la aplicación (GUI) , tanto para leerlas, como para introducir el valor deseado. Por eso son las elegidas.

En lo que a comunicación se refiere, la función *abrirSockets()* abre los dos sockets de entrada y el de salida, pero esta función delega en las funciones de apertura de sockets de la clase *gestorSocket.*

También hay que decir que *flightGearController,* [7], en su bucle infinito además de contener la recepción y el envío de las tramas también contiene muchas llamadas al sistema para imprimir numerosas variables. Estas variables han sido impresas por pantalla porque era de vital importancia conocer sus parámetros a medida que el proyecto avanzaba. Aunque al final en la GUI sólo se impriman unas pocas, conocer el resto de variables ha sido muy necesario. Como por ejemplo el caso de las variables de salida de los PID, o las distintas partes de los mismos, como por ejemplo la parte integral y derivativa, las cuales era necesario visualizar para comprobar que no crecieran infinitamente, poniendo en peligro la linealidad de los PID. O los parámetros de las variables actuadoras en el cabeceo y alabeo del avión, *elevator* y *aileron*, respectivamente. También el rumbo expresado en grados en las escalas [-180, 180] ha sido importante a la hora de implementar la clase *gestorRumbo*.

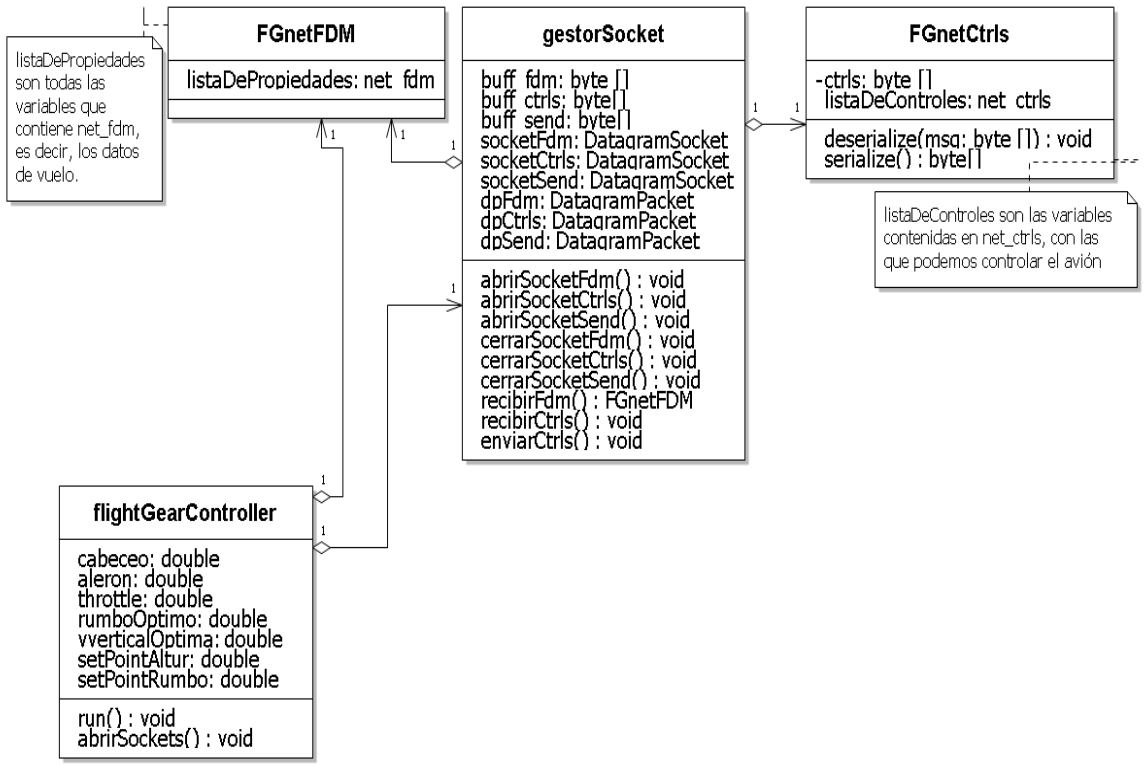


Figura 49. Diagrama UML con las clases encargadas de la entrada/salida de datos.

La figura 49 representa la parte encargada de la comunicación de la aplicación con FG. Básicamente ya lo hemos dicho todo sobre ella (ver el apartado *3.1.1. Implementación del protocolo de* comunicación

- **FGnetFDM:** Clase que es idéntica a la estructura *net_fdm*, con las mismas variables, mismo tipo y mismo orden. La asignación se hace en el constructor, que necesita como entrada un buffer de bytes, que contiene a *net_fdm*. Para hacer la asignación mediante el método *getInt()* o *getDouble()*, hay que obtener un objeto de la clase *ByteBuffer* mediante la función *wrap()* a la que le pasamos como argumento el array de bytes.
- **FGnetCtrls:** Clase igual a la estructura *net_ctrls*. Al igual que la clase anterior, las variables coinciden en nombre, tipo y orden para que haya coincidencia en el envío y la recepción. No tiene definido ningún constructor. Al recibir los datos se llama a la función *deserialize(byte [])* al que le pasamos el buffer de bytes recibido, cuyo contenido es *net_ctrls*. A continuación se realiza la asignación de la variables de instancia con los parámetros recibidos con la función *getInt()* o *getDouble()*. Antes de enviar los datos se llama a la función *serialize()*, la cual devuelve un buffer de bytes, con los parámetros de las variables ya modificadas mediante la función *putInt()* o *putDouble()* y listas para el envío.
- **gestorSocket:** Crea los tres sockets udp, con sus respectivos *DatagramPacket* asociados. Contiene las funciones encargadas de la gestión de los sockets, como la apertura, que en realidad es la creación del socket, el

cierre del mismo y la recepción y el envío de la información a través de dichos sockets.

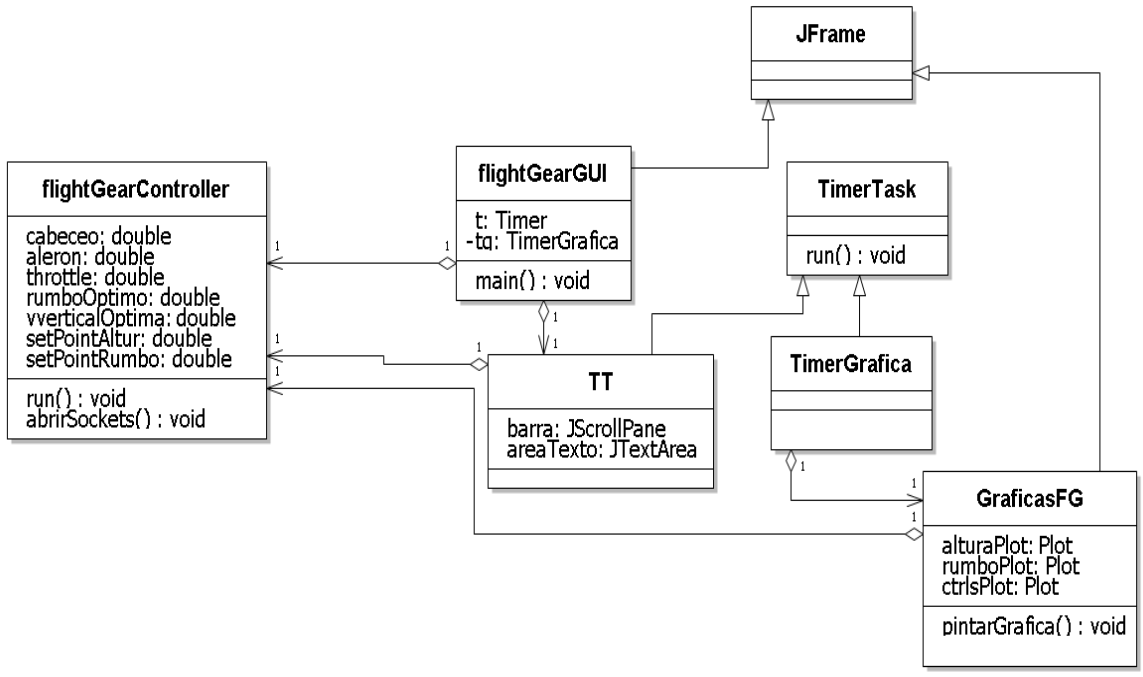


Figura 50. Diagrama UML de la interfaz gráfica de usuario.

En la figura 50 se ven las clases que se encargan de crear la interfaz gráfica de usuario y de dibujar las gráficas de la aplicación, así como de pasar los datos que piden para imprimirlos y dibujarlos cada cierto tiempo.

- **flightGearGUI:** Es la clase encargada de dibujar la GUI de la aplicación. Contiene el método *main()*. En su constructor se arranca el hilo *flightGearController* para que éste obtenga los datos, como hemos visto en el módulo de entrada/salida. Una vez arrancado le pasa los datos a la GUI para que los imprima en su *TextArea*. Contiene también un objeto de la clase *TT* y *Timer* configurados para que la impresión de datos sea cada dos segundos. Contiene también un objeto *TimerGráfica* para que las gráficas se actualicen cada segundo. Mirar [8], otra forma de implementar ActionListener.
- **TimerTask:** Clase abstracta para programar una tarea y repetirla cada cierto tiempo. Contiene un método *run()* para la programación de la tarea a realizar.
- **TT:** Hereda de *TimerTask* y contiene un hilo para el acceso a los datos, así como un *JTextArea* y una barra *JScrollPane*. Al constructor le pasamos una área de texto y el hilo. En su método *run()* se imprimen los datos y hacemos que *JScrollPane* se deslice automáticamente a medida que llegan los datos y se imprimen.

- **TimerGrafica:** Hereda también de *TimerTask.* En su constructor se crea la clase *GraficasFG* y en su método *run()* se llama al método *pintarGrafica()*.
- **GraficasFG:** Crea y pinta las gráficas de altura, rumbo, controles y velocidad. En el constructor le pasamos el thread *flightGearController* para que en el método *pintarGrafica()* obtenga los datos que queremos pintar.

La figura 51 nos muestra el diagrama con las clases encargadas de controlar el simulador.

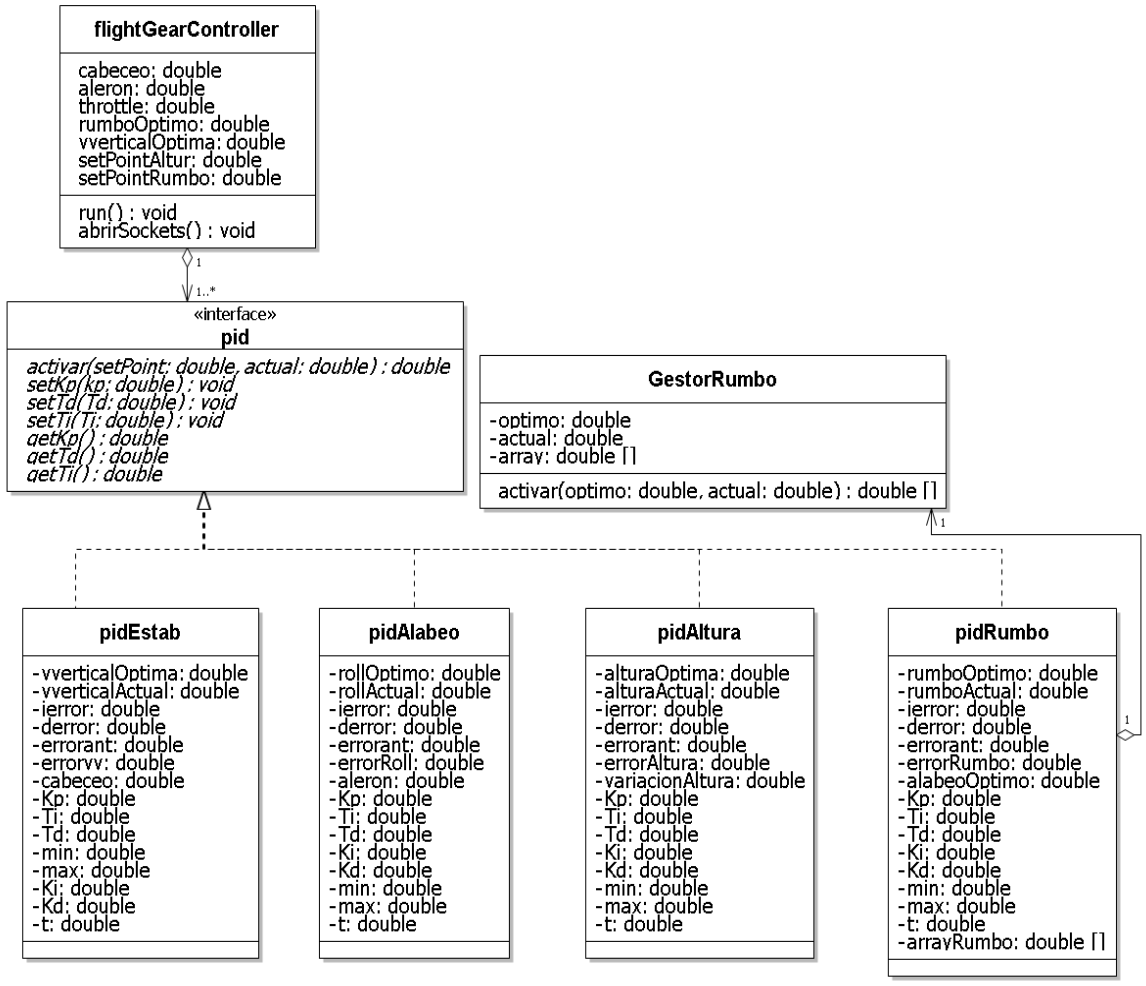


Figura 51. Diagrama UML con los PID encargados de controlar el avión.

Básicamente son 4 controladores o reguladores proporcional-integral-derivativo, también llamados PID que se encargan de controlar:

1. Cabeceo del avión para controlar la velocidad vertical y por tanto la maniobra de ascenso y descenso del avión.
2. Alabeo del avión para realizar la maniobra de giros y por tanto controlar el rumbo deseado.
3. Maniobra fundamental de vuelo recto y nivelado.

Las clases son:

- **pid:** esta clase es una interfaz para que la implementen los pid específicos. Contiene la función *activar(setPoint, actual)* cuyos parámetros de entrada son pasados por el hilo *flightGearController*, que como hemos visto anteriormente es el encargado de recibir los datos del simulador. También contiene los métodos *get()/set()* para modificar los parámetros del pid.
- **pidEstab:** es la clase que define el pid del cabeceo. Mediante ella se controla la velocidad vertical del avión y por tanto los ascensos y los descensos. Si deseamos que el avión no varíe su altura, para realizar el vuelo recto y nivelado, le pasaremos a la función *activar(setPoint, actual)* un cero como *setPoint*.
- **pidAlabeo:** se encarga del control del alabeo y por tanto de la maniobra de giro. Para realizar el vuelo recto y nivelado le pasaremos a la función *activar(setPoint, actual)* un cero en *setPoint*.
- **pidAltura:** este pid se usa en cascada con el pid de cabeceo, es decir, la salida de éste pid es la entrada del controlador del cabeceo. Así que en el *setPoint* le pasamos la altura que queremos alcanzar y la salida será el *setPoint* del *pidEstab*.
- **pidRumbo:** es igual a *pidAltura* pero para controlar el rumbo. La salida de este pid es el *setPoint* del *pidAlabeo*. En la función *activar()* de esta clase se llama a la función *activar()* de la clase *GestorRumbo* para optimizar y realizar los giros correctamente.
- **GestorRumbo:** Clase que hace eficiente la maniobra de giro. Impide que el avión quede dando vueltas indefinidamente, además asegura que el giro se realice por el camino más corto. Para realizar esto se llama a la función *activar()*, esta función transforma el rango del rumbo entre [0, 360] a [-180, 180] y viceversa, para evitar la discontinuidad o salto de cada rango.

3.5. Interfaz gráfica

La interfaz gráfica de la aplicación para controlar y supervisar al simulador FlightGear consta de una GUI (Interfaz Gráfica de Usuario) y de 4 gráficas que muestran las variables más importantes. Gracias a ellas es posible saber si el avión está bajo control y volando correctamente sin necesidad de verlo. Desde la GUI es desde donde hacemos que el avión efectúe las maniobras de ascenso, descenso, giros y vuelo recto y nivelado.

3.5.1. Implementación de la GUI

Para poder interactuar en tiempo real con el simulador de vuelo creamos una Interfaz Gráfica de Usuario (Graphical User Interface). Es una interfaz sencilla, que consta de un área de texto (*JTextArea*) en dónde se imprimen los datos más importantes para la supervisión del aparato, como son la altura, rumbo, velocidad, aceleración, rpm y velocidad vertical. Estos datos son los más importantes desde el punto de vista de las maniobras que se efectúan en el proyecto.

La GUI nos ofrece la posibilidad de modificar 3 parámetros escribiendo en las barritas blancas (*JTextField*) que están al lado de sus respectivas variables altura, rumbo y acelerador. En la figura 52 vemos como se han introducido los parámetros de las tres variables. Bastaría con darle al botón PLAY situado en la parte inferior para decirle a la aplicación que se hicieran efectivos los cambios.

Figura 52. Interfaz Gráfica de Usuario.

Para empezar a hablar de su implementación hay que decir que la clase flightGearGUI, clase que crea la GUI, hereda directamente de JFrame, que es una de las clases que nos proporciona JAVA para la creación de ventanas. Además de los antes mencionados *JTextField* y *JTextArea*, la interfaz contiene un *JButton*, botón de PLAY, al que se le añade un *actionListener* de esta forma:

```
play=new JButton("PLAY");
       play.addActionListener( new ActionListener() {
             public void actionPerformed( ActionEvent event ) {
       try{
              throttle=Double.parseDouble(aceleracion.getText());
              }//try
        catch(NumberFormatException nfe){
              throttle=fgc.gs.ctrls.throttle[0];
              }//catch
```

Con este código, decimos que al pulsar PLAY asignamos a la variable *throttle* de la estructura *net_ctrls* lo que hay escrito en el *JTextField* aceleración. Una vez se ha hecho lo mismo con las otras dos variables, se hacen efectivos los cambios así:

```
fgc.setAltura(altitud);
fgc.setRumbo(rumbod);
fgc.setAceleracion(throttle);
```

siendo *fgc* un objeto de la clase *flightGearController*.

La impresión de los datos en el *JTextArea* es sencilla. Su implementación no se hace en el constructor de la GUI, debido a que queremos que los datos se impriman en intervalos de tiempo. Para hacer esto, simulamos hacer otro hilo, creamos la clase *TT* que hereda de la clase *Timer*. A parte, a esta clase le pasamos el hilo fgc, para que obtenga los datos que se van a imprimir en el área de texto. Vemos su constructor:

```
public TT(JTextArea areaTexto,flightGearController fgc){
        this.areaTexto=areaTexto;
        areaTexto.setEnabled(false);
        barra=new JScrollPane(areaTexto);
        this.fgc=fgc;

    }//constructor
```

Vemos que los argumentos de entrada son el hilo fgc y el área de texto, pasados desde el constructor de la GUI:

```
tt=new TT(areaTexto,fgc);
```

En cambio, la barra (*JScrollPane*) se crea en el constructor de *TT*. En el método *run()* imprimimos en *JTextArea* los datos recibidos por fgc:

```
public void run(){

    try{

    rumbo=fgc.getRumbo();
    if(rumbo<0)rumbo=rumbo+360;
    areaTexto.append("\nAltura:"+fgc.getAltura()+"\nRumbo:"+rumbo+"\nAcelerac
    ion:"+fgc.getAceleracion()+"\nRPM:"+fgc.getRpm()+"\nVelocidad
    lineal:"+fgc.fdm.vcas+"\nVelocidad
    Vertical:"+fgc.fdm.climb_rate*0.3048+"(m/s)"+"\n");
    ...
```

Para pasarle la barra *JScrollPane* a la GUI hacemos:

```
p2.add(areaTexto);

p4.add(tt.barra);//pasamos el JScrollPane de TT

p2.add(p4);//pasamos el panel de la barra al del área texto

....
```

siendo p2 y p4 objetos de la clase *JPanel*.

Y por último, en el método principal del programa, es decir, en el *main()* que está en la clase de la GUI *flightGearGUI* creamos el objeto GUI y llamamos a *TT*:

```
public static void main(String[] args)
            ...

            flightGearGUI fgGUI=new flightGearGUI();
            fgGUI.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

            t.schedule(tt, fecha, 2000);//implementamos el método run() cada
            2000 milisegundos.
            ...
```

Por tanto tenemos una GUI capaz de imprimir cada 2 segundos la información que le pasa el hilo fgc, y a su vez ser capaz de recoger los datos que la pasamos por teclado y hacer que los cambios lleguen al simulador.

3.5.2. Implementación de las gráficas

La clase *GraficasFG* crea una ventana que alberga 4 gráficas que se van pintando y desplazando en el tiempo automáticamente. Así simplemente con un vistazo el usuario podrá obtener los datos más importantes (rumbo, altura, velocidad lineal y controles de cabeceo y alabeo) y decidir si todo está bajo control o no.

Para implementar la clase *GraficasFG* lo primero que hicimos fue importar el paquete:

```
import ptolemy.plot.Plot;
...
```

Habiendo añadido a nuestro proyecto los JAR (librerías) externos correspondientes.

Al igual que *flightGearGUI*, también hereda de *JFrame*, pero únicamente le pasamos el hilo fgc en el constructor. Es en el constructor donde se crean y configuran las gráficas: se les pone el nombre, rango de valores, opciones...

```
public GraficasFG(flightGearController fgc) {
            // Instantiate the two plots.
       super("GráficasFG");
        alturaPlot = new Plot();
        alturaPlot.setSize(250, 400);
        alturaPlot.setButtons(true);
        alturaPlot.setTitle("Altura");
        alturaPlot.setYRange(0, 4000);//antes de -4 a 4
         alturaPlot.setXLabel("tiempo");

       ...
```

así como la colocación de cada una de ellas en la ventana que las contiene:

```
       //Velocidad plot
       c.gridx = 1;
        c.gridy = 1;
        c.fill = GridBagConstraints.HORIZONTAL;
       gridbag.setConstraints(velocidadPlot, c);
        getContentPane().add(velocidadPlot);
```

```
... ...
```

En el fragmento de código se ve como se configura la posición de la gráfica en la ventana respecto a los ejes xy, en este caso al ser 1,1 la gráfica de la velocidad iría en la parte inferior derecha de la ventana. La línea de código *c.fill*....indica que se ajuste la gráfica a la ventana en horizontal, una vez hecho esto se le pasa la gráfica al panel contenedor junto con los parámetros de configuración.

En el método *pintar()* es dónde se imprimen los datos:

```
public void pintar(){
	boolean first = true;
	try{
	alturaPlot.addPoint(1, timeInSeconds = ( System.currentTimeMillis()-
	initTime ) / 1000 , fgc.getAltura(), first);
	...
```

El método *addPoint()* dibuja un punto del color indicado con un número, en este caso un 1. El segundo parámetro de la función es la posición en el eje X, como aquí vamos a dibujar en todas las gráficas los parámetros con respecto al tiempo, restamos el tiempo en que se creó la ventana con las gráficas y el tiempo actual a la hora de dibujar, como se dibuja cada segundo, tendremos que el eje X es un segundero de reloj. El tercer parámetro es la posición del punto respecto del eje Y, en nuestro caso es el valor del parámetro de la variable requerida, dependiendo de cada gráfica. Y por último, el cuarto parámetro es para pintar únicamente por puntos (si es false), o por rayas, es decir, uniendo los puntos que se van pintando.

Para que las gráficas se pinten cada segundo, creamos una clase llamada *TimerGrafica* que, al igual que *TT*, simula que es un hilo y es invocado cada cierto tiempo mediante su método *run()*. Por eso hereda de *TimerTask*:

```
public class TimerGrafica extends TimerTask{
	GraficasFG grafica;
	flightGearController fgc;
		.........
```

Requiere que le pasemos un hilo fgc, no para pintar las gráficas aquí, sino porque lo necesitamos para pasarselo al constructor de *GraficasFG*, para que una vez creado un objeto de esta clase invoque al método *pintar()* en el método *run()* de este timer:

```
public TimerGrafica(flightGearController fgc){
	this.fgc=fgc;
	grafica=new GraficasFG(fgc);
	}//constructor

public void run(){
	 grafica.pintar();
	}//run
```

Por tanto, en el constructor de la GUI escribimos:

```
t=new Timer();
...
tg=new TimerGrafica(fgc);
```

y en el método *main()* escribimos:

```
t.schedule(tg, fecha, 1000);
```

tendremos una ventana que contendrá 4 gráficas, que pintan cada segundo:

- El rumbo actual del avión y el que queremos alcanzar (setpoint).
- La altura actual del avión y la que queremos alcanzar.
- Los controles elevador y alerón, que son los encargados del cabeceo y alabeo del avión.
- La velocidad actual del avión y la velocidad óptima del avión.

Como se puede apreciar en la figura 53:

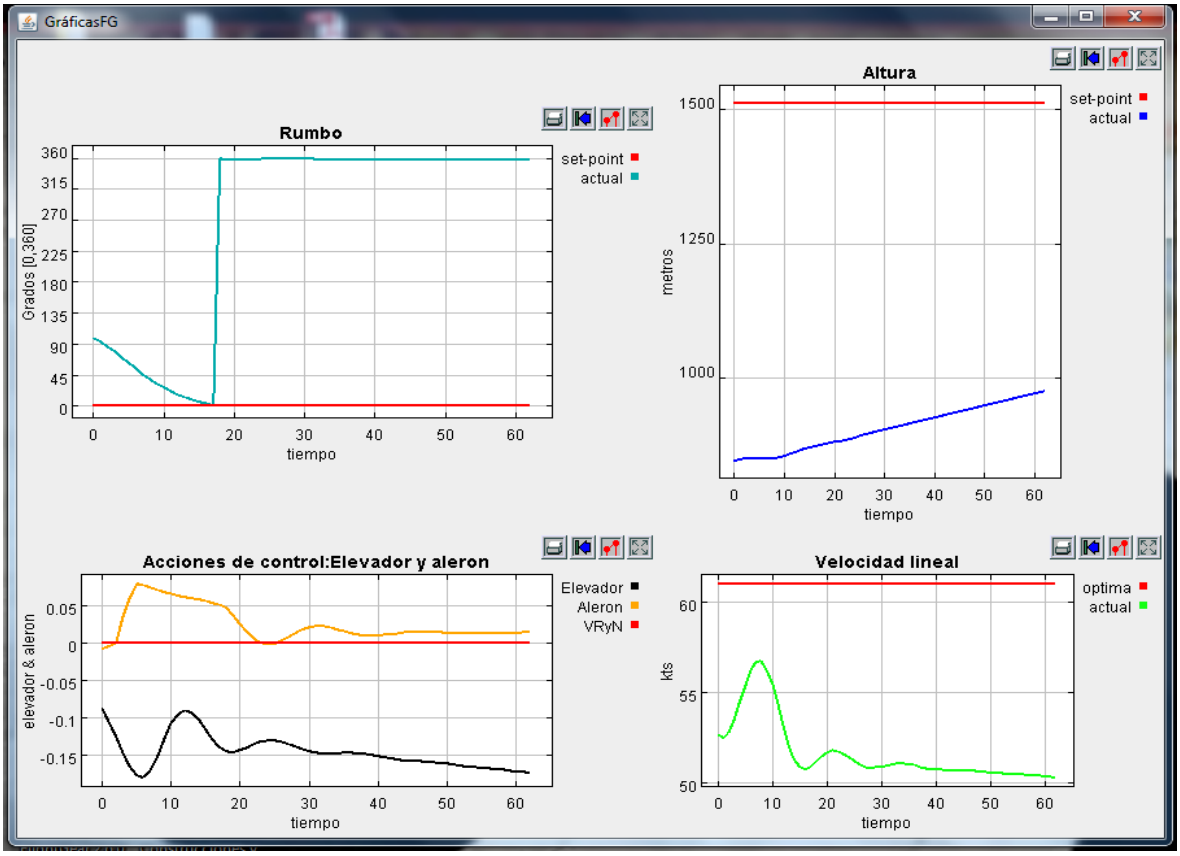


Figura 53. Gráficas FlightGear.

3.6. Diseño de los controladores

Nuestra aplicación se podría dividir en tres partes o funciones: comunicación con el simulador, visualización de los datos de interés mediante GUI y gráficas y por último la del control del avión.

En este apartado vamos a hablar de la forma en que nuestra aplicación controla al simulador de vuelo. Hemos optado por la estrategia de control mediante reguladores PID debido a su sencillez y eficacia, aunque el ajuste de los parámetros sea una labor un poco tediosa. A continuación detallamos su implementación y configuración.

3.6.1. Implementación de los reguladores PID

Nuestra aplicación consta de 4 reguladores o controladores PID que implementan un interfaz llamado *pid*. Este interfaz contiene los métodos *get()/set()* para la obtención y modificación de los parámetros de los distintos PID, además de una función *activar()* a la que le pasamos como parámetros de entrada el setpoint o consigna que queremos alcanzar y la variable que queremos cambiar con su valor actual.

El primer regulador que implementamos fue el que controla la variable *elevator*, que actúa sobre la velocidad vertical. Nuestro objetivo era conseguir que el avión mantuviera una altura constante, es decir, que ni ascendiera ni descendiera (velocidad vertical nula).

```
public class pidEstab implements pid {

	double vverticaloptima;
	double vverticalactual;
```

.....

En el constructor inicializamos todas las variables de vuelo, de controles y parámetros del PID:

```
public pidEstab(){

	vverticaloptima=0;
	ierror=0;//error integral
	errorvv=0;//error
	derror=0;//error derivada
	errorant=0;//error anterior derivada
	cabeceo=0;
	Ti=1/(25*0.5);
	Td=0.01;
	Kp=0.008;//0.008
	Ki=Kp*Ti;
	Kd=Kp*Td;
	t=1;
	min=-9;
	max=9;
```

hay que explicar que por ejemplo *Ti* aparece multiplicando, en vez de dividiendo, que es como debe de ser en la operación *Ki=Kp*Ti*. Esto está hecho así porque a la hora de ajustar el PID parece más fácil darle valores a un parámetro que multiplica que a uno que divide. En el código vemos que está en forma de fracción, pero en los demás PID no es así. También hay que decir que la variable *t=1* es el periodo de muestreo, que coincide con las veces que el simulador y la aplicación intercambian información en un segundo. Mirar pág.*44*.

Una vez definidos todos los errores y demás variables implementamos la función *activar():*

```
public double activar(double setPoint,double actual){

        ....
cabeceo=errorvv*Kp+ ierror*Ki*t + (derror*Kd)/t;//t=1

        return cabeceo;
}
```

esta es la línea de código que define al PID, como se ha visto en la página *41* con la ecuación discreta del PID en el apartado *Discretización de controladores PID.* Es también dentro de la función donde los errores y las variables se definen así:

```
vverticaloptima=setPoint;
vverticalactual=actual;
errorvv=vverticaloptima-vverticalactual;
ierror += errorvv;
derror = errorvv - errorant;
```

actualizándose la variable *errorant* justo antes de terminar la función de esta forma:

```
errorant = errorvv;
```

por último la función devuelve la variable *cabeceo* que es donde hemos almacenado el resultado de la ecuación y que por tanto será la variable de control que le enviemos al simulador de vuelo para corregir el cabeceo y para que el avión no oscile en altura.

Así que en el hilo fgc, dentro del bucle infinito invocamos a la función *activar()* y modificamos el valor de la variable que controla el cabeceo, dentro de la estructura *net_ctrls* de esta manera:

```
cabeceo=controlEstabilidad.activar(0,fdm.v_down);

.....

gs.ctrls.elevator=cabeceo;
gs.enviarCtrls();
```

la última línea se ejecuta cuando hayamos modificado todas las variables mediante los PID adecuados. Por ejemplo el PID que regula el alabeo sería exactamente igual que el del cabeceo. Ahora bien, no sería así ni para el regulador de la altura, ni para el del rumbo.

Los PID que corrigen altura y rumbo dependen directamente de las variables *elevator y roll*, es decir, de las mismas variables que los PID de alabeo y cabeceo que acabamos de ver. La forma de implementar estos reguladores es hacerlo en cascada, es decir, la salida de los reguladores altura y rumbo ha de ser el objetivo o setpoint de los reguladores del cabeceo y alabeo respectivamente. Por tanto, la implementación de las clases de estos PID son iguales que las de los anteriores, la única variación estaría en el

hilo fgc y en los parámetros de entrada de los PID de cabeceo y alabeo, los cuales ya no serían cero:

```
vverticaloptima=controlAltura.activar(setPointAltura, fdm.altitude);
cabeceo=controlEstabilidad.activar(-vverticaloptima,fdm.v_down);

rumboOptimo=controlRumbo.activar(setPointRumbo,Math.atan2(fdm.v_east,fdm.v_nort
h));
aleron=controlAlabeo.activar(rumboOptimo,fdm.phi);
```

A esto se le llama PID en cascada. Hay que decir que el signo negativo de la variable *vverticaloptima* es debido a que es justo el signo opuesto al de su controlador *cabeceo*. Así si el avión, por ejemplo está volando a una altura de 500 metros y queremos que baje hasta los 100 metros, obtendríamos una salida negativa del primer PID, el de la altura, debido a la resta que se hace para calcular el error (100-500), como un cabeceo negativo significa que el avión ascienda no queda más remedio que cambiar el signo del setpoint.

La maniobra de vuelo recto y nivelado que fue la que nos propusimos como primer objetivo en el control del avión se alcanza automáticamente sin tener que poner como setpoints de los PID de alabeo y cabeceo y sin tener que suprimir o anular los PID de la altura y del rumbo, ya que cuando éstos han alcanzado la altura y el rumbo que deseamos el error y la salida en estos PID prácticamente es nula, consiguiendo así la maniobra de vuelo recto y nivelado automáticamente.

3.6.2. Ajuste de los PID

Para realizar el ajuste de los PID no hemos aplicado ninguna fórmula ni nada que haga que el ajuste de los parámetros del regulador sea rápido. Simplemente el ajuste se hizo a mano y probando muchos y muchos valores para cada variable.

El algoritmo del ajuste manual era siempre el mismo:

Se comienza dando valores a *Kp*, cuando el avión comience a oscilar ajustamos *Kp* hasta el mínimo valor posible, pero siempre que siga existiendo oscilación. Después se ajusta el parámetro *Ti* hasta que el valor alcanzado sea prácticamente igual al setpoint. Para corregir la oscilación ajustamos el parámetro *Td* hasta que ésta desaparezca.

Por lo general, es una labor bastante pesada.

3.7. Problemas encontrados

El primer problema que surgió durante la ejecución del presente proyecto fue un **problema de concordancia de datos**. Los datos que recibíamos y enviábamos (*FGnetCtrls*) no eran iguales a los que esperábamos intercambiar. La primera sugerencia fue la de comprobar la versión del simulador y de su código fuente. Una vez vimos que eran correctas, se procedió al envío de un correo electrónico a uno de los creadores y desarrolladores de FlightGear, Curtis L. Olson, preguntándola sobre la posible causa del

problema. Su respuesta fue rápida y amable, pero desgraciadamente no nos sirvió de nada, ya que decía que se encontraba muy ocupado esos días y que le volviésemos a escribir pasadas unas semanas. Entonces, examinando las tramas recibidas, vimos que el único campo que recibíamos correctamente era el primero, la variable que contenía la versión del programa, una variable de tipo entero Integer. Por tanto, a continuación de este campo introducimos un padding o relleno de tipo entero también, en las funciones *deserialize* y *serialize*, solucionando así el problema.

El problema del rumbo óptimo y del giro por el camino más corto surgió cuando hacíamos la implementación del PID que regula el rumbo y el giro del avión. Inicialmente el rumbo lo transformamos de radianes a grados, en el rango de valores comprendido entre [0, 360]. El avión siempre giraba por el camino más largo, debido al signo de la resta al calcular el error en el PID. La solución inmediata, aunque nada eficaz, fue el cambio del valor absoluto del resultado de la diferencia. Esto solucionaba que el avión girara por el lado más largo, pero entonces surgió otro problema: debido a la discontinuidad en la escala de valores del rumbo, el avión quedaba dando vueltas indefinidamente. Es decir, si el avión al girar pasaba por el salto que se produce al pasar de 0 grados a 359 se quedaba atrapado en un bucle infinito que lo hacía girar constantemente. Por ejemplo:

El rumbo actual es de 5 grados y queremos alcanzar los 300, la resta 300-5 da un resultado de signo positivo, con lo cual el avión giraría a la derecha. Haciendo el cambio de signo, el avión giraría hacia la izquierda, pero al pasar la discontinuidad volvería a cambiar el signo de la resta: 300-359, resultado negativo, cambiando el valor absoluto, volveríamos a tener un valor positivo y el avión volvería a girar a la derecha, repitiendo de nuevo todo este proceso al volver a pasar por la discontinuidad.

La solución a este problema fue la de transformar el rango de valores del rumbo entre [0, 360] y [-180, 180], de tal forma que cuando el avión tenga que pasar por la discontinuidad 0/359 se cambiaría el rango por [-180, 180] y al revés. Para evitar la discontinuidad 179/-179 transformamos los valores del rumbo a [0, 360].

La implementación de la solución es una clase llamada *GestorRumbo* que es la encargada de hacer las transformaciones convenientes. Este es su constructor:

```
public GestorRumbo(){
	optimo=0;
	actual=0;
	array=new double [2];
}//constructor
```

Las variables de instancia son el rumbo óptimo o setpoint de rumbo, el rumbo actual y un array donde devolvemos los resultados a la clase *pidRumbo*.

La función que contiene es ésta:

```
public double[] activar(double rumboOptimo, double rumboActual);
```

A la cual se le pasan los valores del rumbo actual y del rumbo que queremos alcanzar, para que la función estudie si hay o no que hacer transformación.

```
if(optimo<180 && optimo!=0 && actual>180 && Math.abs(optimo-actual)>180){
	actual=Math.toRadians(actual-360);//cambiamos el rango de valores
	que pueden tomar los ángulos [-180,180]
	optimo=Math.toRadians(optimo);
System.out.println("Caso 1: Transformación rumbo actual [-180,180]");
	}//primer if

.....
	array[0]=optimo;
	array[1]=actual;
	return array;//los ángulos se devuelven en radianes
```

Este sería uno de los 4 casos posibles, todos los casos en el Anexo I.

Entonces en la clase *pidRumbo* tenemos un objeto de la clase *GestorRumbo* que es quien llama a su función desde la función *activar()* de *pidRumbo*:

```
arrayRumbo=gr.activar(setPoint, actual);
	rumboOptimo=arrayRumbo[0];
	rumboActual=arrayRumbo[1];
```

Una vez obtenidos los valores de los rumbos óptimo y actual se procede a realizar la ecuación característica del PID y a operar con normalidad.

Otro problema, mucho más sencillo que el anterior, fue la **excepción de puntero nulo** debido a que esperábamos recibir objetos o variables sin haber inicializado antes el hilo, por ejemplo en la clase de la GUI.

El problema de **autodeslizamineto de la barra *JScrollPane*** también tuvo una rápida y fácil solución. Vemos en el método *run()* de la clase TT:

```
tamañoAreaTexto = areaTexto.getSize();
p = new Point(0, tamañoAreaTexto.height);
barra.getViewport().setViewPosition(p);//para que se desplace JScrollPane
automaticamente hacia los ultimos datos recibidos
```

bastó con obtener el tamaño del *JTextArea* y pasarle la altura al nuevo punto. Así, si le pasamos el punto (0, altura de area texto) la barra llegará hasta dicha altura, sin desplazarse a lo ancho. Como se dice en [9].

Y por último, para solucionar que las **gráficas no avanzaran en el tiempo**, llamamos a la función *fillPlot()* en el método *pintar()* de la clase *GraficasFG*.

```
alturaPlot.fillPlot();//fillPlot() para que la gráfica se desplace
automáticamente por ejes XY
rumboPlot.fillPlot();
ctrlsPlot.fillPlot();
velocidadPlot.fillPlot();
```

Capítulo 4. Conclusiones y líneas futuras

4.1. Conclusiones y experiencia

En este proyecto se ha logrado comunicarnos con el simulador de vuelo FlightGear mediante mediante sockets UDP, el conocimiento de el contenido de las tramas recibidas y enviadas, gracias a que el proyecto entero de FlightGear es de código abierto, ha sido fundamental para la realización del proyecto.

También se ha conseguido interactuar en tiempo real con el simulador, gracias la Interfaz Gráfica de Usuario que hemos implementado, consiguiendo que el avión consiga hacer las maniobras de ascenso, descenso, giros y la maniobra de vuelo recto y nivelado.

El tema del control del avión ha sido muy divertido de hacer y sobre todo de ver los resultados que son capaces obtener los reguladores PID, aunque su ajuste no ha sido tan divertido.

El objetivo de la supervisión de los datos de interés del avión durante el vuelo se ha logrado gracias también a la GUI que imprime información de vuelo fundamental y gracias también a las 4 gráficas, que simplemente con mirarlas se sabe si hay algún conflicto durante el vuelo o no.

Mi experiencia durante la realización de este proyecto ha sido muy positiva en todos los aspectos. Antes de realizarlo no tenía ni idea de aviación, pero ahora que conozco un poco este campo me resulta bastante entretenido y divertido. También a la hora de realizar toda la programación en Java he aprendido muchas cosas que durante la enseñanza universitaria no se aprenden, porque no se profundiza bastante, algo que es normal. Y el conocimiento de la existencia de los reguladores PID me ha resultado muy interesante. Tengo que decir que desconocía por completo la existencia de estos reguladores, pues en la carrera que estoy cursando no cursamos ninguna asignatura que contenga materia alguna sobre este tema.

En general la realización de este proyecto me ha servido para adquirir muchos conocimientos, tanto del lenguaje de programación JAVA, como de reguladores PID y sobre todo del mundo de la aviación.

4.2. Trabajos futuros

Este proyecto no es una continuación de un trabajo ya empezado por otro alumno, en este proyecto partimos de cero. Es el primero de una línea de futuras ampliaciones y de nuevas ideas.

Lo bueno de trabajar con el simulador de vuelo FlightGear y el mundo de los UAV, es que ofrecen una gran variedad de ideas. Desde por ejemplo, seguir ampliando la gran variedad de maniobras que podemos realizar con un avión (aterrizaje, planeo...), hasta el vuelo en formación pasando por la realización automática de un plan de vuelo que finalice con el aterrizaje.

Respecto a los UAV (aviones no tripulados) se me ocurre la idea de uno controlado mediante la tecnología 3G que actualmente se usa en telefonía móvil,aunque más bien sería HSPA o incluso la tecnología LTE que aún está por llegar. El avión podría tener integrado un GPS que nos diera su posición en todo momento e incluso una cámara de vídeo para ver nosotros mediante un teléfono móvil, ordenador u otro dispositivo todo lo que capturase la cámara. Esto podría tener muchos usos, desde el salvamento, reconocimiento de terrenos, lucha contra incendios, espionaje, etc.

Otro proyecto que se me ocurre aprovechando el tirón que están teniendo los nuevos teléfonos móviles es el control de un simulador o maqueta mediante el giroscopio o sensor de giro que los teléfonos llevan integrado, además de una interfaz en el teléfono para dar más o menos potencia, etc. Esto sería como una especie de control remoto pero a poca distancia, por ejemplo por conexión bluetooth, wifi o wimax.

ANEXO I. CÓDIGO FUENTE

Clase FGnetCtrls

```
import java.nio.ByteBuffer;

public class FGnetCtrls {

	private byte[] ctrls = new byte[744];

	int FG_MAX_ENGINES = 4;
	int FG_MAX_WHEELS = 16;
	int FG_MAX_TANKS = 8;

	int version;                // increment when data values change

		// Aero controls
		double aileron;                // -1 ... 1
		double elevator;               // -1 ... 1
		double rudder;                 // -1 ... 1
		double aileron_trim;           // -1 ... 1
		double elevator_trim;          // -1 ... 1
		double rudder_trim;                  // -1 ... 1
		double flaps;                  //  0 ... 1
		double spoilers;
		double speedbrake;

		// Aero control faults
		int flaps_power;                 // true = power available
		int flap_motor_ok;

		// Engine controls
		int num_engines;             // number of valid engines
		int master_bat[]=new int[FG_MAX_ENGINES];
		int master_alt[]=new int[FG_MAX_ENGINES];
		int magnetos[]=new int[FG_MAX_ENGINES];
		int starter_power[]=new int[FG_MAX_ENGINES];
		double throttle[]=new double[FG_MAX_ENGINES];     //  0 ... 1
		double mixture[]=new double[FG_MAX_ENGINES];      //  0 ... 1
		double condition[]=new double[FG_MAX_ENGINES];    //  0 ... 1
		int fuel_pump_power[]=new int[FG_MAX_ENGINES];// true = on
		double prop_advance[]=new double[FG_MAX_ENGINES]; //  0 ... 1
		int feed_tank_to[]=new int[4];
		int reverse[]=new int[4];

		// Engine faults
		int engine_ok[]=new int[FG_MAX_ENGINES];
		int mag_left_ok[]=new int[FG_MAX_ENGINES];
		int mag_right_ok[]=new int[FG_MAX_ENGINES];
		int spark_plugs_ok[]=new int[FG_MAX_ENGINES];
		int oil_press_status[]=new int[FG_MAX_ENGINES];//
		int fuel_pump_ok[]=new int[FG_MAX_ENGINES];

		// Fuel management
		int num_tanks;                       // number of valid tanks
		int fuel_selector[]=new int[FG_MAX_TANKS];    // false = off
		int xfer_pump[]=new int[5];
		int cross_feed;                      // false = off, true = on
```

```
                // Brake controls
                double brake_left;
                double brake_right;
                double copilot_brake_left;
                double copilot_brake_right;
                double brake_parking;

                // Landing Gear
                int gear_handle; // true=gear handle down;
                // Switches
                int master_avionics;

                // nav and Comm
                double   comm_1;
                double   comm_2;
                double   nav_1;
                double   nav_2;

                // wind and turbulance
                double wind_speed_kt;
                double wind_dir_deg;
                double turbulence_norm;

                // temp and pressure
                double temp_c;
                double press_inhg;

                // other information about environment
                double hground;                 // ground elevation (meters)
                double magvar;             // local magnetic variation in degs.

                // hazards
                int icing;                      // icing status could me much
                                                     // more complex but I'm
                                                     // starting simple here.


                // simulation control
                int speedup;                    // integer speedup multiplier
                int freeze;                     // 0=normal
                                                // 0x01=master
                                                // 0x02=position
                                                // 0x04=fuel


public void deserialize(byte[] msg) {

        ByteBuffer data = ByteBuffer.wrap(msg);
                    //asignacion relativa de las variables.
        version=data.getInt();                  // increment when data values change
        data.getInt();//para que coincidan datos
        // Aero controls
        aileron=data.getDouble();               // -1 ... 1
        elevator=data.getDouble();                   // -1 ... 1
        rudder=data.getDouble();                // -1 ... 1
        aileron_trim=data.getDouble();          // -1 ... 1
        elevator_trim=data.getDouble();         // -1 ... 1
        rudder_trim=data.getDouble();                // -1 ... 1
        flaps=data.getDouble();                 //  0 ... 1
        spoilers=data.getDouble();
```

```
speedbrake=data.getDouble();

// Aero control faults
flaps_power=data.getInt();                  // true = power available
flap_motor_ok=data.getInt();

// Engine controls
num_engines=data.getInt();              // number of valid engines

      for(int i=0; i<FG_MAX_ENGINES; i++){
            master_bat[i]=data.getInt();
          }//for

          for(int i=0; i<FG_MAX_ENGINES; i++){
            master_alt[i]=data.getInt();
          }//for

          for(int i=0; i<FG_MAX_ENGINES; i++){
            magnetos[i]=data.getInt();
          }//for

          for(int i=0; i<FG_MAX_ENGINES; i++){
            starter_power[i]=data.getInt();
          }//for// true = starter power

          data.getInt();//linea para que coincidan datos

          for(int i=0; i<FG_MAX_ENGINES; i++){
            throttle[i]=data.getDouble();
          }//for//  0 ... 1

          for(int i=0; i<FG_MAX_ENGINES; i++){
            mixture[i]=data.getDouble();
          }//for //  0 ... 1

          for(int i=0; i<FG_MAX_ENGINES; i++){
            condition[i]=data.getDouble();
          }//for//  0 ... 1

          for(int i=0; i<FG_MAX_ENGINES; i++){
            fuel_pump_power[i]=data.getInt();
          }//for// true = on

          for(int i=0; i<FG_MAX_ENGINES; i++){
            prop_advance[i]=data.getDouble();
          }//for//  0 ... 1

          for(int i=0; i<4; i++){
            feed_tank_to[i]=data.getInt();
          }//for

          for(int i=0; i<4; i++){
            reverse[i]=data.getInt();
          }//for

          // Engine faults
          for(int i=0; i<FG_MAX_ENGINES; i++){
            engine_ok[i]=data.getInt();
          }//for
```

```
for(int i=0; i<FG_MAX_ENGINES; i++){
   mag_left_ok[i]=data.getInt();
}//for

for(int i=0; i<FG_MAX_ENGINES; i++){
   mag_right_ok[i]=data.getInt();
}//for

for(int i=0; i<FG_MAX_ENGINES; i++){
   spark_plugs_ok[i]=data.getInt();
}//for// false = fouled plugs

for(int i=0; i<FG_MAX_ENGINES; i++){
   oil_press_status[i]=data.getInt();
}//for// 0 = normal, 1 = low, 2 = full fail

for(int i=0; i<FG_MAX_ENGINES; i++){
   fuel_pump_ok[i]=data.getInt();
}//for

// Fuel management
num_tanks=data.getInt();  // number of valid tanks

for(int i=0; i<FG_MAX_TANKS; i++){
   fuel_selector[i]=data.getInt();
}//for// false = off, true = on

for(int i=0; i<5; i++){
   xfer_pump[i]=data.getInt();
}//for// specifies transfer from
// value tank to tank specified by
cross_feed=data.getInt();     // false = off, true = on

// Brake controls
 brake_left=data.getDouble();
 brake_right=data.getDouble();
 copilot_brake_left=data.getDouble();
 copilot_brake_right=data.getDouble();
 brake_parking=data.getDouble();

// Landing Gear
gear_handle=data.getInt(); // true=gear handle down;
// Switches
master_avionics=data.getInt();

// nav and Comm
comm_1=data.getDouble();
comm_2=data.getDouble();
nav_1=data.getDouble();
nav_2=data.getDouble();

// wind and turbulance
wind_speed_kt=data.getDouble();
wind_dir_deg=data.getDouble();
turbulence_norm=data.getDouble();

// temp and pressure
temp_c=data.getDouble();
```

```
                    press_inhg=data.getDouble();

                        // other information about environment
                    hground=data.getDouble();// ground elevation (meters)
                    magvar=data.getDouble();// local magnetic variation in degs.

                        // hazards
                    icing=data.getInt();

                        // simulation control
                    speedup=data.getInt();    // integer speedup multiplier
                    freeze=data.getInt();
                       //this.data=data;
                  }//deserialize

public byte[] serialize() {
              ByteBuffer  bb = ByteBuffer.wrap(ctrls);
              //bb.putDouble(alerions);
              bb.putInt(version);            // increment when data values change
              bb.putInt(0);//para que coincidan datos

            // Aero controls
             bb.putDouble(aileron);                    // -1 ... 1
             bb.putDouble(elevator);                   // -1 ... 1
             bb.putDouble(rudder);                     // -1 ... 1
             bb.putDouble(aileron_trim);         // -1 ... 1
             bb.putDouble(elevator_trim);              // -1 ... 1
             bb.putDouble(rudder_trim);                // -1 ... 1
             bb.putDouble(flaps);                //  0 ... 1
             bb.putDouble(spoilers);
             bb.putDouble(speedbrake);

            // Aero control faults
             bb.putInt(flaps_power);                 // true = power available
             bb.putInt(flap_motor_ok);

            // Engine controls
            bb.putInt(num_engines);             // number of valid engines

            for(int i=0; i<FG_MAX_ENGINES; i++){
              bb.putInt(master_bat[i]);
            }//for

            for(int i=0; i<FG_MAX_ENGINES; i++){
              bb.putInt(master_alt[i]);
            }//for

            for(int i=0; i<FG_MAX_ENGINES; i++){
              bb.putInt(magnetos[i]);
            }//for

            for(int i=0; i<FG_MAX_ENGINES; i++){
              bb.putInt(starter_power[i]);
            }//for// true = starter power

            bb.putInt(0);//para que coincidan datos

            for(int i=0; i<FG_MAX_ENGINES; i++){
              bb.putDouble(throttle[i]);
```

```
}//for//  0 ... 1

for(int i=0; i<FG_MAX_ENGINES; i++){
  bb.putDouble(mixture[i]);
}//for //  0 ... 1

for(int i=0; i<FG_MAX_ENGINES; i++){
  bb.putDouble(condition[i]);
}//for//  0 ... 1

for(int i=0; i<FG_MAX_ENGINES; i++){
  bb.putInt(fuel_pump_power[i]);
}//for// true = on

for(int i=0; i<FG_MAX_ENGINES; i++){
  bb.putDouble(prop_advance[i]);
}//for//  0 ... 1

for(int i=0; i<4; i++){
  bb.putInt(feed_tank_to[i]);
}//for

for(int i=0; i<4; i++){
  bb.putInt(reverse[i]);
}//for

// Engine faults
for(int i=0; i<FG_MAX_ENGINES; i++){
  bb.putInt(engine_ok[i]);
}//for

for(int i=0; i<FG_MAX_ENGINES; i++){
  bb.putInt(mag_left_ok[i]);
}//for

for(int i=0; i<FG_MAX_ENGINES; i++){
  bb.putInt(mag_right_ok[i]);
}//for

for(int i=0; i<FG_MAX_ENGINES; i++){
  bb.putInt(spark_plugs_ok[i]);
}//for// false = fouled plugs

for(int i=0; i<FG_MAX_ENGINES; i++){
  bb.putInt(oil_press_status[i]);
}//for// 0 = normal, 1 = low, 2 = full fail

for(int i=0; i<FG_MAX_ENGINES; i++){
  bb.putInt(fuel_pump_ok[i]);
}//for

// Fuel management
bb.putInt(num_tanks);  // number of valid tanks

for(int i=0; i<FG_MAX_TANKS; i++){
  bb.putInt(fuel_selector[i]);
}//for// false = off, true = on

for(int i=0; i<5; i++){
```

```
            bb.putInt(xfer_pump[i]);
        }//for// specifies transfer from array

        bb.putInt(cross_feed);            // false = off, true = on

        // Brake controls
        bb.putDouble(brake_left);
        bb.putDouble(brake_right);
        bb.putDouble(copilot_brake_left);
        bb.putDouble(copilot_brake_right);
        bb.putDouble(brake_parking);

        // Landing Gear
        bb.putInt(gear_handle);//true=gear handle down; false= gear handle up

        // Switches
        bb.putInt( master_avionics);

        // nav and Comm
        bb.putDouble(comm_1);
        bb.putDouble(comm_2);
        bb.putDouble(nav_1);
        bb.putDouble(nav_2);

        // wind and turbulance
        bb.putDouble(wind_speed_kt);
        bb.putDouble(wind_dir_deg);
        bb.putDouble(turbulence_norm);

        // temp and pressure
        bb.putDouble(temp_c);
        bb.putDouble(press_inhg);

        // other information about environment
        bb.putDouble(hground);                // ground elevation (meters)
        bb.putDouble(magvar);          // local magnetic variation in degs.

        // hazards
        bb.putInt(icing);
        // simulation control
         bb.putInt(speedup);                 // integer speedup multiplier
         bb.putInt(freeze);
          return bb.array();        //devolvemos un array de bytes (byte [])

    }
}//clase
```

Clase FGnetFDM

```
import java.nio.ByteBuffer;

public class FGnetFDM {

    int FG_MAX_ENGINES = 4;
    int FG_MAX_WHEELS = 3;
    int FG_MAX_TANKS = 4;
    int version;        // increment when data values change
    int padding;        // padding
```

```
// Positions
 double longitude;           // geodetic (radians)
 double latitude;            // geodetic (radians)
 double altitude;            // above sea level (meters)
 float agl;                  // above ground level (meters)
 float phi;                  // roll (radians)
 float theta;          // pitch (radians)
 float psi;                  // yaw or true heading (radians)
 float alpha;                // angle of attack (radians)
 float beta;                 // side slip angle (radians)

 // Velocities
 float phidot;         // roll rate (radians/sec)
 float thetadot;       // pitch rate (radians/sec)
 float psidot;         // yaw rate (radians/sec)
 float vcas;                   // calibrated airspeed
 float climb_rate;           // feet per second
 float v_north;              // north velocity in local/body frame, fps
 float v_east;               // east velocity in local/body frame, fps
 float v_down;             // down/vertical velocity in local/body frame, fps
 float v_wind_body_north;    // north velocity in local/body frame
                             // relative to local airmass, fps
 float v_wind_body_east;     // east velocity in local/body frame
                             // relative to local airmass, fps
 float v_wind_body_down;     // down/vertical velocity in local/body
                             // frame relative to local airmass, fps

 // Accelerations
 float A_X_pilot;            // X accel in body frame ft/sec^2
 float A_Y_pilot;            // Y accel in body frame ft/sec^2
 float A_Z_pilot;            // Z accel in body frame ft/sec^2

 // Stall
 float stall_warning;        // 0.0 - 1.0 indicating the amount of stall
 float slip_deg;       // slip ball deflection
 // Engine status
 int num_engines;           // Number of valid engines
 int eng_state[] = new int [FG_MAX_ENGINES];// Engine state (off, cranking
 float rpm[] = new float [FG_MAX_ENGINES];     // Engine RPM rev/min
 float fuel_flow[]= new float [FG_MAX_ENGINES]; // Fuel flow gallons/hr
 float fuel_px[]= new float [FG_MAX_ENGINES];   // Fuel pressure psi
 float egt[]=new float [FG_MAX_ENGINES];   // Exhuast gas temp deg F
 float cht[]=new float [FG_MAX_ENGINES];       // Cylinder head temp deg F
 float mp_osi[]=new float [FG_MAX_ENGINES];   // Manifold pressure
 float tit[]=new float [FG_MAX_ENGINES];       // Turbine Inlet Temperature
 float oil_temp[]=new float [FG_MAX_ENGINES];  // Oil temp deg F
 float oil_px[]=new float [FG_MAX_ENGINES];    // Oil pressure psi

 // Consumables
 int num_tanks;        // Max number of fuel tanks
 float fuel_quantity[] = new float[FG_MAX_TANKS];

 // Gear status
 int num_wheels;
 int wow[]=new int [FG_MAX_WHEELS];
 float gear_pos[] = new float [FG_MAX_WHEELS];
 float gear_steer[]=new float[FG_MAX_WHEELS];
 float gear_compression[]=new float[FG_MAX_WHEELS];
// Environmen
```

```
    int cur_time;           // current unix time
    int warp;                // offset in seconds to unix time
    float visibility;            // visibility in meters (for env. effects)

    // Control surface positions (normalized values)
    float elevator;
    float elevator_trim_tab;
    float left_flap;
    float right_flap;
    float left_aileron;
    float right_aileron;
    float rudder;
    float nose_wheel;
    float speedbrake;
    float spoilers

public FGnetFDM(byte [] msg) {

            ByteBuffer data = ByteBuffer.wrap(msg);
             version = data.getInt();
             padding = data.getInt();
             longitude = data.getDouble();
             latitude = data.getDouble();
             altitude = data.getDouble();
             agl = data.getFloat();   // above ground level (meters)
             phi = data.getFloat();                // roll (radians)
             theta = data.getFloat();        // pitch (radians)
             psi = data.getFloat();          // yaw or true heading (radians)
             alpha = data.getFloat();             // angle of attack (radians)
             beta = data.getFloat();              // side slip angle (radians)

             // Velocities
             phidot = data.getFloat();       // roll rate (radians/sec)
             thetadot = data.getFloat();           // pitch rate (radians/sec)
             psidot = data.getFloat();       // yaw rate (radians/sec)
             vcas = data.getFloat();                // calibrated airspeed
             climb_rate = data.getFloat();         // feet per second
             v_north = data.getFloat();//north velocity in local/body frame
            v_east = data.getFloat();// east velocity in local/body frame fps
             v_down = data.getFloat();// down/vertical velocity in local/body
             v_wind_body_north = data.getFloat
            v_wind_body_east = data.getFloat();
             v_wind_body_down = data.getFloat();

            // Accelerations
             A_X_pilot = data.getFloat();// X accel in body frame ft/sec^2
             A_Y_pilot = data.getFloat();// Y accel in body frame ft/sec^2
             A_Z_pilot = data.getFloat();// Z accel in body frame ft/sec^2

             // Stall
             stall_warning = data.getFloat();// 0.0 - 1.0 indicating the amount
             slip_deg = data.getFloat();           // slip ball deflection
             // Engine status
             num_engines = data.getInt();         // Number of valid engines
             // Engine state (off, cranking, running)
             for(int i=0; i<FG_MAX_ENGINES;i++){
               eng_state[i]=data.getInt();
             }//for
```

```
for(int i=0; i<FG_MAX_ENGINES;i++){
  rpm[i]=data.getFloat();
}//for// Engine RPM rev/min

for(int i=0; i<FG_MAX_ENGINES;i++){
  fuel_flow[i]=data.getFloat();
}//for// Fuel flow gallons/hr

for(int i=0; i<FG_MAX_ENGINES;i++){
  fuel_px[i]=data.getFloat();
}//for// Fuel pressure psi

for(int i=0; i<FG_MAX_ENGINES;i++){
  egt[i]=data.getFloat();
}//for// Exhuast gas temp deg F

for(int i=0; i<FG_MAX_ENGINES;i++){
  cht[i]=data.getFloat();
}//for// Cylinder head temp deg F

for(int i=0; i<FG_MAX_ENGINES;i++){
  mp_osi[i]=data.getFloat();
}//for// Manifold pressure

for(int i=0; i<FG_MAX_ENGINES;i++){
  tit[i]=data.getFloat();
}//for// Turbine Inlet Temperature

for(int i=0; i<FG_MAX_ENGINES;i++){
  oil_temp[i]=data.getFloat();
}//for// Oil temp deg F

for(int i=0; i<FG_MAX_ENGINES;i++){
  oil_px[i]=data.getFloat();
}//for// Oil pressure psi

// Consumables
num_tanks=data.getInt();        // Max number of fuel tanks

for(int i=0; i<FG_MAX_TANKS;i++){
  fuel_quantity[i]=data.getFloat();
}//for

// Gear status
int num_wheels=data.getInt();

for(int i=0; i<FG_MAX_WHEELS;i++){
  wow[i]=data.getInt();
}//for

for(int i=0; i<FG_MAX_WHEELS;i++){
 gear_pos[i]=data.getInt();
}//for

for(int i=0; i<FG_MAX_WHEELS;i++){
  gear_steer[i]=data.getFloat();
}//for

for(int i=0; i<FG_MAX_WHEELS;i++){
```

```
                gear_compression[i]=data.getFloat();
            }//for

            // Environment
            cur_time=data.getInt();   // current unix time
            warp=data.getInt();           // offset in seconds to unix time
            visibility=data.getFloat();// visibility in meters

            // Control surface positions (normalized values)
            elevator=data.getFloat();
            elevator_trim_tab=data.getFloat();
            left_flap=data.getFloat();
            right_flap=data.getFloat();
            left_aileron=data.getFloat();
            right_aileron=data.getFloat();
            rudder=data.getFloat();
            nose_wheel=data.getFloat();
            speedbrake=data.getFloat();
            spoilers=data.getFloat();

        }
    }
```

Clase gestorRumbo

```
public class GestorRumbo {
double optimo,actual;
double [] array;//contiene la transformacion de los rumbos y un indicador
                //para la clase pidRumbo.

    public GestorRumbo(){
            optimo=0;
            actual=0;
            array=new double [2];
    }//constructor
    public double[] activar(double rumboOptimo, double rumboActual){//función
que cambia el intervalo de valores en el que varían los angulos, de 0-360 a +-
180
            optimo=rumboOptimo;//pidRumbo nos lo pasa en grados [0,360]
            if(optimo==0)optimo=360;
            if(rumboActual<0)actual=Math.toDegrees(Math.PI*2+rumboActual);//
grados [0,360]
            else actual=Math.toDegrees(rumboActual);//si no es negativo no hace
falta el cambio: los valores positivos son iguales en +-180 con respecto a 0-
360
            System.out.println("Verificación Rumbo actual:"+rumboActual);

            if(optimo<180 && optimo!=0 && actual>180 && Math.abs(optimo-
actual)>180){
                    actual=Math.toRadians(actual-360);//cambiamos el rango
de valores que pueden tomar los ángulos [-180,180]
                    optimo=Math.toRadians(optimo);
                    System.out.println("Caso 1: Transformación rumbo
actual [-180,180]");
            }//primer if

            else if(optimo>180 && actual<180 && Math.abs(optimo-actual)>180){
                    optimo=Math.toRadians(optimo-360);//cambiamos el rango
de valores que pueden tomar los ángulos [-180,180]
```

```
                        actual=Math.toRadians(actual);
                        System.out.println("Caso 2: Transformación rumbo
optimo [-180,180]");
            }//primer elseif
            else if(optimo==0 || optimo==360){//caso que el setpoint sea 0
                  optimo=360;
                  optimo=Math.toRadians(optimo-360);
                  if (actual>180)actual=Math.toRadians(actual-360);
                  System.out.println("Caso 3: Transformación rumbo actual [-
180,180] setPoint=0");
            }//segundo else if
            else{//caso en el que no hay que transformar el rango de los
ángulos, x tanto estamos en [0,360]
                  optimo=Math.toRadians(optimo);
                  actual=Math.toRadians(actual);
                  System.out.println("Caso 4: No transformación");
            }//último else
            //falta devolver ángulos en caso de que no se cumplan estos if, por
tanto entre 0 y 6.28 rad. y signo=0;
            array[0]=optimo;
            array[1]=actual;
            return array;//los ángulos se devuelven en radianes.
      }//activar
}//clase
```

Clase Iflightgear

```
public interface Iflightgear {
      public void setAceleracion(double rmp);
      public void setAltura(double altura);
      public void setRumbo(double rumbo);

      public double getAceleracion();
      public double getAltura();
      public double getRumbo();

}
```

Clase flightGearController

```
import javax.swing.SwingUtilities;

public class flightGearController extends Thread implements Iflightgear {
      double cabeceo;
      double aleron;
      double rumboOptimo;
      double rudder;
      double throttle;
      double vverticaloptima;
      double vvertical;
      double setPointAltura=0;
      double setPointRumbo=0;
      int t;
      double tiempoInicio=0;
      gestorSocket gs;
      FGnetFDM fdm;

      pidAltura controlAltura;
      pidEstab controlEstabilidad;
```

```
	pidAlabeo controlAlabeo;
	pidRumbo controlRumbo;

	public flightGearController(){
		cabeceo=0;
		aleron=0;
		rumboOptimo=0;
		rudder=0;
		throttle=0.8;
		vverticaloptima=0;
		vvertical=0;
		t=0;
		controlAltura=new pidAltura();
		controlEstabilidad=new pidEstab();
		controlAlabeo=new pidAlabeo();
		controlRumbo=new pidRumbo();
		gs=new gestorSocket();
		tiempoInicio=System.currentTimeMillis();

	}//constructor

	public void run(){

		setPointAltura=1510;//error. no recibir en el constructor
		setPointRumbo=0;//

	while(true){
		t=(int)(System.currentTimeMillis()-tiempoInicio)/1000;
		fdm=gs.recibirFdm();
		//FGnetCtrls ctrls=new FGnetCtrls();
		gs.recibirCtrls();
		System.out.println("------------"+"tiempo:"+t+"--------------");
		System.out.println("bola_nivel:"+fdm.slip_deg);
		System.out.println("----------------------------------------");
		System.out.println("ACELERACION"+"\nrpm
	fdm:"+fdm.rpm[0]+"\nthrottle:"+gs.ctrls.throttle[0]);
		System.out.println("----------------------------------------");

	System.out.println("RUMBO"+"\nPidRumbo(rudder):"+rumboOptimo+"\nGuiñada:"
+fdm.psi+"\nrudder(controlador):"+fdm.rudder+"\nrumbo grados[-
pi,pi]:"+Math.toDegrees(Math.atan2(fdm.v_north,fdm.v_east))+"\nrumbo
radianes:"+Math.atan2(fdm.v_north,fdm.v_east)+"\nV_Norte:"+fdm.v_north+"\nV_Est
e:"+fdm.v_east);

	if(Math.atan2(fdm.v_east,fdm.v_north)<0)System.out.println("Rumbo grados
[0,360]:"+(Math.toDegrees(Math.atan2(fdm.v_east,fdm.v_north))+360));
			else System.out.println("Rumbo grados
[0,360]:"+Math.toDegrees(Math.atan2(fdm.v_east,fdm.v_north)));
			System.out.println("\nRumbo grados [-
180,180]:"+Math.toDegrees(Math.atan2(fdm.v_east,fdm.v_north)));
			System.out.println("----------------------------------------
-");
			System.out.println("CONTROL
ALABEO"+"\nalerón:"+gs.ctrls.aileron+"\nalerón(controlador):"+gs.ctrls.aileron+
"\nROLL(grados):"+fdm.phi*180/Math.PI+"\nphidot:"+fdm.phidot);
			System.out.println("----------------------------------------
-");
			//System.out.println();
```

```
		vverticaloptima=controlAltura.activar(setPointAltura, fdm.altitude);
		cabeceo=controlEstabilidad.activar(-vverticaloptima,fdm.v_down);

		rumboOptimo=controlRumbo.activar(setPointRumbo,Math.atan2(fdm.v_east,fdm.
v_north));
		aleron=controlAlabeo.activar(rumboOptimo,fdm.phi);
				vvertical+=fdm.v_down;

				//vverticalmedia=vvertical/t;
		System.out.println("ALTURAS Y CONTROL DE ALTURA"+"\nPID
ALTURA:"+vverticaloptima+"\naltitud tierra fdm:"+fdm.agl+"\nALTITUD MAR
fdm:"+fdm.altitude);

		System.out.println("\nt:"+t+"\nSetPointaltura:"+setPointAltura+"\nSetPoin
trumbo:"+setPointRumbo);
		System.out.println("----------------------------------------");
		System.out.println("CONTROL CABECEO Y VELOCIDADES");
System.out.println("PID
CABECEO(elevator):"+cabeceo+"\ncabeceo(controlador):"+fdm.elevator+"\nVVERTICAL
:"+fdm.v_down+"\nVELOCIDAD LINEAL:"+fdm.vcas);
		System.out.println("ACELERACION"+"\nrpm
fdm:"+fdm.rpm[0]+"\nthrottle:"+gs.ctrls.throttle[0]);
				/*System.out.println("----------------------------------------
---");
				System.out.println("Proporcional
estabilidad:"+controlEstabilidad.Kp*controlEstabilidad.errorvv);
				System.out.println("Integral
estabilidad:"+controlEstabilidad.Ki*controlEstabilidad.ierror);
				System.out.println("Derivada
estabilidad:"+controlEstabilidad.Kd*controlEstabilidad.derror);
				System.out.println("----------------------------------------
-");
				System.out.println("Proporcional
alabeo:"+controlAlabeo.Kp*controlAlabeo.errorvv);
				System.out.println("Integral
alabeo:"+controlAlabeo.Ki*controlAlabeo.ierror);
				System.out.println("Derivada
alabeo:"+controlAlabeo.Kd*controlAlabeo.derror);*/
		gs.ctrls.elevator=cabeceo;
		for(int i=0;i<fdm.num_engines;i++)gs.ctrls.throttle[i]=throttle;
				//gs.ctrls.aileron=aleron;
				//gs.ctrls.rudder=rudder;
				gs.ctrls.aileron=aleron;
				gs.enviarCtrls();
System.out.println("_____________________________________________________");

			}//while
		}
		public void setAceleracion(double aceleracion){throttle=aceleracion;}
		public void setAltura(double altura){setPointAltura=altura;}
		public void setRumbo(double rumbo){setPointRumbo=rumbo;}

		public double getAceleracion(){
			return gs.ctrls.throttle[0];
		}
		public double getAltura(){
			return fdm.altitude;
		}
		public double getRumbo(){
```

```
return Math.toDegrees(Math.atan2(fdm.v_east, fdm.v_north));
	}
	public double getRpm(){
		return fdm.rpm[0];
	}

	public void abrirSockets(){
		gs.abrirSocketCtrls();
		gs.abrirSocketFdm();
		gs.abrirSocketSend();
	}


}//clase
```

Clase flightGearGUI

```
import java.awt.*;
import javax.swing.*;
import javax.swing.event.AncestorListener;
import java.awt.event.*;
import java.lang.reflect.InvocationTargetException;
import java.util.Timer;
import java.util.Date;
public class flightGearGUI extends JFrame {
	private static final long serialVersionUID = 1L;//warning

	JTextField altura,rumbo,aceleracion;
	JLabel labelAltura,labelRumbo,labelAceleracion;
	JTextArea areaTexto;
	JButton play,stop ;
	JPanel p1, p2, p3,p4,p5,p6;
	static //JScrollPane barra;
	flightGearController fgc;
	static Timer t;
	static TT tt;
	static TimerGrafica tg;
	Color c;

	double altitud,rumbod,throttle;

	public flightGearGUI(){

		super("FlightGearGUI");

		Container container= getContentPane();

		fgc=new flightGearController();
		fgc.abrirSockets();
		altura= new JTextField();
		altura.setPreferredSize(new Dimension(40, 20));
		rumbo= new JTextField();
		rumbo.setPreferredSize(new Dimension(40, 20));
		aceleracion= new JTextField();
		aceleracion.setPreferredSize(new Dimension(40, 20));
		labelAltura=new JLabel("Altura");
		labelRumbo=new JLabel("Rumbo");
		labelAceleracion=new JLabel("Acelerador");
		areaTexto=new JTextArea(15,30);
```

```
        areaTexto.setEnabled(false);
        areaTexto.setFont(new Font("arial",Font.BOLD,14));
        //areaTexto.setBackground(c.BLACK);
        //areaTexto.setDisabledTextColor(c.BLACK);
        //barra=new JScrollPane(areaTexto);
        p1=new JPanel();
        p2=new JPanel();
        p3=new JPanel();
        p4=new JPanel();
        p5=new JPanel();
        p6=new JPanel();
        fgc.start();

        play=new JButton("PLAY");
        play.addActionListener( new ActionListener() {
public void actionPerformed( ActionEvent event ) {
           try{
              altitud=Double.parseDouble(altura.getText());}//try
           catch(NumberFormatException nfe){
              altitud=fgc.setPointAltura;
           }//catch
              if(altitud-fgc.getAltura()<0){
                    fgc.controlAlabeo.setKp(0.1);//
                    fgc.controlAlabeo.setTd(0.009);
                    fgc.controlAlabeo.setTi(80);
              }//if altura
           try{
              rumbod=Double.parseDouble(rumbo.getText());}//try
           catch(NumberFormatException nfe){
              rumbod=fgc.setPointRumbo;
           }//catch
              if(rumbod-fgc.getRumbo()!=0){
                    fgc.controlAlabeo.setKp(0.1);// valores alabeo2
                    fgc.controlAlabeo.setTd(0.009);
                    fgc.controlAlabeo.setTi(80);
              }//if rumbo
           try{
              throttle=Double.parseDouble(aceleracion.getText());}//try
            catch(NumberFormatException nfe){
                           throttle=fgc.gs.ctrls.throttle[0];
           }//catch
              fgc.setAltura(altitud);
              fgc.setRumbo(rumbod);
              fgc.setAceleracion(throttle);



      }

        p1.add(labelAltura);
        p1.add(altura);
        p2.add(labelRumbo);
        p2.add(rumbo);
        p2.add(labelAceleracion);
        p2.add(aceleracion);

        p2.add(areaTexto);

        p3.add(play);
```

```
            t=new Timer();
            tt=new TT(areaTexto,fgc);
            p4.add(tt.barra);

            container.setLayout (new BorderLayout());
            container.add(p1, BorderLayout.NORTH);
            p2.add(p4);
            container.add(p2, BorderLayout.CENTER);
            //container.add(p5, BorderLayout.CENTER);

            container.add(p3, BorderLayout.SOUTH);
            //p3.add(p6);

            setSize(460, 400);//anchoxalto
            setVisible(true);

      tg=new TimerGrafica(fgc);

      }//constructor

      public static void main(String[] args) {
            Date fecha=new Date();

            flightGearGUI fgGUI=new flightGearGUI();
            fgGUI.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

            t.schedule(tt, fecha, 2000);
            t.schedule(tg, fecha, 1000);

      }//main

}//clase
```

Clase TT

```
import java.awt.Color;
import java.awt.Dimension;
import java.awt.Point;
import java.util.TimerTask;
import javax.swing.JPanel;
import javax.swing.JScrollPane;
import javax.swing.JTextArea;

public class TT extends TimerTask{
      JTextArea areaTexto;
      JScrollPane barra;
      flightGearController fgc;
      Dimension tamañoAreaTexto;
      Point p;
      double rumbo;

      public TT(JTextArea areaTexto,flightGearController fgc){
            this.areaTexto=areaTexto;
            areaTexto.setEnabled(false);
            barra=new JScrollPane(areaTexto);
            this.fgc=fgc;

      }//constructor
```

```
public void run(){

	try{

	areaTexto.append("=============trama:"+fgc.t+"=============");
	rumbo=fgc.getRumbo();
	if(rumbo<0)rumbo=rumbo+360;
	areaTexto.append("\nAltura:"+fgc.getAltura()+"\nRumbo:"+rumbo+"\nAcelerac
ion:"+fgc.getAceleracion()+"\nRPM:"+fgc.getRpm()+"\nVelocidad
lineal:"+fgc.fdm.vcas+"\nVelocidad
Vertical:"+fgc.fdm.climb_rate*0.3048+"(m/s)"+"\n");
	areaTexto.append("================================\n");
	tamañoAreaTexto = areaTexto.getSize();
	p = new Point(0, tamañoAreaTexto.height);
	barra.getViewport().setViewPosition(p);//para que se desplace JScrollPane
automaticamente hacia los ultimos datos recibidos

	}//try
	catch(NullPointerException npe){
		areaTexto.append("\nesperando recibir datos para imprimir...\n");

	}//catch

}//run
}
```

Clase GraficasFG

```
import java.awt.Color;
import java.awt.Container;
import java.awt.Dimension;
import java.awt.GridBagConstraints;
import java.awt.GridBagLayout;
import java.awt.Point;
import java.awt.Rectangle;
import java.io.File;
import java.io.FileInputStream;
import java.io.FileNotFoundException;
import java.io.IOException;
import javax.swing.JFrame;
import javax.swing.SwingUtilities;
import ptolemy.plot.Plot;

public class GraficasFG extends JFrame {

Plot alturaPlot,rumboPlot,ctrlsPlot,velocidadPlot;
	flightGearController fgc;
	long initTime ;
	long timeInSeconds;
	double rumbo;
	GridBagLayout gridbag;
	GridBagConstraints c ;
	double [] rango;
	Rectangle rectangulo;

	public GraficasFG(flightGearController fgc) {
	  // Instantiate the two plots.
	super("GráficasFG");
```

```
 alturaPlot = new Plot();
 rumboPlot = new Plot();
 ctrlsPlot = new Plot();
 velocidadPlot=new Plot();
 this.fgc=fgc;
 initTime= System.currentTimeMillis();
 rumbo=0;


// Set the size of the toplevel window.
setSize(900, 700);//ventana graficasfg (anchoxlargo)

// Create the left plot by calling methods.
// Note that most of these methods should be called in
// the event thread, see the Plot.java class comment.
// In this case, main() is invoking this constructor in
// the event thread.
alturaPlot.setSize(250, 400);
alturaPlot.setButtons(true);
alturaPlot.setTitle("Altura");
alturaPlot.setYRange(0, 4000);//antes de -4 a 4
//alturaPlot.setXRange(0, 600);//en sg??
alturaPlot.setXLabel("tiempo");
alturaPlot.setYLabel("metros");
alturaPlot.addYTick("0", 0);//antes -pi
alturaPlot.addYTick("250", 250);
alturaPlot.addYTick("500", 500);//antes -pi/2
alturaPlot.addYTick("750", 750);
alturaPlot.addYTick("1000", 1000);//0
alturaPlot.addYTick("1250", 1250);
alturaPlot.addYTick("1500", 1500);//pi/2
alturaPlot.addYTick("1750", 1750);
alturaPlot.addYTick("2000", 2000);//antes pi
alturaPlot.addYTick("2250", 2250);
alturaPlot.addYTick("2500", 2500);//antes -pi/2
alturaPlot.addYTick("2750", 2750);
alturaPlot.addYTick("3000", 3000);//0
alturaPlot.addYTick("3250", 3250);
alturaPlot.addYTick("3500", 3500);//pi/2
alturaPlot.addYTick("3750", 3750);
alturaPlot.addYTick("4000", 4000);//antes pi
alturaPlot.setMarksStyle("none");
alturaPlot.setImpulses(true);
alturaPlot.addLegend(0,"set-point");
alturaPlot.addLegend(1,"actual");
alturaPlot.setImpulses(false);       //para quitar los Stems

//grafica rumbo
rumboPlot.setButtons(true);
rumboPlot.setSize(300, 300);
rumboPlot.setTitle("Rumbo");
rumboPlot.setYRange(0, 360);//antes de -4 a 4
rumboPlot.setXLabel("tiempo");
rumboPlot.setYLabel("Grados [0,360]");
rumboPlot.addYTick("0", 0);//antes -pi
rumboPlot.addYTick("45", 45);
rumboPlot.addYTick("90", 90);//antes -pi/2
rumboPlot.addYTick("135", 135);
```

```
        rumboPlot.addYTick("180", 180);//0
        rumboPlot.addYTick("225", 225);
        rumboPlot.addYTick("270", 270);//pi/2
        rumboPlot.addYTick("315", 315);
        rumboPlot.addYTick("360", 360);//antes pi
        rumboPlot.setMarksStyle("none");
        rumboPlot.setImpulses(true);
        rumboPlot.addLegend(0,"set-point");
        rumboPlot.addLegend(2,"actual");
        rumboPlot.setImpulses(false);

        //grafica acciones de control
        ctrlsPlot.setButtons(true);
        ctrlsPlot.setSize(350, 250);
ctrlsPlot.setTitle("Acciones de control:Elevador y aleron");
        ctrlsPlot.setYRange(-1, 1);//antes de -4 a 4
        ctrlsPlot.setXLabel("tiempo");
        ctrlsPlot.setYLabel("elevador & aleron");
        ctrlsPlot.addYTick("-1", -1);//antes -pi
        ctrlsPlot.addYTick("-0.75", -0.75);
        ctrlsPlot.addYTick("-0.5", -0.5);//antes -pi/2
        ctrlsPlot.addYTick("-0.25", -0.25);
        ctrlsPlot.addYTick("-0.15", -0.15);
        ctrlsPlot.addYTick("-0.1", -0.1);
        ctrlsPlot.addYTick("-0.05", -0.05);
        ctrlsPlot.addYTick("0", 0);//0
        ctrlsPlot.addYTick("0.15", 0.15);
        ctrlsPlot.addYTick("0.1", 0.1);
        ctrlsPlot.addYTick("0.05", 0.05);
        ctrlsPlot.addYTick("0.25", 0.25);
        ctrlsPlot.addYTick("0.5", 0.5);//pi/2
        ctrlsPlot.addYTick("0.75", 0.75);
        ctrlsPlot.addYTick("1", 1);//antes pi
        ctrlsPlot.setMarksStyle("none");
        ctrlsPlot.setImpulses(true);
        ctrlsPlot.addLegend(3,"Elevador");
        ctrlsPlot.addLegend(4,"Aleron");
        ctrlsPlot.addLegend(0, "VRyN");
        ctrlsPlot.setImpulses(false);

        velocidadPlot.setSize(250, 250);
        velocidadPlot.setButtons(true);
        velocidadPlot.setTitle("Velocidad lineal");
        velocidadPlot.setYRange(0, 80);//antes de -4 a 4
        velocidadPlot.setXLabel("tiempo");
        velocidadPlot.setYLabel("kts");
        velocidadPlot.addYTick("0", 0);//antes -pi
        velocidadPlot.addYTick("15", 15);
        velocidadPlot.addYTick("45", 45);//antes -pi/2
        velocidadPlot.addYTick("50", 50);
        velocidadPlot.addYTick("55", 55);
        velocidadPlot.addYTick("60", 60);
        velocidadPlot.addYTick("65", 65);//0
        velocidadPlot.addYTick("70", 70);
        velocidadPlot.addYTick("80", 80);//antes -pi/2
        velocidadPlot.setMarksStyle("none");
        velocidadPlot.setImpulses(true);
        velocidadPlot.addLegend(0,"optima");
        velocidadPlot.addLegend(10,"actual");
```

```
                velocidadPlot.setImpulses(false);//para quitar los Stems

                alturaPlot.setConnected(true, 1);

                gridbag = new GridBagLayout();
                c = new GridBagConstraints();
                getContentPane().setLayout(gridbag);

                // Handle the rumboPlot
                c.gridx = 0;
                c.gridy = 0;
                c.gridwidth = 1;
                c.gridheight = 1;
                c.weightx = 2.0;
                c.weighty = 2.0;
                c.fill = GridBagConstraints.HORIZONTAL;
                gridbag.setConstraints(rumboPlot, c);
                getContentPane().add(rumboPlot);

                // Handle the alturaPlot
                c.gridx = 1;
                c.gridy = 0;
               // c.gridwidth = 4;
                c.fill = GridBagConstraints.HORIZONTAL;
                gridbag.setConstraints(alturaPlot, c);
                getContentPane().add(alturaPlot);

                //ctrlsPlot
                c.gridx = 0;
                c.gridy = 1;
                c.fill = GridBagConstraints.HORIZONTAL;
                gridbag.setConstraints(ctrlsPlot, c);
                getContentPane().add(ctrlsPlot);

                //Velocidad plot
                c.gridx = 1;
                c.gridy = 1;
                c.fill = GridBagConstraints.HORIZONTAL;
                gridbag.setConstraints(velocidadPlot, c);
                getContentPane().add(velocidadPlot);


                setVisible(true);


            }//const

               public void pintar(){
                 boolean first = true;
                     //for (int i = 0; i <= 10; i++){
                     try{
                            alturaPlot.addPoint(1, timeInSeconds = (
System.currentTimeMillis()-initTime ) / 1000 , fgc.getAltura(), first);
                            alturaPlot.addPoint(0, timeInSeconds = (
System.currentTimeMillis()-initTime ) / 1000 , fgc.setPointAltura, first);
                            rumbo=fgc.getRumbo();
                            if(rumbo<0)rumbo=rumbo+360;
                            rumboPlot.addPoint(2, timeInSeconds = (
System.currentTimeMillis()-initTime ) / 1000 , rumbo, first);
```

```
                        rumboPlot.addPoint(0, timeInSeconds = (
System.currentTimeMillis()-initTime ) / 1000 , fgc.setPointRumbo, first);
                        ctrlsPlot.addPoint(3, timeInSeconds = (
System.currentTimeMillis()-initTime ) / 1000 , fgc.gs.ctrls.elevator, first);
                        ctrlsPlot.addPoint(4, timeInSeconds = (
System.currentTimeMillis()-initTime ) / 1000 , fgc.gs.ctrls.aileron, first);
                        ctrlsPlot.addPoint(0, timeInSeconds = (
System.currentTimeMillis()-initTime ) / 1000 , 0, first);
                        velocidadPlot.addPoint(10, timeInSeconds = (
System.currentTimeMillis()-initTime ) / 1000 , fgc.fdm.vcas, first);
                        velocidadPlot.addPoint(0, timeInSeconds = (
System.currentTimeMillis()-initTime ) / 1000 , 61, first);

                        alturaPlot.fillPlot();//fillPlot() para que la gráfica
se desplace automáticamente por ejes XY
                        rumboPlot.fillPlot();
                        ctrlsPlot.fillPlot();
                        velocidadPlot.fillPlot();
                    }//try
                    catch(NullPointerException npe){
                        //continue;
                    }//catch

                   first = false;

}//clase GraficasFG
```

Clase TimerGrafica

```
import java.util.TimerTask;

import javax.swing.SwingUtilities;

public class TimerGrafica extends TimerTask{
      GraficasFG grafica;
      flightGearController fgc;

      public TimerGrafica(flightGearController fgc){
            this.fgc=fgc;
            grafica=new GraficasFG(fgc);
      }//constructor

      public void run(){

                grafica.pintar();
      }//run
}//clase
```

Clase pid

```
public interface pid {
      public double activar(double setPoint, double actual);  //actual es el
valor actual de la variable que queremos que alcance el valor de SetPoint
      public void setKp(double Kp);
      public double getKp();
      public void setTi(double Ti);
      public void setTd(double Td);
      public double getTi();
```

```
public double getTd();
}
```

Clase pidEstab

```
public class pidEstab implements pid {

	double vverticaloptima;
	double min, max;
	double vverticalactual;
	double ierror;//sumatorio de errorvv para la integral
	double errorvv;
	double derror; //diferencia error para la derivada
	double errorant;//variable en la que se guarda errorvv
	double cabeceo;
	double Ti;
	double Td;
	double Kp;//0.0005
	double Ki;
	double Kd;
	double t;
	double accionAnterior;
	double diferencia;
	double incrementoAccion;

	public pidEstab(){
		vverticaloptima=0;
		//vverticalactual=0;
		ierror=0;
		errorvv=0;
		derror=0;
		errorant=0;
		cabeceo=0;
		Ti=1/(25*0.5);	//25*...
		Td=0.01;
		Kp=0.008;//0.008
		Ki=Kp*Ti;
		Kd=Kp*Td;
		t=1;
		min=-9;
		max=9;
		accionAnterior=0;
		diferencia=0;
		incrementoAccion=0.15;

	}//constructor

	public double activar(double setPoint,double actual){

		if (setPoint < min) setPoint=min;
		else if (setPoint > max) setPoint=max;

		vverticaloptima=setPoint;
		vverticalactual=actual;
		errorvv=vverticaloptima-vverticalactual;
		ierror += errorvv;
		derror = errorvv - errorant;
```

```
		cabeceo=errorvv*Kp+ ierror*Ki*t + (derror*Kd)/t;//t=1
		diferencia=accionAnterior-cabeceo;
		accionAnterior=cabeceo;
		errorant = errorvv;

		return cabeceo;
	}

	public void setKp(double Kp){
		this.Kp=Kp;
	}
	public void setTi(double Ti){
		this.Ti=Ti;
	}
	public void setTd(double Td){
		this.Td=Td;
	}
	public double getKp(){
		return Kp;
	}
	public double getTi(){
		return Ti;
	}
	public double getTd(){
		return Td;
	}
}
```

Clase pidAlabeo

```
public class pidAlabeo implements pid{

	double rollOptimo;
	double min, max;
	double rollActual;
	double ierror;//sumatorio de errorvv para la integral
	double errorvv;
	double derror; //diferencia error para la derivada
	double errorant;//variable en la que se guarda errorvv
	double aleron;
	double Ti;
	double Td;
	double Kp;//0.0005
	double Ki;
	double Kd;
	double t;

	public pidAlabeo(){
		rollOptimo=0;
		rollActual=0;
		ierror=0;
		errorvv=0;
		derror=0;
		errorant=0;
		aleron=0;
		Ti=80; //7/2
		Td=0.0009;//7/8; //0.04
		Kp=0.1;            //0.5
```

```
		Ki=Kp/Ti;
		Kd=Kp*Td;
		t=1;
		min=-0.34;   //0.34 radianes de roll, es decir 20 grados de alabeo
		max=0.34;
	}//constructor

	public double activar(double setPoint,double actual){

		if (setPoint < min) setPoint=min;
		else if (setPoint > max) setPoint=max;

		rollOptimo=setPoint;
		rollActual=actual;
		errorvv=rollOptimo-rollActual;
		ierror += errorvv;
		derror = errorvv - errorant;

		aleron=errorvv*Kp+ ierror*Ki*t+ (derror*Kd)/t;//t=1

		errorant = errorvv;

		return aleron;
	}

	public void setKp(double Kp){
		this.Kp=Kp;
	}
	public void setTi(double Ti){
		this.Ti=Ti;
	}
	public void setTd(double Td){
		this.Td=Td;
	}
	public double getKp(){
		return Kp;
	}
	public double getTi(){
		return Ti;
	}
	public double getTd(){
		return Td;
	}

}//clase
```

Clase pidAltura

```
public class pidAltura implements pid{

	double variacionAltura;//es lo que tenemos a la salida del pid
	double alturaoptima;
	double alturactual;
	double ierror;//sumatorio de errorvv para la integral
	double errorvv;
	double erroraltura;
	double derror; //diferencia error para la derivada
	double errorant;//variable en la que se guarda errorvv
	double cabeceo;
```

```
	double Ti;
	double Td;
	double Kp;//0.0005
	double Ki;
	double Kd;
	double t;

	public pidAltura(){
		variacionAltura=0;
		//alturactual=0;
		ierror=0;
		errorvv=0;
		derror=0;
		errorant=0;
		cabeceo=0;
		Ti=0.1;//0.08
		Td=1;//0.01
		Kp=0.55;//0.5
		Ki=Kp*Ti;//Ki=Kp/Ti
		Kd=Kp*Td;
		//si no pone aquí lo contrario, poner parámetros igual que los
comentados(21/07/2010)
		t=1;
	}//constructor
	public double activar(double setPoint,double actual){
		alturaoptima=setPoint;
		alturactual=actual;
		erroraltura=alturaoptima-alturactual;//fdm.altitude; la mandamos
desde gui

		ierror+=erroraltura;
		derror=erroraltura-errorant;
		variacionAltura=erroraltura*Kp;//+ierror*Ki*t;//+(derror*Kd)/t;
		errorant=erroraltura;

		return variacionAltura;//es decir, la velocidad vertical
	}

	public void setKp(double Kp){
		this.Kp=Kp;
	}
	public void setTi(double Ti){
		this.Ti=Ti;
	}
	public void setTd(double Td){
		this.Td=Td;
	}
	public double getKp(){
		return Kp;
	}
	public double getTi(){
		return Ti;
	}
	public double getTd(){
		return Td;
	}
}
```

Clase pidRumbo

```
public class pidRumbo implements pid{

	double rumboOptimo;
	double rumboActual;
	double ierror;//sumatorio de errorvv para la integral
	double errorRumbo;
	double derror; //diferencia error para la derivada
	double errorant;//variable en la que se guarda errorvv
	double alabeoOptimo;
	double Ti;
	double Td;
	double Kp;//0.0005
	double Ki;
	double Kd;
	double t;
	double min;
	double max;
	GestorRumbo gr;
	double []arrayRumbo;

	public pidRumbo(){

		rumboOptimo=0;	//velocidad este
		rumboActual=0;  //velocidad este
		ierror=0;
		errorRumbo=0;
		derror=0;
		errorant=0;
		alabeoOptimo=0;
		Ti=0.000005;//0.000005
		Td=10;//10 probar con 11 el proximo dia (hoy es 24/05/2010)
		Kp=1.1;//0.8
		Ki=Kp*Ti;
		Kd=Kp*Td;
		t=1;
		gr=new GestorRumbo();
		arrayRumbo=new double[2];

	}//constructor

public double activar(double setPoint,double actual){

		arrayRumbo=gr.activar(setPoint, actual);
		rumboOptimo=arrayRumbo[0];
		rumboActual=arrayRumbo[1];
System.out.println("Rumbo actual en grados:"+Math.toDegrees(rumboActual));
		errorRumbo=rumboOptimo-(rumboActual);
		ierror += errorRumbo;
		derror = errorRumbo - errorant;

		alabeoOptimo=errorRumbo*Kp + ierror*Ki*t + (derror*Kd)/t;//t=1

		errorant = errorRumbo;

		return alabeoOptimo;
	}
```

```
	public void setKp(double Kp){
		this.Kp=Kp;
	}
	public void setTi(double Ti){
		this.Ti=Ti;
	}
	public void setTd(double Td){
		this.Td=Td;
	}
	public double getKp(){
		return Kp;
	}
	public double getTi(){
		return Ti;
	}
	public double getTd(){
		return Td;
	}
}//clase
```

Clase GestorRumbo

```
public class GestorRumbo {
double optimo,actual;
double [] array;//contiene la transformacion de los rumbos y un indicador
                                                    //para la clase pidRumbo.

	public GestorRumbo(){
		optimo=0;
		actual=0;
		array=new double [2];
	}//constructor
	public double[] activar(double rumboOptimo, double rumboActual){//función
que cambia el intervalo de valores en el que varían los angulos, de 0-360 a +-
180
		optimo=rumboOptimo;//pidRumbo nos lo pasa en grados [0,360]
		if(optimo==0)optimo=360;
		if(rumboActual<0)actual=Math.toDegrees(Math.PI*2+rumboActual);//
grados [0,360]
		else actual=Math.toDegrees(rumboActual);//si no es negativo no hace
falta el cambio: los valores positivos son iguales en +-180 con respecto a 0-
360
		System.out.println("Verificación Rumbo actual:"+rumboActual);

		if(optimo<180 && optimo!=0 && actual>180 && Math.abs(optimo-
actual)>180){
			actual=Math.toRadians(actual-360);//cambiamos el rango
de valores que pueden tomar los ángulos [-180,180]
			optimo=Math.toRadians(optimo);
			System.out.println("Caso 1: Transformación rumbo
actual [-180,180]");
		}//primer if

		else if(optimo>180 && actual<180 && Math.abs(optimo-actual)>180){
			optimo=Math.toRadians(optimo-360);//cambiamos el rango
de valores que pueden tomar los ángulos [-180,180]
			actual=Math.toRadians(actual);
```

```
                        System.out.println("Caso 2: Transformación rumbo
optimo [-180,180]");
            }//primer elseif
            else if(optimo==0 || optimo==360){//caso que el setpoint sea 0
                  optimo=360;
                  optimo=Math.toRadians(optimo-360);
                  if (actual>180)actual=Math.toRadians(actual-360);
                  System.out.println("Caso 3: Transformación rumbo actual [-
180,180] setPoint=0");
            }//segundo else if
            else{//caso en el que no hay que transformar el rango de los
ángulos, x tanto estamos en [0,360]
                  optimo=Math.toRadians(optimo);
                  actual=Math.toRadians(actual);
                  System.out.println("Caso 4: No transformación");
            }//último else
            //falta devolver ángulos en caso de que no se cumplan estos if, por
tanto entre 0 y 6.28 rad. y signo=0;
            array[0]=optimo;
            array[1]=actual;
            return array;//los ángulos se devuelven en radianes.
      }//activar
}//clase
```

ANEXO II. ESPECIFICACIONES DEL AVIÓN ELEGIDO

El Piper j-3 Cub es un avión simple, pequeño y liviano. Cuenta con dos asientos en tándem (longitudinalmente), este avión fue fabricado por la empresa Piper Aircraft entre los años 1934 y 1947 y fue pensado para el entrenamiento de pilotos (avión entrenador). Dadas sus características de vuelo, esta aeronave se transformó en un clásico por su simplicidad de vuelo y maniobrabilidad a la hora de entrenar a pilotos novatos. Hoy en día sigue siendo muy usado por muchas escuelas de vuelo.

Figura 54. Piper J3 Cub.

La historia del J3 comienza en la década del ´20 cuando se crea el monoplano Chummy, su antecesor. Durante la gran depresión de la década del 30, el petrolero William T. Piper compró la producción e introdujo el Taylor E-2, que más tarde, con la llegada de los motores Continental, dió lugar al Taylor J-2, el antecesor más cercano del J-3. En 1937 la fábrica se muda a Pennsylvania y cambia su nombre a Piper Aircraft Corporation. Es allí donde nace el primer Piper J-3 Cub, en 1938.

El Piper J-3 fue de gran ayuda durante la Segunda Guerra Mundial, junto al J-4, sirvieron de avión de observación y ambulancia. Sólo en 1940 se construyeron 3016, saliendo de la fábrica uno cada 20 minutos. En 1947 cuando terminó su fabricación, se habían realizado ya 14.125 Piper J-3 Cub.

ESPECIFICACIONES

Características generales

Tripulación: 1 piloto

Capacidad: 1 pasajero

Longitud: 6,8 m (22,4 ft)

Envergadura: 10,7 m (35,2 ft)

Altura: 2 m (6,7 ft)

Superficie alar: 16,6 m2 (178,5 ft2)

Peso vacío: 345 kg (760,4 lb)

Peso útil: 208,8 kg (460,3 lb)

Peso máximo al despegue: 550 kg (1.212,2 lb)

Planta motriz: 1× motor bóxer de cuatro cilindros Continental A-65-8.

Potencia: 66 kW (91 HP; 90 CV) a 2.350 rpm

Rendimiento

Velocidad nunca excedida (Vne): 196,3 km/h (122 MPH; 106 kt)

Velocidad máxima operativa (Vno): 140 km/h (87 MPH; 76 kt)

Velocidad crucero (Vc): 121 km/h (75 MPH; 65 kt)

Alcance: 354 km (191 nmi; 220 mi)

Techo de servicio: 3.500 m (11.483 ft)

Régimen de ascenso: 2,3 m/s (453 ft/min)

Carga alar: 33,4 kg/m2 (6,8 lb/ft2)

Potencia/peso: 11.35

ANEXO III. INSTALACIÓN DE SOFTWARE

Descarga de FlightGear.

Para descargar el simulador vamos a su página web http://www.flightgear.org/ una vez allí pinchamos en el menú desplegable superior "Get FlightGear" y a continuación pinchamos en "Download Main Program". Lo vemos en la figura 55:

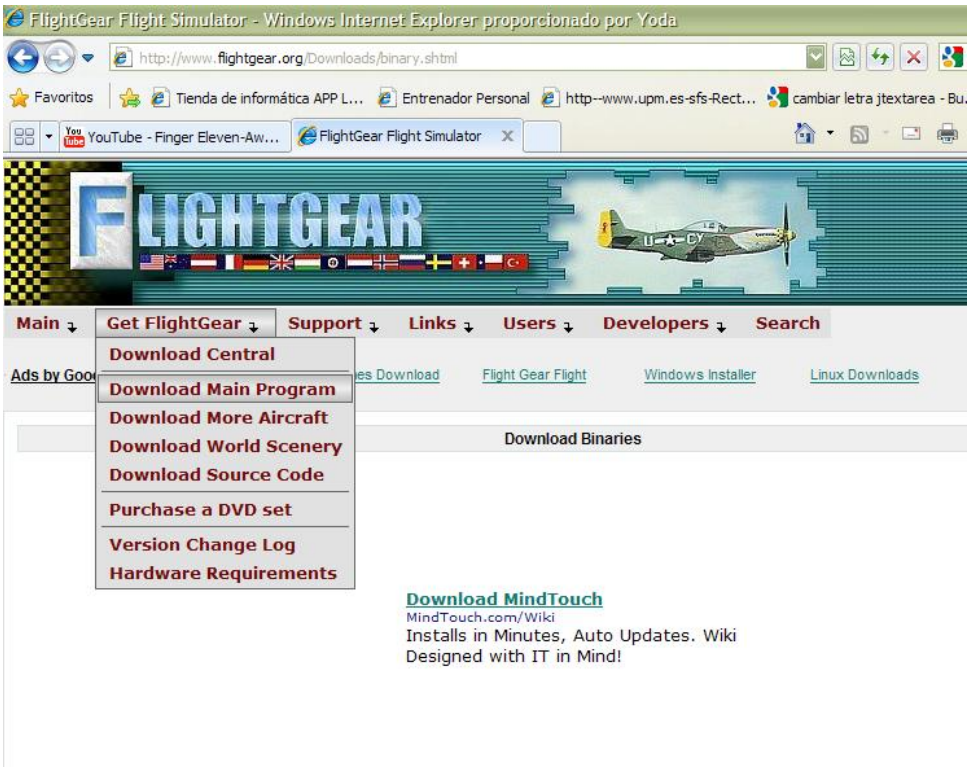


Figura 55. Página principal de FG.

Una vez hecho esto, nos aparecerá una pantalla como la que aparece en la figura 56. Se trata de elegir el Sistema Operativo en el cual trabajará FlightGear, como nuestro pc trabaja sobre Windows XP pinchamos en el servidor correspondiente a Windows. A continuación comienza la descarga del instalador de FG.

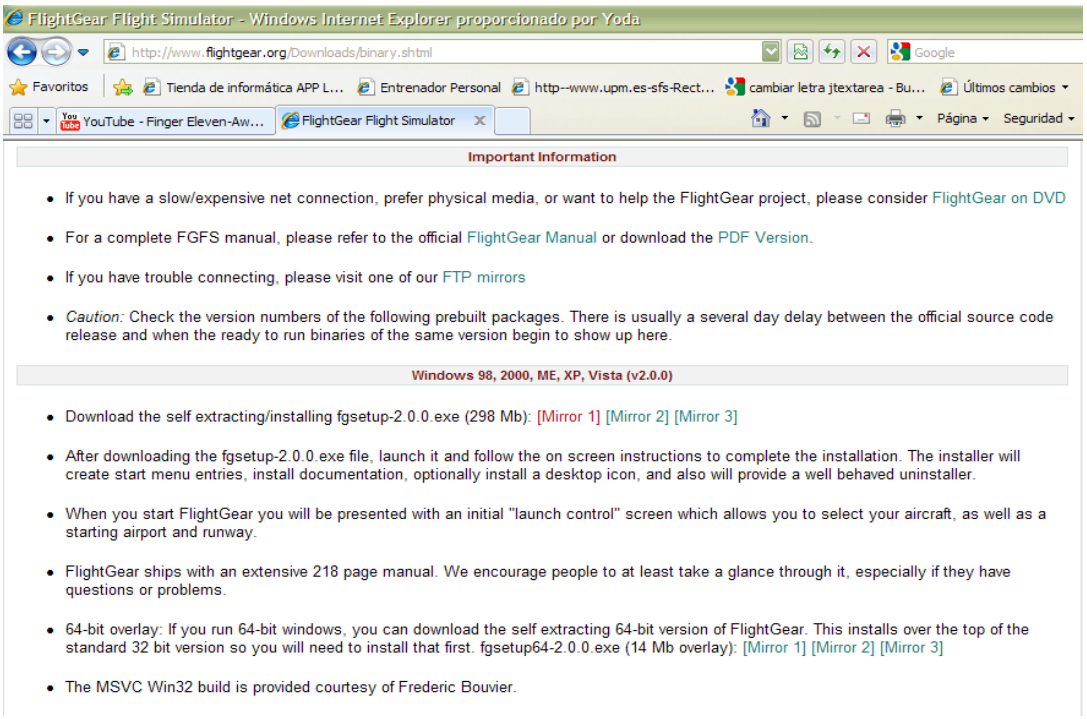


Figura 56. Elección del S.O.

Una vez descargado el instalador de FG, seguimos los pasos para instalar el simulador de vuelo, de tal modo que al final nos aparece el icono de inicio rápido como el de la figura 57.

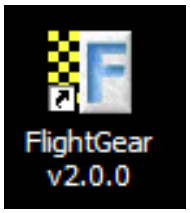


Figura 57. Icono de acceso directo a FlightGear.

Cuando pinchamos en el icono nos aparece una pantalla llamada “FlgihtGear Wizard” en donde lo primero que se nos permite hacer es la selección del avión. Es importante decir que desde la página oficial de FlightGear podemos descargarnos muchos aviones más, para ello tendremos que pinchar en el ya mencionado menú desplegable “Get FlightGear” y seleccionar “Download more aircraft”. Nos aparecerá una lista de aviones con sus correspondientes fotos, simplemente pinchamos sobre los aviones que queramos bajarnos, en formato .zip y los descomprimimos en la carpeta llamada “Aircraft” que se ubica en la carpeta “data” que a su vez está en la carpeta principal llamada “FlightGear”, que por defecto se instala en “Archivos de Programa”. Es decir, en nuestro caso “C:\\Archivos de Programa\FlightGear\data\Aircraft”, así la próxima vez que abramos el FlightGear Wizard la lista de aviones ya estará actualizada y ya estarán disponibles los aviones descargados. Figura 58.

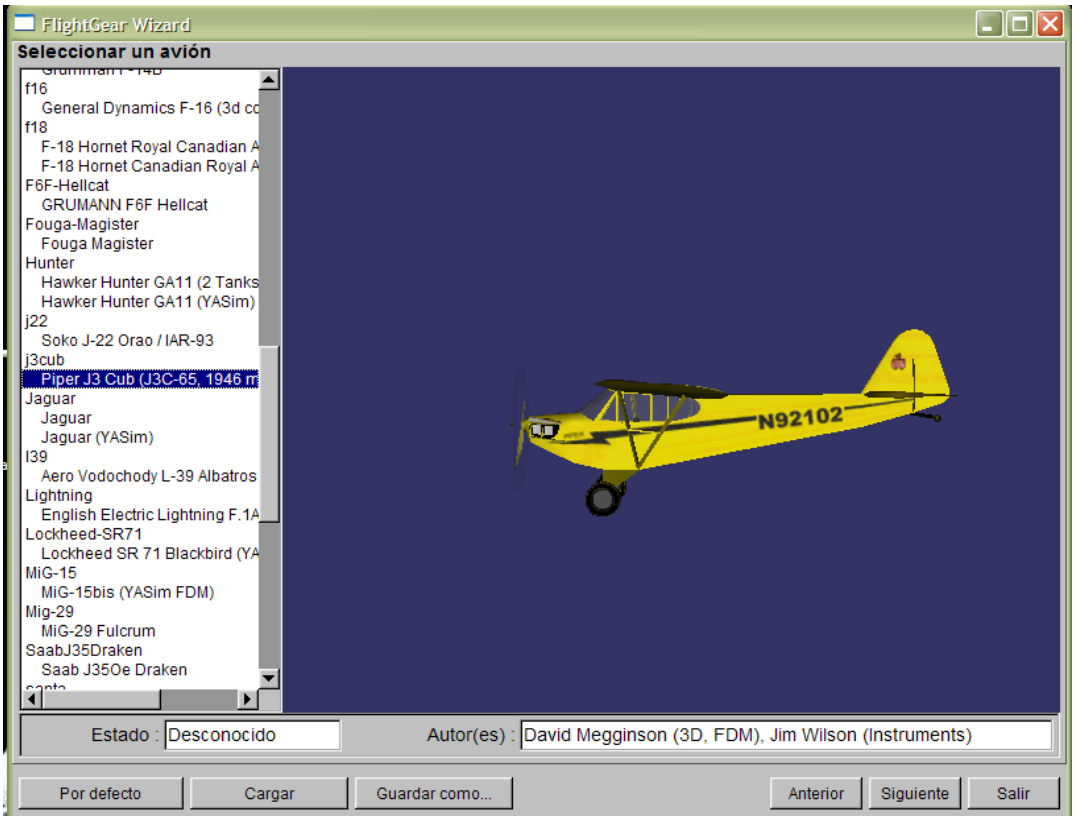


Figura 58. Elección del avión.

Cuando hayamos elegido el avión pulsamos sobre el botón "Siguiente" y nos aparecerá la pantalla para elegir aeropuerto. Al igual que sucede con los aviones, también podemos descargarnos más aeropuertos de todo el mundo desde la página oficial. Por defecto FlightGear trae los escenarios de los alrededores del área del Aeropuerto Internacional de San Francisco. Para ampliar el número de escenarios pinchamos en "Get FlightGear", "Download World Scenary" y nos aparecerá un mapa mundial dividido por zonas, las cuales nos indican en color verde si están disponibles o rojo si no lo están. Una vez hayamos pinchado sobre las zonas deseadas se descargarán en nuestro ordenador. Para instalarlas vamos al menú de inicio, seleccionamos "Flightgear", "tools", "Install & Uninstall Scenary" nos aparecerá una ventana con los escenarios que hayamos descargado pero que aún no están instalados en nuestro ordenador, seleccionamos los que queramos y pulsamos sobre el botón "Install Selected Files", automáticamente la próxima vez que abramos el FlightGear Wizard nos aparecerá la lista de aeropuertos ya actualizada. En nuestro caso elegimos descargarnos el escenario que contenía al aeropuerto de San Javier. Elegido el aeropuerto clicamos en "Siguiente" y nos aparece una pantalla como la de la figura 59:

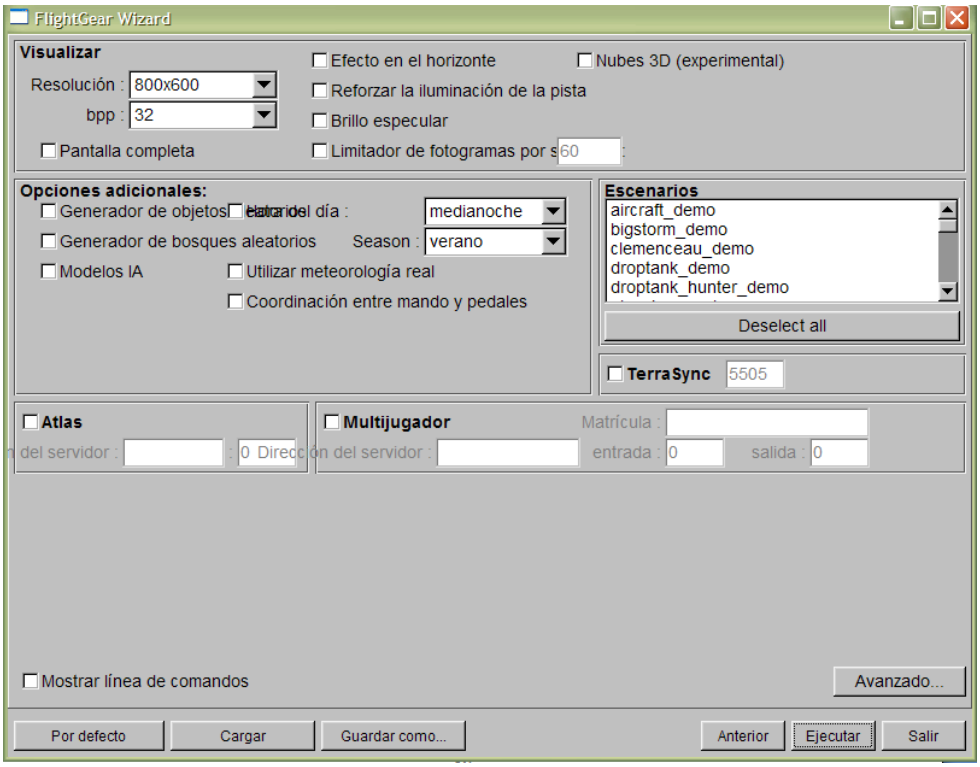


Figura 59. Pantalla de opciones.

No vamos a entrar en detalles sobre ésta pantalla, si pulsamos sobre el botón "Ejecutar" lanzaremos el simulador. Si pulsamos en "Avanzado…" iremos a un menú de opciones avanzadas muy interesante, ya visto en *3.1.1. Implementación del protocolo de comunicación*

Descarga de código fuente de FG.

Es muy fácil, vamos a la página web http://www.flightgear.org/ pinchamos en *Get FlightGear* y a continuación en el menú desplegable pinchamos en *download source code.*

Descarga de Wireshark.

Para analizar las tramas que enviamos y recibimos descargamos el programa Wireshark. Es un analizador de protocolos basado en las librerías pcap [14] utilizado comúnmente como herramienta de diagnóstico de redes y de desarrollo de aplicaciones de red. En la figura 60 vemos la página web desde dónde nos descargamos la aplicación.

[14] Librerías que proporcionan funciones para la captura de paquetes a nivel de usuario.

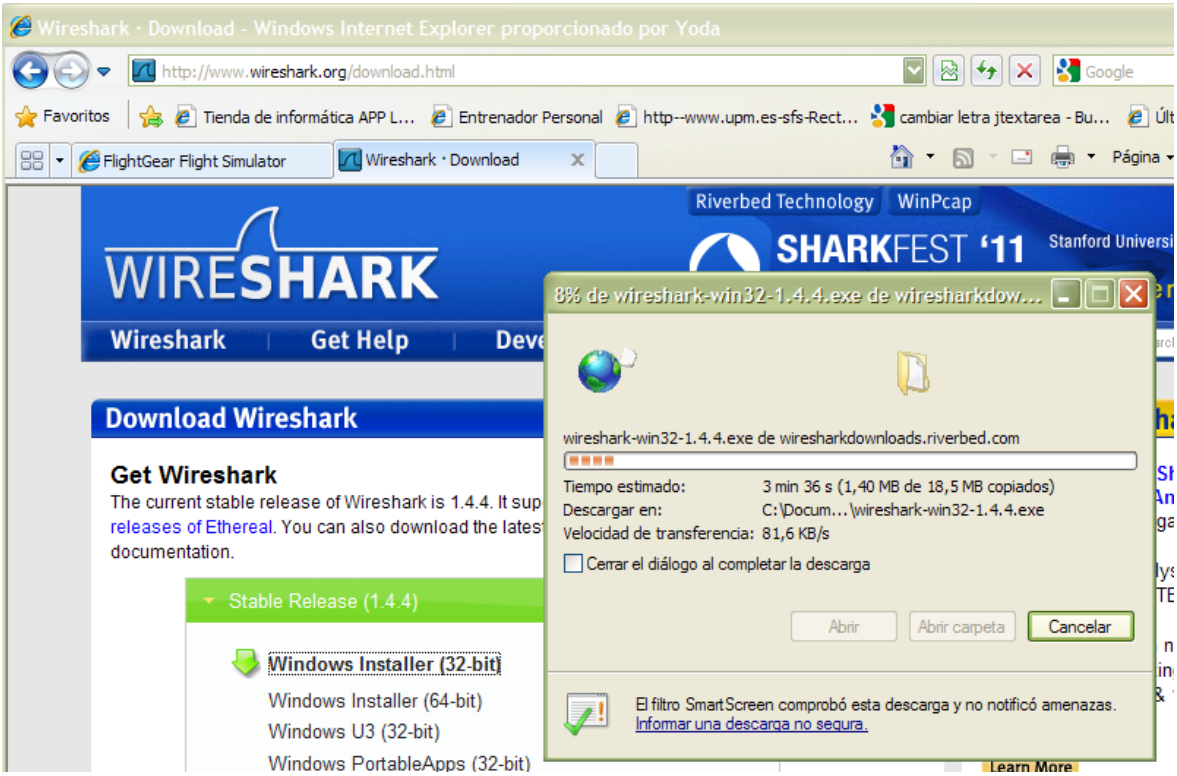


Figura 60. Descargando Wireshark.

Para descargar el instalador del programa sólo tenemos que entrar en su página web http://www.wireshark.org/ y pinchamos sobre el enlace de descarga, seleccionamos nuestra versión de Windows, en nuestro caso la de 32 bits y comienza la descarga.

Pinchamos en el instalador y comienza la instalación del programa y también la de las librerías pcap. Una vez instalado el programa, lo abrimos, clicamos en "Capture/Capture Options" y veremos una pantalla igual a la de la figura 61, elegimos la interfaz sobre el que queramos capturar los datos enviados y recibidos en la casilla "Interface" y pulsamos en el botón inferior "Start". Así es como Wireshark comienza a capturar las tramas entrantes y salientes sobre la interfaz seleccionada.

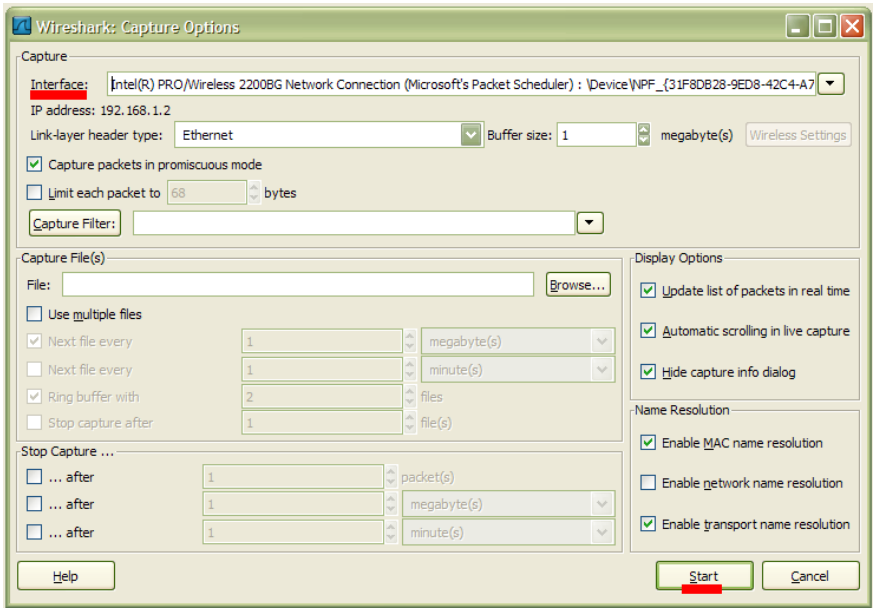


Figura 61. Opciones de captura de Wireshark.

Descarga de eclipse.

Eclipse es un IDE [15] (entorno de desarrollo integrado) de código abierto y multiplataforma, en nuestro caso el lenguaje de programación es Java, para el desarrollo de aplicaciones. Fue desarrollado por IBM para sustituir a VisualAge, aunque ahora es Eclipse Foundation quien se encarga de su desarrollo, es por esto que Eclipse es de código abierto. Eclipse también es una comunidad de usuarios.

Podemos descargar el programa desde su página web oficial http://www.eclipse.org/downloads/, figura 62. Pinchamos en "Eclipse IDE for Java EE Developers" en la versión de 32 bits para Windows. Al lado de la descarga hay un aviso que nos recuerda que para que eclipse funcione necesitamos tener instalado Java runtime environment (JRE) o Java development kit (JDK). Se pueden descargar desde la misma página.

Una vez bajado, se accede a él directamente entrando en su carpeta correspondiente. No necesita instalación.

[15] Un IDE es un entorno de programación que ha sido empaquetado como un programa de aplicación. Consiste en un editor de código, un compilador, un depurador y un constructor de interfaz gráfica (GUI).

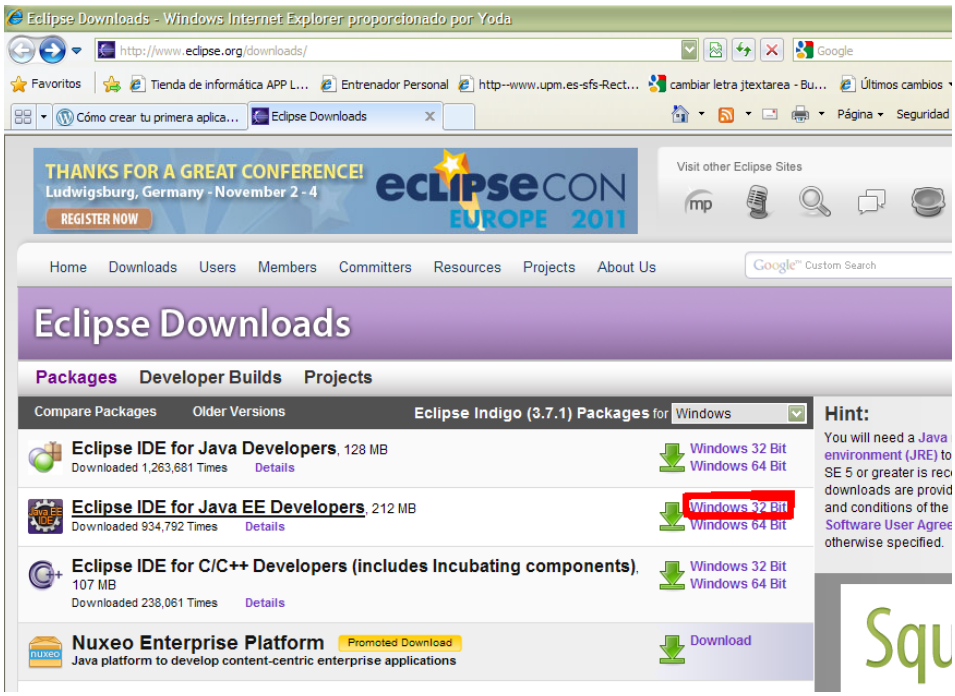


Figura 62. Descarga de Eclipse.

BIBLIOGRAFÍA

[1] Página web: http://www.upm.es/sfs/Rectorado/Gabinete%20del%20Rector/Revista%20UPM/Historico/UPMocho.pdf

[2] Página web: http://asterion.almadark.com/2006/12/29/tutorial-basico-del-simulador-de-vuelo-flightgear/

[3] Página web: http://www.manualvuelo.com/TCV/TCV50.html.

[4] Página web: http://www.sapiensman.com/control_automatico/control_automatico6.htm

[5] Página web: http://www.elprisma.com/apuntes/ingenieria_quimica/regulaciondeprocesos/default4.as

[6] Página web: http://www.tec.uji.es/asignatura/dir_asignaturas/3/67/Tema1.pdf

[7] Página web: http://chuwiki.chuidiang.org/index.php?title=Hilos_en_Java

[8] Página web: http://www.coderanch.com/t/416796/Beginning-Java/java/Event-handling-Implementing-ActionListener-vs

[9] Página web: http://www.chuidiang.com/chuwiki/index.php?title=JScrollPane