

Proyecto Fin de Máster

Máster Universitario en Ingeniería Aeronáutica

Diseño de herramientas de simulación de vuelo
basadas en software de código abierto

Autor: Miguel Vitoria Gallardo

Tutor: Dr. Francisco Gavilán Jiménez

Dpto. de Ingeniería Aeroespacial y Mecánica de
Fluidos

Escuela Técnica Superior de Ingeniería

Sevilla, 2021



Proyecto Fin de Máster
Máster Universitario en Ingeniería Aeronáutica

Diseño de herramientas de simulación de vuelo basadas en software de código abierto

Autor:

Miguel Vitoria Gallardo

Tutor:

Francisco Gavilán Jiménez

Profesor Contratado Doctor

Dpto. de Ingeniería Aeroespacial y Mecánica de Fluidos

Escuela Técnica Superior de Ingeniería

Universidad de Sevilla

Sevilla, 2021

Proyecto Fin de Carrera: Diseño de herramientas de simulación de vuelo basadas en software de código abierto

Autor: Miguel Vitoria Gallardo

Tutor: Francisco Gavilán Jiménez

El tribunal nombrado para juzgar el Proyecto arriba indicado, compuesto por los siguientes miembros:

Presidente:

Vocales:

Secretario:

Acuerdan otorgarle la calificación de:

Sevilla, 2021

El Secretario del Tribunal

A todos aquellos que confiaron en mí, y sobretodo, a mi familia.

Agradecimientos

A mis padres, a mi hermana y a mi familia por confiar en mí, ayudarme en todo momento y darme siempre la posibilidad avanzar.

A mis amigos por servirme de ancla y darme consejo, siempre obsequiándome con diferentes puntos de vista.

A todas las personas involucradas en el CD Rugby Mairena, de las cuales he aprendido el valor del trabajo, el sacrificio y me han dado tantas satisfacciones como retos.

A María Bueno Fernández, por confiar en mi a la hora de seleccionarme para la beca que me sirvió de impulso para introducirme en el mundo laboral, ampliar mis conocimientos en el mundo aeronáutico y la oportunidad de conocer las virtudes de la simulación.

Al tutor de este proyecto, Francisco Gavilán Jiménez, por haber apostado desde el primer día por este proyecto y haberme incitado a ir aún más allá, haciendo posible el desarrollo total del mismo.

A todos ellos y a muchos más, gracias.

Miguel Vitoria Gallardo

Toulouse, 2021

Con el objetivo de implementar un simulador funcional, se hizo un análisis de varias herramientas utilizadas durante el Máster Universitario en Ingeniería Aeronáutica, siéndole estudiadas en profundidad *Matlab/Simulink*, *Python* y *Flightgear*.

Se inicio un proceso de estudio individualizado de cada una de las herramientas para poder explorar sus virtudes e inconvenientes. Para *Matlab/Simulink* se implementó un prototipo de un modelo de dinámica de vuelo modular.

Para modelo de dinámica de vuelo *JSBSim* se hizo un estudio exhaustivo, donde se integra la librería para el desarrollo un prototipo de simulador de vuelo en ley directa basado en *Python* y se ejecutan varios test predefinidos en bucle abierto para el estudio de la respuesta de las aeronaves ante diferentes escenarios.

Finalmente, para *Flightgear*, se hizo un estudio de todas las posibilidades y herramientas disponibles para su utilización como núcleo central de la simulación.

Después de un testeo intensivo a cada una de ellas, se llegó a la conclusión que una simulación matemática veloz es necesaria a la hora del diseño e integración de diferentes herramientas, pero debe estar acompañada siempre de herramientas con interfaces gráficas que ayudan a su comprensión y la manipulación de los parámetros de control desde un punto de vista del usuario final.

Para conseguir los objetivos fijados, se implementó una cabina física de estructura escalable mediante la utilización de *JSBSim* y *Flightgear* como base de la simulación, y una red LAN con el objetivo de hacer las conexiones entre las diferentes herramientas externas, siéndole desarrollada un panel del autopiloto totalmente funcional gracias a herramientas *Python*.

Abstract

With the aim to implement an actual simulator, an analysis has been performed among several tools utilised during the University Master in Aeronautic Engineering, where *Matlab/Simulink*, *Python* y *Flightgear* were deeply studied.

To that end, an individualized study of each tool has been performed, in order to expose their advantages and disadvantages. For *Matlab/Simulink*, it has been implemented a prototype of a modular flight dynamic model.

It has been performed an exhaustive study for the flight dynamic model *JSBSim*, where this library has been implemented to developed a prototype of a direct-law flight simulator based in Python and where several predefined tests in open-loop have been executed to study the response of the aircraft against different scenarios.

Finally, for *Flightgear*, a study of all possibilities and available tools has been performed to explore the chance of using it has a central hub of the simulation.

After an intensive testing process for each solution, it was concluded that a powerful mathematical simulation is mandatory for designing and development goals, but it should come along with graphical interfaces tools for enhancing the comprehension and the feasibility of manipulating the control parameters of the final user.

To achieve the original objectives, a functional scalable cockpit has been developed physically using *JSBSim* and *Flightgear* as the core of the simulation, and Local Area Network for the interconnection among the different external tools, where a totally functional autopilot panel has been developed through Python tools.

Agradecimientos	ix
Resumen	xi
Abstract	xiii
Índice	xiv
Índice de Figuras	xvi
1 Motivación e introducción	21
1.1. <i>Motivación del proyecto</i>	21
Un simulador en código libre	21
Un simulador modular	22
Un simulador versátil	22
1.2. <i>Introducción histórica</i>	22
1.3. <i>Estructura teórica del simulador</i>	25
1.4. <i>Estructura del documento</i>	26
2. Ecuaciones integradas en los modelos dinámicos.	29
2.1. <i>Sistemas de referencia</i>	29
2.2. <i>Velocidades angulares</i>	30
2.3. <i>Velocidades lineales</i>	31
2.4. <i>Ecuaciones del movimiento y rotación</i>	31
2.5. <i>Modelo aerodinámico</i>	31
2.6. <i>Modelo propulsivo</i>	33
3. Exploración de viabilidad/opciones disponibles	34
3.1. <i>Matlab/Simulink</i>	34
3.1.1. Descripción de los distintos módulos	36
3.1.2. Implementación	46
3.1.3. Ventajas e inconvenientes	49
3.1.4. Conclusión	50
3.2. <i>JSBSim</i>	51
3.2.1. Introducción a <i>JSBSim</i>	51
3.2.2. Archivo de configuración de avión	53
3.2.3. Archivo de inicialización de la aeronave	71
3.2.4. Modos de funcionamiento de <i>JSBSim</i>	72
3.2.5. Ventajas e inconvenientes	85
3.2.6. Conclusiones	86
3.3. <i>Flightgear</i>	88
3.3.1. Introducción a <i>Flightgear</i>	88
3.3.2. Modos de lanzamiento	89
3.3.3. <i>Property tree</i> y herramientas auxiliares	92
3.3.4. Ventajas e inconvenientes	104
3.3.5. Conclusiones	105
3.4. <i>Decision entre las soluciones disponibles</i>	106
4. Desarrollo del simulador de vuelo	107

4.1.	<i>Arquitectura del simulador</i>	107
4.2.	<i>Desarrollo APpanel</i>	111
4.2.1	Apartado gráfico	112
4.2.2	Apartado de funcionalidades	114
4.3.	<i>Resultados y conclusiones</i>	129
4.2.3	Ventajas e inconvenientes	135
5	Conclusiones	136
	Referencias	137
	ANEXO A: Manual de despliegue	139
	ANEXO A: Código Python JSBSim simulación tiempo real	11
	ANEXO B: Código Python Prueba pygame	14

ÍNDICE DE FIGURAS

Ilustración 1 Simulador fabricado por la compañía Antoinette	23
Ilustración 2 Sesión de entrenamiento en un Link Trainer	23
Ilustración 3 Bahía de simuladores del ITC, Sevilla	24
Ilustración 4 El entorno de simulación OpenEaagles	26
Ilustración 5 Representación de los sistemas de referencia [1] <i>JSBSim: AN open source, platform-independent, flight dynamics model in C++</i> , Jon S. Berndt, 2011	29
Ilustración 6 Módulos de simulación	35
Ilustración 7 Bloque integrador	36
Ilustración 8 Detalle bloque integrador	37
Ilustración 9 Detalle de la codificación de las ecuaciones dinámicas	37
Ilustración 10 Detalle obtención coordenadas ejes ECEF	38
Ilustración 11 Bloque aerodinámico	39
Ilustración 12 Detalle bloque cálculo de incidencias	40
Ilustración 13 Detalle cálculo fuerza de resistencia o Drag	40
Ilustración 14 Motor eléctrico acoplado a hélice	41
Ilustración 15 Detalle diagrama del bloque estimador motor eléctrico acoplado a hélice parte uno	42
Ilustración 16 Detalle diagrama del bloque estimador motor eléctrico acoplado a hélice parte dos	42
Ilustración 17 Bloque atmosfera estándar internacional	43
Ilustración 18 Detalle de las ecuaciones de la atmosfera estándar internacional	43
Ilustración 19 Sub bloque modelo de Tierra	44
Ilustración 20 Detalle ecuaciones de modelo de tierra WGS84	44
Ilustración 21 Sub bloque cálculo coordenadas ECEF	45
Ilustración 22 Detalle simulación de test para bloque aerodinámico y propulsivo	46
Ilustración 23 Detalle simulación completa parte uno	47
Ilustración 24 Detalle simulación completa parte dos	47
Ilustración 25 Detalle captura altitud	48
Ilustración 26 Detalle captura ángulo de balance	48
Ilustración 27 Detalle captura rumbo	49
Ilustración 28 Jerarquía de clases en JSBSim	52
Ilustración 29 Proceso de inicialización y utilización de la clase FGFDMEExec	53
Ilustración 30 Detalle codificación métrica	54
Ilustración 31 Detalle codificación masa y centrado	55
Ilustración 32 Detalle fuerzas de flotación	56
Ilustración 33 Detalle reacciones con tierra	57
Ilustración 34 Detalle reacción tipo BOGEY	57
Ilustración 35 Detalle ejemplo reacciones externas	58

Ilustración 36 Detalle planta propulsiva	59
Ilustración 37 Detalle ejemplo planta motora turbofán	60
Ilustración 38 Detalle ejemplo hélice nº1	61
Ilustración 39 Detalle ejemplo hélice nº2	62
Ilustración 40 Detalle modelo tanque genérico	63
Ilustración 41 Detalle tipo de combustible	63
Ilustración 42 Detalle modelo Concorde	64
Ilustración 43 Detalle modelo aerodinámico	65
Ilustración 44 Detalle variables de control	66
Ilustración 45 Detalle de deflexión de flaps	67
Ilustración 46 Detalle resistencia inducida por deflexión timón de profundidad	67
Ilustración 47 Detalle definición <i>socket</i> de entrada	68
Ilustración 48 Detalle comandos en los <i>sockets</i> de entrada	69
Ilustración 49 Detalle comando de lectura	69
Ilustración 50 Detalle comando de escritura	69
Ilustración 51 Detalle comando de información general	69
Ilustración 52 Detalle configuración estándar de un <i>socket</i> de salida	69
Ilustración 53 Detalle grupos de datos	70
Ilustración 54 Detalle salida tipo CSV	70
Ilustración 55 Detalle archivo CSV	70
Ilustración 56 Detalle salida tipo <i>socket</i>	71
Ilustración 57 Detalle salida tipo <i>Flightgear</i>	71
Ilustración 58 Detalle archivo de inicialización	72
Ilustración 59 Detalle script para ejecutar <i>JSBSim</i> en bucle cerrado	73
Ilustración 60 Detalle eventos disparados por condiciones	73
Ilustración 61 Detalle ejemplo genérico <i>Script</i>	74
Ilustración 62 Detalle ejecución <i>JSBSim</i> consola Windows	75
Ilustración 63 Detalle ejecución <i>JSBSim</i>	75
Ilustración 64 Detalle ejecución de <i>JSBSim</i> en consola	76
Ilustración 65 Detalle ejecución de <i>JSBSim</i> en consola, inicio del script	76
Ilustración 66 Detalle ejecución <i>JSBSim</i> , inicialización	77
Ilustración 67 CSV resultante de la ejecución de <i>JSBSim</i>	77
Ilustración 68 Detalle parámetros Altitude ASL (ft) y V_{total} (ft/s) respecto al tiempo obtenidos mediante <i>JSBSim</i>	78
Ilustración 69 Detalle parámetros Alpha (deg) y Theta (deg) respecto al tiempo obtenidos mediante <i>JSBSim</i>	78
Ilustración 70 Detalle parámetros máxicos obtenidos de <i>JSBSim</i>	78
Ilustración 71 Detalle inicialización <i>Pygame</i>	80
Ilustración 72 Detalle detección del mando	80
Ilustración 73 Detalle evento botón	80

Ilustración 74 Detalle evento <i>joystick</i>	81
Ilustración 75 Detalle salida tipo <i>Flightgear</i>	81
Ilustración 76 Detalle sincronización <i>Flightgear</i> con <i>JSBSim</i>	82
Ilustración 77 <i>Flightgear</i> usado como interfaz gráfica.	82
Ilustración 78 Detalle inicialización prototipo simulación tiempo real	83
Ilustración 79 Detalle bucle de simulación <i>JSBSim</i> en <i>Python</i>	84
Ilustración 80 Extracto datos de la simulación	84
Ilustración 81 Detalle prototipo más interfaz gráfica	85
Ilustración 82 Interfaz de descarga de escenarios	89
Ilustración 83 Detalle Launcher <i>Flightgear 2018.3.6</i>	90
Ilustración 84 Detalle programa lanzado <i>Flightgear</i>	91
Ilustración 85 Detalle ejemplo ejecución <i>Flightgear</i> desde consola Windows	91
Ilustración 86 Detalle árbol de propiedades	92
Ilustración 87 Detalle árbol de propiedades <i>fdm/</i>	93
Ilustración 88 Detalle árbol de propiedades <i>fdm/jsbsim</i>	93
Ilustración 89 Detalle <i>c172p-WebPanel</i>	94
Ilustración 90 Detalle código HTML <i>c172p-WebPanel</i>	95
Ilustración 91 Detalle <i>Flightgear</i> ejecutándose con <i>Phi</i>	96
Ilustración 92 Detalle <i>Phi</i> opción <i>Map</i>	96
Ilustración 93 Detalle <i>Environment/Position</i>	97
Ilustración 94 Detalle caso uno masa y centrado	97
Ilustración 95 Detalle caso dos masas y centrado	98
Ilustración 96 Ejemplo <i>checklist</i> caso C172P	98
Ilustración 97 Detalle interfaz <i>TQPanel</i>	99
Ilustración 98 Detalle esquema funcionamiento de Kivy	100
Ilustración 99 Detalle código <i>Python</i> ejemplo	101
Ilustración 100 Ejemplo <i>self</i> y <i>*args</i>	102
Ilustración 101 Detalle lenguaje <i>Kv</i>	103
Ilustración 102 Detalle aplicación <i>example</i>	104
Ilustración 103 Detalle salida de texto por consola	104
Ilustración 104 Detalle esquema de la arquitectura final	107
Ilustración 105 Detalle <i>WebPanel</i> lanzado a traves de un teléfono inteligente	108
Ilustración 106 Detalle aplicación <i>Phi</i> lanzada desde Tablet	109
Ilustración 107 Detalle lanzamiento socket entrada y salida <i>Flightgear</i>	110
Ilustración 108 Detalle comunicación <i>Flightgear</i> y <i>APpanel</i>	110
Ilustración 109 Detalle datos enviados	110
Ilustración 110 Detalle <i>APpanel</i>	110
Ilustración 111 Detalle esquema de desarrollo <i>APpanel</i>	111
Ilustración 112 Detalle cabina y árbol de propiedades	112

Ilustración 113	Detalle del panel simulado	113
Ilustración 114	Detalle código <i>Kv</i>	114
Ilustración 115	Detalle definición puertos de entrada a <i>Flightgear</i>	115
Ilustración 116	Detalle archivo de configuración de <i>sockets</i> en <i>Flightgear</i>	115
Ilustración 117	Detalle código <i>Python</i>	116
Ilustración 118	Detalle inicialización	116
Ilustración 119	Detalle botón AP	116
Ilustración 120	Detalle función <code>app_enabled()</code> : encendido	117
Ilustración 121	Detalle función <code>app_enabled()</code> : apagado	118
Ilustración 122	Detalle botón HDG	118
Ilustración 123	Detalle de la función <code>hdg_enabled()</code> parte 1	119
Ilustración 124	Detalle de la función <code>hdg_enabled()</code> parte 2	120
Ilustración 125	Detalle botón ALT	120
Ilustración 126	Detalle de la función <code>alt_enabled()</code> parte 1	121
Ilustración 127	Detalle de la función <code>alt_enabled()</code> parte 2	122
Ilustración 128	Detalle de las funciones <code>up_enabled()</code> y <code>dn_enabled()</code>	122
Ilustración 129	Detalle botón BARO y selectores	122
Ilustración 130	Detalle función <code>baro_enabled()</code> y ejemplo botón de configuración	123
Ilustración 131	Detalle selectores simulados, OBS KNOB y HDG BUG	123
Ilustración 132	Detalle HSI y giro direccional respectivamente	124
Ilustración 133	Detalle funciones de configuración OBS KNOB y HDG BUG	124
Ilustración 134	Detalle selectores de frecuencias de comunicación y de navegación	125
Ilustración 135	Detalle funciones de configuración de las frecuencias	125
Ilustración 136	Detalle indicador DME, estimación de tiempo	126
Ilustración 137	Detalle indicador DME, estimación de velocidad	126
Ilustración 138	Detalle función <code>dme_enable()</code> y <code>dme_min2kt()</code>	126
Ilustración 139	Detalle función <code>update_label_dme()</code>	127
Ilustración 140	Detalle función auxiliar	127
Ilustración 141	Detalle función <code>udp_tx()</code>	128
Ilustración 142	Detalle envío de la información	129
Ilustración 143	Detalle cambios de rumbo	129
Ilustración 144	Detalle seleccionador de frecuencias	130
Ilustración 145	Detalle capturas diferentes radiales	130
Ilustración 146	Detalle capturas diferentes radiales	131
Ilustración 147	Detalle traza maniobra de aproximación	132
Ilustración 148	Detalle aproximación instrumental en <i>Flightgear</i>	132
Ilustración 149	Detalle traza de un hipódromo de espera	133
Ilustración 150	Detalle set-up final	133

1 MOTIVACIÓN E INTRODUCCIÓN

*Las ciencias tienen las raíces amargas, pero muy dulces
los frutos*

- Aristóteles -

Año a año los simuladores son cada vez más una rama más importante del mundo aeronáutico. Su versatilidad para reproducir eventos en vuelo hace que sea el medio más seguro y barato tanto para el entrenamiento de pilotos noveles en maniobras de alto riesgo o en la familiarización de los posibles fallos de las diferentes aeronaves, como para testear nuevos diseños, configuraciones y equipos antes de llevarlos al mundo real.

Dichas ventajas no solo hacen que su uso sea cada vez más extendido en el mundo empresarial y militar, sino que poco a poco estos aparatos empiezan a llegar a las universidades. Al igual que para el entrenamiento de pilotos, estas herramientas son perfectas para la docencia y el diseño, ya que permiten introducir la dinámica del vuelo y los sistemas embarcados desde un punto de vista más práctico y visual que teórico.

1.1. *Motivación del proyecto*

La motivación de este proyecto es la de sentar las bases del que sería el primer simulador de aeronaves colaborativo de la Universidad de Sevilla, con el objetivo de hacer posible que todo aquel que quiera o necesite disfrutar de un simulador pueda.

Los requerimientos fijados para ello son que dichas herramientas sean útiles tanto para una docencia más enfocada a la propia simulación del vuelo, como para el diseño tanto de sistemas o incluso de aeronaves. Todo esto además sumado a hacerlo posible con un presupuesto relativamente bajo.

Obviamente, este proyecto se puede categorizar de ambicioso, y posiblemente infinito, ya que siempre quedarían cosas por optimizar y posibles características a implementar, por lo que hay que fijar un límite. Este horizonte estaría dado por el desarrollo en un producto mínimo viable que, en este caso, se ha fijado como prototipo un simulador que permita un vuelo tanto manual como instrumental.

Para ello se han explorado muchas opciones que se abordarán durante esta memoria, pero siempre con la vista fijada en unos requerimientos bien claros:

Un simulador en código libre

Para poder no depender de factores externos y mejorar la viabilidad del proyecto, se ha decidido que este sea implementado en código libre. Esto además conlleva cierto valor añadido, ya que lenguajes como *Python* permiten al usuario utilizar librerías desarrolladas por terceros de forma fácil y sencilla, pudiéndose encontrar soluciones rápidas a problemas complejos. Esto ayuda al desarrollador a poder centrarse en construir el núcleo del proyecto sin tener que abordar problemas añadidos.

Un simulador modular

Para que pueda ser un proyecto escalable, es decir, que se pueda mejorar con el paso del tiempo, se ha decidido que este siga una estructura modular. Una vez construido un núcleo sólido que permita una simulación de alto nivel (que refleje al usuario la experiencia de vuelo), la idea es que pueda desarrollar módulos periféricos que refinen la calidad de la simulación. Esto va desde posibles modelos aerodinámicos o propulsivos más detallados y complejos, la implementación de las llamadas malfunctions (averías o fallos de los diferentes sistemas), como la simulación de sistemas de comunicaciones o incluso entidades externas.

Un simulador versátil

Para poder utilizar el simulador como una herramienta también enfocada al diseño, la información tiene que ser codificada de una forma intuitiva y sencilla. La idea es que la implementación de nuevas configuraciones se pueda hacer con un simple archivo de texto, donde estén recogidos los datos fundamentales de las aeronaves. Para el desarrollo de los paneles, se utilizarán librerías que hagan el trabajo abordable y divisible, pudiéndose planificar así los recursos y la extensión de un nuevo proyecto.

1.2. *Introducción histórica*

Antes de entrar con el proyecto en sí, se procederá a un pequeño repaso por la historia de los simuladores, siendo siempre paralela a la de la propia aviación.

Según la RAE la definición de simulador sería un aparato que reproduce el comportamiento de un sistema en determinadas condiciones, aplicado generalmente para el entrenamiento de quienes deben manejar dicho sistema.

Como se puede entender, los simuladores siempre han estado enfocados a reproducir la realidad para saber resolver los problemas antes de que estos ocurran en un entorno potencialmente peligroso.

Para una fidedigna reproducción del comportamiento de un avión, se debe caracterizar el sistema a través de las matemáticas. Estas sirven para emular el vuelo, las diferentes superficies de control, los diferentes sistemas embarcados y la reacción de las aeronaves ante factores externos como pueden ser turbulencias, viento cruzado, etc.

En los primeros años, esta caracterización como tal era rudimentaria, y los simuladores más que intentar reproducir fidedignamente un avión, estaban más centrados en ser un apoyo para la familiarización del piloto con los mandos de vuelo.

La idea de un entrenamiento previo se debe a la peligrosidad de la aviación en sus primeros años y la nueva coyuntura que suponían estos aparatos en el mundo militar, haciéndolos una pieza clave a principios de siglo.

Del primer modelo que se tiene constancia data de 1910, construido por la empresa francesa *Antoinette*, un simulador rustico el cual permitía familiarizarse con los mandos, mientras que los instructores ejecutaban perturbaciones externas, como se puede observar en la Ilustración 1.

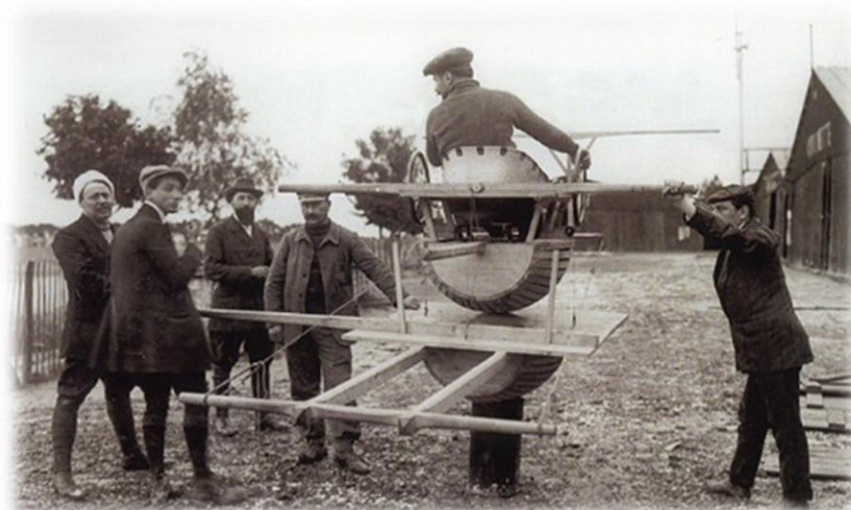


Ilustración 1 Simulador fabricado por la compañía Antoinette

No fue hasta 1927 con el *Link Trainer* (más conocido como *The Blue Box*) cuando se empezó a vislumbrar el concepto de simulador moderno que tenemos hoy en día. Dicho simulador estadounidense, de pequeño tamaño y fuselaje metálico, podía reproducir los movimientos de una aeronave a través de bombas y engranajes. Pero el motivo real de su éxito fue que disponía de unos instrumentos totalmente funcionales. En la Ilustración 2 se observa el entrenamiento de un piloto junto con su instructor.



Ilustración 2 Sesión de entrenamiento en un Link Trainer

Esta última característica no era habitual en aquellos años, y debido a ello, los pilotos noveles no solían estar familiarizados con dichos instrumentos, lo cual desencadenó en el escándalo del correo aéreo. Doce pilotos perecieron en un periodo de setenta y ocho días en Estados Unidos, debido a una inexperiencia en el vuelo instrumental. El ejército de los Estados Unidos de América tomaría entonces cartas en el asunto, y haciendo obligatorio la necesidad de un entrenamiento previo al vuelo real en simulador.

Con la llegada de la segunda guerra mundial, la fabricación de dichos simuladores se disparó, llegando a las 10.000 unidades, las cuales entrenarían a casi 500.000 nuevos pilotos para las naciones aliadas. Este nuevo fin, impulsó el desarrollo de nuevas funcionalidades, como por ejemplo la simulación de la esfera celeste para habituarse con los sextantes utilizados para la navegación astronómica.

Fue ya en la década de los cincuenta, con el desarrollo de la informática y el contexto de la guerra fría, cuando se desarrollaron los primeros simuladores que implementaban sistemas visuales (reproducción virtual del

entrono), sistemas de sonido y de movimiento (a través de conjuntos de actuadores). Habían nacido los Full Flight Simulators.



Ilustración 3 Bahía de simuladores del ITC, Sevilla

Sus grandes capacidades y realismo han permitido entrenar en condiciones seguras a miles y miles de pilotos. Las estimaciones últimas hechas por CAE, la empresa líder mundial en construcción de simuladores, fue que en el periodo de 2017 a 2027, se formarían gracias a dichas herramientas 255.000 nuevos pilotos de líneas y 180.000 primeros oficiales pasarían a capitanes.

Estos prácticamente han permanecidos invariables hasta el día de hoy, añadiendo nuevas funcionalidades (nuevas simulaciones como repostaje en vuelo) y cambiando sus estructuras internas (actualización de sistemas y mecanismos), pero no así de concepto.

Así mismo, cuando los ordenadores personales empezaron a ser de dominio público, el proyecto de un simulador amateur personal fue forjándose poco a poco. El primer simulador de esta índole llegó de la mano de Bruce Artwick, lanzando el *FSI Flight Simulator* para el Apple II en 1979. A lo largo del tiempo han surgido nuevos simuladores de esta índole, donde el más mundialmente conocido quizás sea el *Microsoft Flight Simulator*, por sus continuas innovaciones en el sector.

Estos softwares permiten el vuelo tanto manual como instrumental de la aeronave, y entre otras muchas funcionalidades, replican el desnivel del terreno mediante bases de datos, replican sistemas, embarcados, condiciones meteorológicas, radio-ayudas etc.

Gracias a que la potencia de las computadoras ha ido creciendo generación tras generación durante los años venideros, se han implementado mejoras cualitativas en el realismo de las simulaciones, tanto como la inclusión de tecnología 3D para la representación de inmensos escenarios fieles a la realidad, como la implementación de modelos de dinámica del vuelo más fieles a la realidad.

Esta última característica es el valor añadido ofrecido por el simulador *X-plane (1995-)*, el cual en vez de recurrir a datos empíricos recogidos en tablas (método bastante extendido por su ligereza) utiliza modelos aerodinámicos computacionales que estiman las fuerzas y momentos que aportan cada uno de los elementos que conforman la aeronave. En función de la potencia del ordenador anfitrión, se pueden ejecutar estos modelos en tiempo real o antes de lanzar la simulación con el fin de estimar a priori estas tablas aerodinámicas. Esto abre al gran público la capacidad de diseñar sus propios modelos de aeronave y de testarlos en vuelo simulado.

Y aunque ambos ejemplos anteriores estaban implementados en código cerrado debido a sus fines comerciales, la simulación aérea también llegó al código abierto con su máximo exponente siendo *Flightgear* a día de hoy.

Este simulador ha sido desarrollado desde 1996 por voluntarios gracias a la rápida expansión de internet. Este nació fruto de la falta de funcionalidades y el oscurantismo de los anteriormente nombrados simuladores comerciales, los cuales carecían de la posibilidad de expandir sus funcionalidades una vez cerrados y tampoco

permitían adentrarse en los sistemas y módulos que los conformaban.

Además, el hecho que este desarrollado por voluntarios ha hecho que esta herramienta haya sido acogida con entusiasmo por el mundo académico, siendo el apoyo software de miles de proyectos de investigación.

1.3. **Estructura teórica del simulador**

Evidentemente, a medida que la tecnología ha evolucionado, la forma de implementar dichos simuladores ha cambiado con esta. Con el fin de tener una idea general de cómo se estructura un simulador, se va a exponer un modelo de funcionamiento basado en EAAGLES (Extensible Architecture for the Analysis and Generation of Linked Simulation, que traducido libremente sería Arquitectura Extensible para el Análisis y Generación de Simulación Vinculada). Este es un marco de desarrollo expuesto por la fuerza aérea de los Estados Unidos de América.

Como marco, tiene como objetivo dar al desarrollador un esquema de funcionamiento global para crear unas aplicaciones de simulación robustas, escalables, modulares e independientes las unas de las otras. Esto se puede observar en la Ilustración 4, donde dicho marco esta resumido.

Se puede diferenciar en dos secciones, la estación y la simulación:

- Estación

La estación recoge todo aquello que sirve de enlace a los usuarios con el bucle de simulación.

Por una parte, están todo aquello que físicamente necesita el usuario, como son las pantallas para visualizar los instrumentos, como todo aquello que es necesario para gobernar la aeronave, ya sean palancas de control o botonería.

Es más, dentro de los simuladores, los instructores deben ser capaces de lanzar la simulación y de modificar el entorno y las condiciones de vuelo, por lo que en esta sección también incluiría dichos puestos de control. En ellos, más allá de interactuar con la aeronave se modifican valores internos, como la geolocalización o la caracterización del viento.

Para la comunicación entre los distintos sistemas, se debe definir el marco de trabajo que se va a utilizar. En la Ilustración 4 se muestra como ejemplo los marcos DIS (Distributed Interactive Simulation) y HLA (High Level Architecture), los cuales definen como se codifica, organiza y se envía la información y marcan las reglas del juego. Dichos marcos serán utilizados en función de la finalidad del entrenamiento.

Por ejemplo, un sistema que use el marco HLA que corra una simulación en la cual participen varias entidades, el tiempo de simulación total para cada una de las entidades quedara supeditada a aquella que sea más lenta entre todas ellas. En cambio, esto no sucede así para el marco DIS, el cual correrá siempre a un ritmo constante y si en algún caso una de las entidades tiene más carga computacional y no es capaz de simular a tiempo real, esto cause *stepping* (la entidad se quedará congelada a la espera de nuevos datos).

- Simulación

La simulación estaría más centrada en la emulación de la realidad que queremos representar. Forman parte de ella como se ha cuantificado el paso del tiempo, la base de datos geográfica (como se simula el mundo), el tiempo atmosférico o las distintas entidades.

Dentro de esta última clase, el modelo propuesto por OpenEagles es el de crear diferentes estructuras asociadas a cada una de las entidades, las cuales estarán compuestas a su vez por distintas funciones de forma jerárquica.

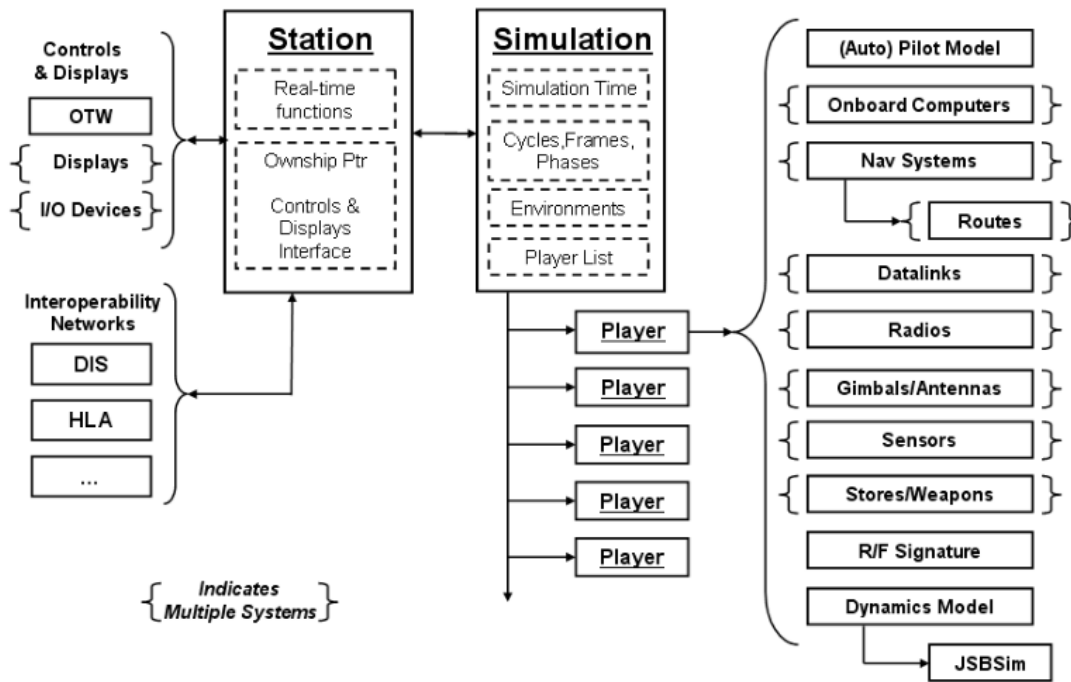


Ilustración 4 El entorno de simulación OpenEagles

Como se puede observar en la Ilustración 4, el número de estructuras asociadas a las entidades simuladas serán prácticamente las mismas para todos los casos (dentro de una misma familia de aeronaves), y la complejidad de cada una de ellas dependerá del objeto a simular.

Debido a que no todas las aeronaves no tienen la misma razón de ser, sus sistemas de navegación o equipamiento deberán estar acorde a ellos. Pero si existe algo que se computa de una forma relativamente similar (siempre teniendo en cuenta los efectos de compresibilidad) es el modelo dinámico, el cual computará la trayectoria de la aeronave en función de la estimación de las fuerzas.

Una de las grandes problemáticas de este tipo de modelos es la solución de compromiso que se tienen que adoptar para que el programa sea lo suficientemente potente para que la simulación sea realista, y lo suficientemente ligero para que la simulación no sea pesada. Esto suele desembocar a que estén construidos de la forma más eficientemente posible.

Además, este bloque suele ser núcleo más importante de la simulación, ya que no solo representa las ecuaciones cinemáticas y dinámicas de las aeronaves, en muchos casos el modelo propulsivo, modelos de actuadores, sistemas eléctricos, etc están embebidos en dicha estructura. Debido a ello, el resto de la simulación suele estar construida alrededor de los datos que aporta este módulo.

Para poder estar más familiarizados con las ecuaciones implementadas en dicho módulo, estas se van a introducir junto a las variables más importantes para la simulación física de un vuelo.

1.4. Estructura del documento

A modo de registro y desarrollo de todas las actividades realizadas durante este Trabajo de Fin de Master, este documento ha sido dividido en varias secciones principales, siendo estas divididas a su vez en subsecciones más específicas. Estas secciones y subsecciones son introducidas a continuación.

a) Motivación e introducción

En esta sección se han plasmado las motivaciones principales del proyecto además de una introducción histórica y teórica de los simuladores, siéndole recogido esto en las siguientes subsecciones.

- Motivación e introducción

En esta subsección se han plasmado tanto las motivaciones y objetivos del proyecto como las premisas a cumplir para el prototipo final.

- Introducción histórica

En esta subsección se ha desarrollado una breve introducción histórica a los inicios de la simulación, los diferentes problemas que propiciaron su desarrollo y la llegada de los simuladores al gran público.

- Estructura teórica del simulador

En esta subsección se ha introducido la estructura teórica del simulador, siéndole definidas las dos partes principales, la estación y la simulación.

- Estructura del documento

Esta misma sección, donde se hace un resumen del contenido del documento.

b) Ecuaciones integradas en los modelos dinámicos

En esta sección se ha introducido las partes más genéricas a la hora de implementar un modelo de dinámica del vuelo, siéndole introducidas las ecuaciones y participaciones de los distintos elementos en las siguientes subsecciones.

- Sistemas de referencia
- Velocidades angulares
- Velocidades lineales
- Ecuaciones del movimiento y rotación
- Modelo aerodinámico
- Modelo propulsivo

c) Exploración de viabilidad/opciones disponibles

En esta sección se han introducido los estudios y conclusiones para a cada una de las herramientas disponibles después de un estudio en profundidad, con el fin de ser utilizadas para el prototipo final. Dicho trabajo está dividido en las siguientes subsecciones.

- *Matlab/Simulink*

En esta subsección se ha desarrollado un modelo de dinámica de vuelo modular, haciéndose un testeo individualizado de cada una de las partes para su integración final mediante el desarrollo de un control por referencias tanto lateral como longitudinal.

- *JSBSim*

En esta subsección se ha hecho un estudio de la estructura general del modelo de dinámica de vuelo, como están desarrollados los diferentes archivos de configuración, para terminar con la implementación de un prototipo de simulador de vuelo en tiempo real en ley directa basado en *Python* y la utilización del mismo para test autónomos mediante la imposición de escenarios.

- *Flightgear*

En esta subsección se ha hecho un estudio de las diferentes funcionalidades ofrecidas por el simulador de vuelo *Flightgear*, donde se explora su estructura general y las herramientas auxiliares que ofrece para su utilización.

d) Desarrollo del simulador de vuelo

En esta sección se ha implementado la solución final para poder alcanzar los objetivos de este Trabajo de Fin de Master, basándose en las herramientas utilizadas en las secciones *JSBSim* y *Flightgear*.

- Arquitectura del simulador

En esta subsección se muestra la arquitectura final del simulador físico propuesto en este proyecto, y la relación de cada una de las partes para permitir las sesiones de simulación.

- Desarrollo APpanel

En esta subsección se ha desarrollado una herramienta APpanel, donde se ha implementado una aplicación basada en *Python* que reproduce visualmente y se implementan las funcionalidades del panel central de una Cessna 172p con el objetivo para completar el simulador.

- Resultados y conclusiones

En esta subsección se exponen los resultados de la herramienta final y se desarrolla una reflexión final sobre el simulador físico implementado.

e) Conclusiones

En esta sección se ha desarrollado una reflexión final del conjunto del proyecto.

2. ECUACIONES INTEGRADAS EN LOS MODELOS DINÁMICOS.

Una vez presentados el concepto de simulador aéreo y su estructura general, se van a introducir las ecuaciones básicas para la simulación, que no son más que las ecuaciones de la dinámica del vuelo. El objetivo de dichas ecuaciones son las de estimar las fuerzas y propagar la trayectoria de una aeronave a lo largo de la superficie terrestre.

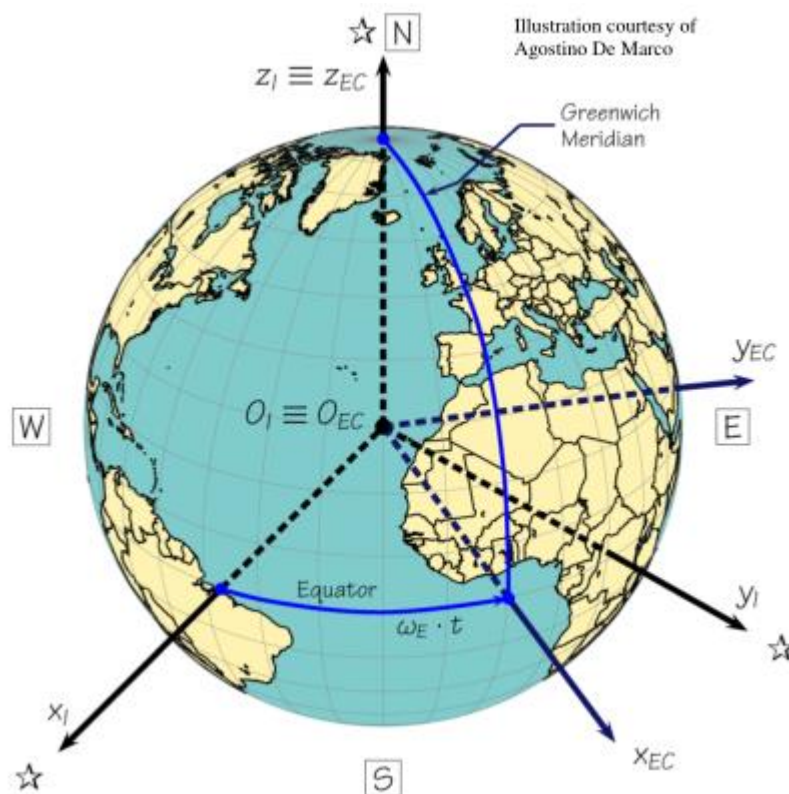
La notación que se utilizará en este documento seguirá la siguiente estructura:

- Los subíndices (por ejemplo $v_{CM/e}$), indica la relación de un objeto o sistema de referencia. Por ejemplo, en el ejemplo anterior nos estamos refiriendo a la velocidad del centro de masas respecto al sistema de referencia ECEF.
- Los superíndices (por ejemplo v^b) indicaran el sistema de referencia en el cual están expresadas las variables, como por ejemplo los ejes cuerpo en el ejemplo anterior.

2.1. Sistemas de referencia

Los sistemas necesarios para caracterizar el movimiento de una aeronave a lo largo del mundo son los siguientes.

- Sistema de referencia NED: v o I
El sistema de referencia NED (North, East, Down), con el origen centrado en el centro de masas del vehículo, con el eje X apuntando al norte, el eje Y apuntando al este y el eje Z como producto vectorial de las dos anteriores.



- Sistema de referencia ejes cuerpo: b

El sistema de referencia fijado en el cuerpo, el origen estaría centrado en el centro de masas del vehículo, con el eje X orientado hacia la nariz del avión, el eje Y apuntando hacia la parte derecha del avión y el eje Z apuntando hacia abajo.

- Sistema de referencia ejes viento: w

También denominado como ejes navegación, el origen estaría centrado en el centro de masas del vehículo, con el eje X según la dirección del viento, el eje Y situado en el plano horizontal, perpendicular al eje X y según a el ala derecha del avión, y el eje Z completando el triedro.

- Sistema de referencia ECEF: e

El sistema de referencia centrado en la tierra ECEF, con el eje Z orientado con el eje de rotación de la Tierra y positivo hacia el norte, el eje X positivo apuntando al punto de latitud y longitud cero. Este sistema gira respecto a la tierra y no se traslada. Véase Ilustración 5.

- Sistema de referencia ECI: i

El sistema inercial centrado en la tierra ECI, con el eje Z orientado con el eje de rotación de la Tierra y positivo hacia el norte, y el eje X e Y contenido en el plano ecuatorial. Dicho eje de referencia gira con la tierra, coincidiendo en el instante inicial con el sistema ECEF. Véase Ilustración 5.

2.2. Velocidades angulares

Para poder caracterizar los movimientos entre los distintos sistemas de referencia necesitaremos definir las contribuciones en velocidad angular.

- $\omega_{b/v}$
Velocidad angular que relaciona el sistema de referencia cuerpo con el sistema de referencia NED. Cuando el vehículo no esté maniobrando, su valor será nulo.
- $\omega_{b/e}$
Velocidad angular que relaciona el sistema de referencia cuerpo con el sistema de referencia ECEF. Cuando el vehículo este en reposo en tierra su valor será nulo. Usualmente se medirá respecto ejes cuerpo, siendo $\omega_{b/e}^b$ y cuyas componentes longitudinal, lateral y vertical son denominadas como p , q y r , respectivamente.
- $\omega_{b/i}$
Velocidad angular que relaciona el sistema de referencia cuerpo con el sistema de referencia ECI.
- $\omega_{v/e}$
Velocidad angular que relaciona el sistema de referencia NED con el sistema de referencia ECEF. Téngase en cuenta que este dependerá de la velocidad relativa del vehículo respecto el sistema de referencia ECEF, no dependiendo así de su actitud.
- $\omega_{e/i}$
Velocidad angular que relaciona el sistema de referencia ECEF respecto el sistema referencia ECI. Básicamente es la velocidad de rotación de la Tierra.

Posteriormente en las ecuaciones se verán productos vectoriales entre los vectores velocidad angular. Se podrán encontrar tanto en su notación vectorial como en su notación matricial, la cual está representada como se puede observar posteriormente.

$$\Omega_{b/i} = \begin{bmatrix} 0 & -r & q \\ r & 0 & -p \\ -q & p & 0 \end{bmatrix} \quad (1)$$

2.3. Velocidades lineales

Al igual que para las velocidades angulares, es necesario caracterizar el movimiento relativo del objeto a seguir mediante las distintas velocidades.

- $\mathbf{v}_{b/e}^b$

Velocidad en ejes cuerpo que mide la velocidad del vehículo respecto el sistema de referencia ECEF. Sus componentes longitudinal, lateral y vertical son denominadas como u , v y w respectivamente. Velocidad nula si aeronave en reposo y viento inexistente.

- $\mathbf{v}_{b/e}^v$

Velocidad medida en sistema de referencia NED, que mide la velocidad del vehículo respecto el sistema de referencia ECEF.

2.4. Ecuaciones del movimiento y rotación

En este apartado plantearemos las ecuaciones básicas del movimiento para un sólido rígido.

Las ecuaciones de Newton son un conjunto de ecuaciones que relacionan las fuerzas que se ejercen en un objeto y el movimiento que este sigue. Estas ecuaciones están mostradas a continuación, dadas utilizando la notación matricial del producto vectorial.

$$\dot{\mathbf{v}}^b = \frac{\mathbf{F}_{A,T}^b}{m} - (\Omega_{b/i}^b + \Omega_{e/i}^b)\mathbf{v}^b + \mathbf{C}_i^b \mathbf{g}^i \quad (2)$$

Donde $\mathbf{F}_{A,T}^b$ son las fuerzas tanto propulsiva como aerodinámicas y de reacción (incluye las fuerzas debidas al contacto), \mathbf{g}^i es la fuerza de la gravedad en ejes inerciales y \mathbf{C}_i^b es la matriz de cambio de ejes para poder relacionar esta fuerza en los ejes correspondientes. Se observa como quedaría dicha ecuación utilizando la notación vectorial en la posterior ecuación número 3.

$$\dot{\mathbf{v}}^b = \frac{\mathbf{F}_{A,T}^b}{m} - (\boldsymbol{\omega}_{b/i}^b + \boldsymbol{\omega}_{e/i}^b) \times \mathbf{v}^b + \mathbf{C}_i^b \mathbf{g}^i \quad (3)$$

Al igual que existen las ecuaciones de Newton, para poder relacionar los momentos angulares con las velocidades angulares, existen las ecuaciones de Euler. En este caso, se tomará como eje de referencia el eje cuerpo, ya que el hecho de que la matriz de inercia tenga los elementos constantes facilita el cálculo y la posterior computación.

$$I^b \dot{\boldsymbol{\omega}}^b + \boldsymbol{\omega}^B \times I^b \boldsymbol{\omega}^B = \mathbf{M}^b \quad (4)$$

Donde I^b es la matriz de inercia en ejes cuerpo y donde \mathbf{M}^b son los momentos aerodinámicos y propulsivos.

2.5. Modelo aerodinámico

Están son las fuerzas y momentos que son generados por una corriente de aire que fluye alrededor de un objeto. Matemáticamente, se pueden estimar dichas fuerzas y momentos aerodinámicos mediante la integral de la presión menos la presión en el infinito sobre la superficie.

Estas fuerzas y momentos suelen estar nombrados con diferentes símbolos en función de los sistemas de referencia que utilizemos. En este caso, se utilizará el sistema de ejes viento para describirlas, utilizándose los siguientes símbolos para denominarlas:

- L : Fuerza de sustentación, conocida en la literatura anglosajona como *Lift*, definida como positiva en el sentido negativo del eje z
- D : Resistencia aerodinámica, conocida en la literatura anglosajona como *Drag*, definida como positiva en el sentido negativo del eje x
- Y : Fuerza lateral, definida positiva en el sentido positivo del eje y

- l : Momento de balance, conocido en la literatura anglosajona como *Roll moment*, definida como positiva en el sentido positivo del eje x
- m : Momento de cabeceo, conocido también como *Pitch moment*, definida como positiva en el sentido positivo del eje y
- n : Momento de guiñada, conocido también como *Yaw moment*, definida como positiva en el sentido positivo del eje z

Este cálculo no es directo, ya que para este se toman hipótesis sobre el flujo a diferentes niveles de profundidad en función de diferentes criterios. Después de esto se desarrollarían las ecuaciones y se integrarían usando distintos métodos, que para la mayoría de los simuladores supondría una carga computacional excesiva y se suele descartar.

El método por el que se opta mayormente en una caracterización de estas fuerzas y momentos mediante coeficientes, debido a su eficiencia computacional y su fiabilidad, teniendo siempre en cuenta sus evidentes limitaciones. Este método consiste en la suma de todas las contribuciones que toman parte en la fuerza o momento.

En este proyecto se utilizarán dichos modelos de coeficientes, dando lugar a la siguiente caracterización de fuerzas y momentos:

$$L = \frac{1}{2} \rho v^2 S C_L \quad (5)$$

$$D = \frac{1}{2} \rho v^2 S C_D \quad (6)$$

$$Y = \frac{1}{2} \rho v^2 S C_Y \quad (7)$$

$$l = \frac{1}{2} \rho v^2 S b C_l \quad (8)$$

$$m = \frac{1}{2} \rho v^2 S c C_m \quad (9)$$

$$n = \frac{1}{2} \rho v^2 S b C_n \quad (10)$$

En las ecuaciones anteriores se observan términos como la presión dinámica $\frac{1}{2} \rho v^2$, la superficie mojada S , una superficie característica b (en este caso la envergadura) y el correspondiente coeficiente adimensional.

Dichos coeficientes adimensionales son función de parámetros tales como:

- La geometría del avión (fija): alargamiento, diedro, etc
- Deflexión de las superficies aerodinámicas: alerones, timón de profundidad, timón de dirección, flaps, etc
- Ángulos del flujo: Alpha, beta, etc
- Número de Reynolds

Estos coeficientes son formados a su vez por diferentes contribuciones, en función de una variable característica representativa. No todas estas fuerzas y momentos son dependientes de las mismas variables de la misma forma, siendo alguna de ellas más dominantes en según qué fuerza o eje. Como ejemplo, se va a tomar los coeficientes de referencia del documento [2] *Stability and control derivative estimates obtained from flight data for the Beech 99 aircraft*.

Estos quedarían de la siguiente forma:

- Coeficientes longitudinales

$$C_N = C_{N_\alpha} \alpha + C_{N_{\delta_e}} \delta_e + C_{N_0} \quad (11)$$

$$C_m = C_{m_\alpha} \alpha + C_{m_{\delta_e}} \delta_e + C_{m_q} \frac{q\bar{c}}{2V} + C_{m_0} \quad (12)$$

$$C_D = C_{D_0} + KC_L^2 \quad (13)$$

- Coeficientes laterales

$$C_Y = C_{Y_\beta} \beta + C_{Y_{\delta_a}} \delta_a + C_{Y_{\delta_r}} \delta_r + C_{Y_0} \quad (14)$$

$$C_l = C_{l_\beta} \beta + C_{l_p} \frac{pb}{2V} + C_{l_r} \frac{rb}{2V} + C_{l_{\delta_a}} \delta_a + C_{l_{\delta_r}} \delta_r + C_{l_0} \quad (15)$$

$$C_n = C_{n_\beta} \beta + C_{n_p} \frac{pb}{2V} + C_{n_r} \frac{rb}{2V} + C_{n_{\delta_a}} \delta_a + C_{n_{\delta_r}} \delta_r + C_{n_0} \quad (16)$$

Los símbolos p , q y r hacen referencia a las velocidades angulares en los ejes x , y y z , los símbolos δ_a , δ_e y δ_r hacen referencia a las deflexiones de los alerones, al timón de profundidad y al timón de dirección, los símbolos l , n y m hacen referencia a los momentos sobre los ejes x , y y z , y finalmente los símbolos α y β hacen referencia al ángulo de ataque y al ángulo de deslizamiento

En esta descripción no está el coeficiente de sustentación, que ha sido sustituido por el coeficiente normal. Para un α pequeño, ambos coeficientes son muy similares y sirve para ejemplificar como se caracterizan dichos coeficientes.

Cada uno de estos sub coeficientes pueden depender a su vez de otras variables, o incluso estar tabulados fruto de ensayos empíricos. Muchas veces estos coeficientes se dan como constantes, haciendo que el modelo sea únicamente fiable alrededor del punto de linealización.

También este tipo de modelos suele ser poco fiable en la pérdida. Estos suelen ser dados por los fabricantes, muchas veces estimados de pruebas empíricas o ensayos en túneles. Para todos aquellos que no disponen de dichos medios, estos se pueden estimar mediante métodos numéricos.

2.6. Modelo propulsivo

El modelo propulsivo es quizás el sistema del cual sea más difícil la obtención de datos libremente, ya que simularlos termodinámicamente en tiempo real supone un coste computacional excesivo y en la industria se suelen requerir a tabulaciones.

Además, existen diferentes tipos de motores que pueden ser simulados, como son los motores de pistón con hélice, los motores cohetes, las turbinas, turbofanos o incluso eléctricos, cuyas caracterizaciones son todas diferentes. Esto implica que dichas caracterizaciones dependen de una serie de variables diferentes en función de su propia naturaleza.

Por otro lado, para hacer una simulación realista, se tienen que tener en cuenta el número de motores y su localización exacta dentro de la geometría de la aeronave, con el fin de estimar más fielmente los momentos.

Aunque esto último no es complicado, la caracterización de cada uno de los motores es algo bastante restrictivo a la hora de generar contenido propio, ya que a diferencia de los coeficientes los cuales (casi todos) se pueden estimar numéricamente de una forma sencilla, para los motores se suelen recurrir a dichas tabulaciones con correcciones en altitud proporcionadas por el fabricante. Estos datos empíricos suelen ser obtenidos gracias a ensayos en banco.

Por ello, obtener en el mundo docente estos datos es algo bastante complicado, y se suele recurrir a documentación especializada o descatalogada (caracterizaciones de aviones antiguos).

3. EXPLORACIÓN DE VIABILIDAD/OPCIONES DISPONIBLES

Una vez estudiadas las bases de la simulación dinámica, el proyecto pasaba a un punto crítico, el de empezar a desarrollar un prototipo. Y antes de dar este paso en falso, se decidió hacer un estudio con el fin de testear las diferentes plataformas para comprobar cuál de ellas sería la mejor para su desarrollo.

Lejos de ser trivial, esta es una decisión compleja, ya que dependiendo de la plataforma seleccionada se obtendrán unas ventajas u otras, como pueden ser la facilidad para la programación o la eficiencia del código, marcando profundamente el rumbo y el alcance del proyecto. Por ello, esto suele ser un proceso iterativo, que consume bastante tiempo, donde la respuesta nunca suele ser evidente, y se obtienen respuestas más propiamente de la experiencia.

Barajando las posibles opciones, se llegó a la conclusión de que había tres herramientas aptas para este prototipo: *Matlab/Simulink*, *C++* y *Python*.

Matlab es una plataforma de programación diseñada para analizar y diseñar sistemas y productos, basados en un lenguaje basado en matrices. *Simulink* es una extensión gráfica de *Matlab* para el modelado y simulación de los sistemas, con la particularidad de poder simular sistemas no lineares.

C++ es un lenguaje de programación de alto nivel, nacido de *C*. Aunque este nació como un lenguaje orientado a objetos, posteriormente se añadieron facilidades de programación genérica, siendo así un lenguaje de programación multiparadigma actualmente. Su principal característica es que es un lenguaje de programación compilado, siendo necesaria una conversión a lenguaje máquina antes de ser ejecutado, generando un archivo ejecutable que se puede ser lanzado desde la consola del ordenador. La principal virtud de estos lenguajes compilados es que son muy eficientes a la hora de ser ejecutados, perfectos para productos maduros.

Python es también un lenguaje de programación de alto nivel y orientado a objetos, con la diferencia que este es un lenguaje interpretado. A diferencia del compilado, esta conversión del código a lenguaje máquina se hace a medida que este va siendo ejecutado, haciendo la escritura y depuración del código más amigable.

Matlab/Simulink se antojaba la más sencilla de utilizar a priori, ya que la interfaz *Simulink* lleva la programación a un alto nivel, pero contaría con las desventajas de no ser código libre y no poder asegurar la ejecución de los modelos en tiempo real a medida que se suma complejidad en estos. De manera contraria, *C++* se antojaba la solución más eficiente de todas, pero a la par la que mayor conocimiento sobre el lenguaje de programación requería y la que más esfuerzo llevaría. Y, por último, como una solución intermedia estaría *Python*, un lenguaje más amigable y versátil por su propia construcción, que además cuenta con multitud de soluciones software ya implementadas gracias a su extensísima comunidad de usuarios.

Entre todas estas, y al tener que iniciar el proceso de desarrollo, se optó por ir de herramienta más intuitiva a herramienta más compleja, con tal de dar un desarrollo coherente, y mejorar la viabilidad del proyecto. Por ello se inició la implementación de un prototipo en *Matlab/Simulink*.

3.1. *Matlab/Simulink*

Matlab/Simulink es una herramienta ampliamente usada en las carreras de índole ingenieril de la Universidad de Sevilla. Como herramienta de cálculo numérico, permite generar funciones personalizadas y cuenta con una gran librería de estas dadas por el fabricante, las cuales pueden ser usadas por otras funciones, dando así una gran flexibilidad y permite reutilizar trabajo funcional ya implementado.

Embebido en dicho programa, se encuentra el módulo *Simulink*, el cual, mediante una interfaz gráfica, lo cual permite generar código a alto nivel. El resumen general sería el de un diagrama de control, en el cual se tienen unos bloques que realizan ciertas operaciones a las variables que fluyen por los buses de datos. Esto se observará más adelante en este propio apartado en las diferentes ilustraciones.

Esta manera de organizar el código por bloques visuales hace que sus partes estén bien diferenciadas y la

trazabilidad de los datos se puede antojar sencilla, ya que la relación de dos bloques esta resumida visualmente una línea o un marcador que tiene inicio y final.

Analizando las ventajas de esta plataforma, y siguiendo las premisas de este TFM, se decide crear una librería de funciones con el objetivo de dar los bloques mínimos necesarios para simular el vuelo de una aeronave. Así, se podría estructurar el trabajo y ampliarse fácilmente.

El patrón de trabajo que se siguió fue el de generar una función con un cierto objetivo, testar dichos módulos para tener certeza de que su funcionamiento es el correcto y seguir así hasta obtener un producto mínimo viable, suma de todas las pequeñas contribuciones, y testar su funcionamiento conjunto.

Si esto se consiguiera, se podrían crear nuevos núcleos alrededor del núcleo principal original, los cuales pueda añadir nuevas funcionalidades y profundidad al proyecto, además de conseguir hacer de este un proyecto escalable, capaz de ser mejorado en el tiempo por distintos individuos.

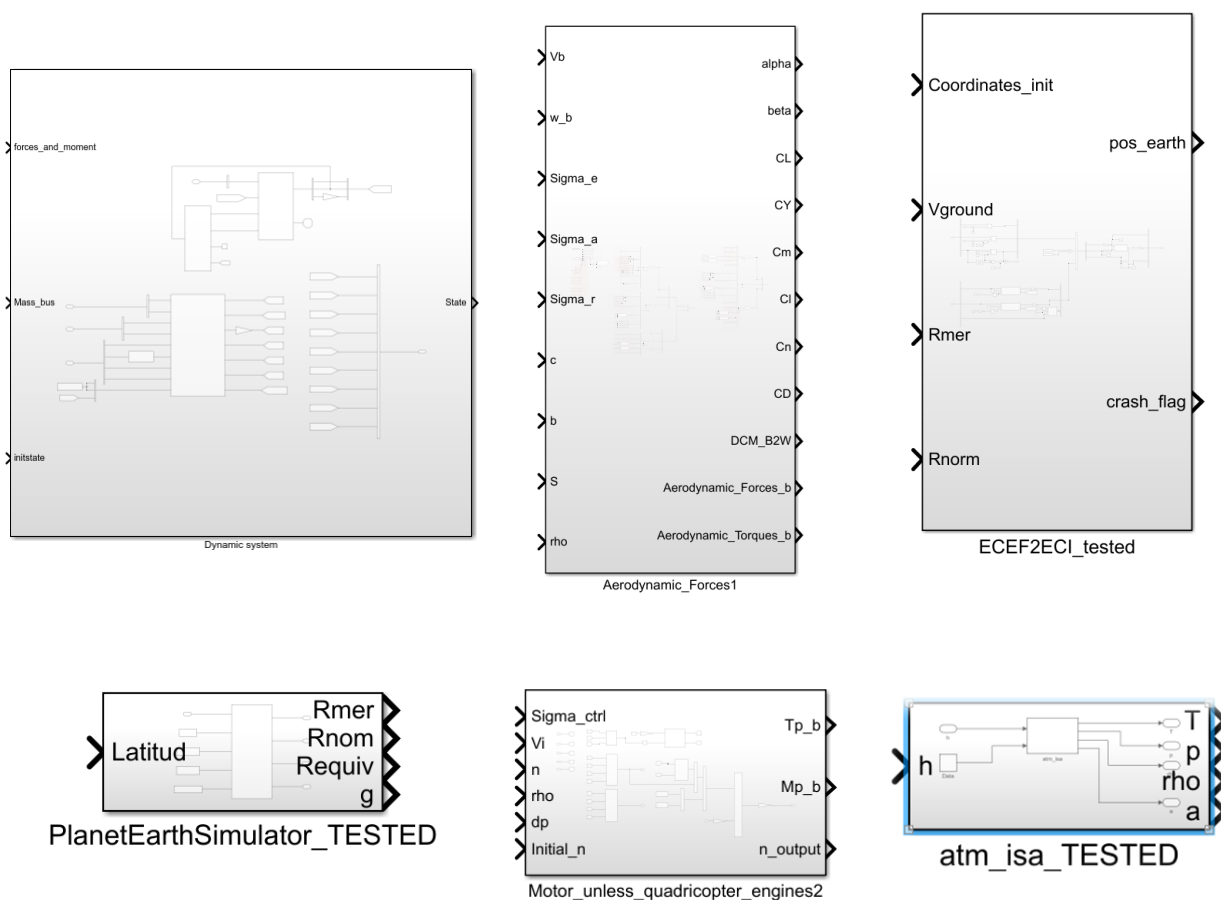


Ilustración 6 Módulos de simulación

En la Ilustración 6 se puede observar los distintos módulos de simulación utilizados durante esta iteración del desarrollo.

Estructurando el sistema, se decidió separar la información por buses de datos, para que sea sencillo saber a cuál acudir cuando se necesite alimentar un nuevo módulo y reducir de dicha forma el número de conexiones, muchas veces molestas para el desarrollo.

A continuación, se describirán de forma breve cada uno de los buses principales:

- El bus de estados, donde quedan recogidos la localización en ejes ECEF, la velocidad NED, ángulos de Euler...
- El bus de fuerzas y momentos, donde quedan recogidos fuerzas y momentos aerodinámicos, propulsivos, de contacto...

- El bus de masas donde quedan recogidos la masa, la inercia, el centro de gravedad...
- El bus aerodinámico donde están recogidos ángulo de ataque, de resbalamiento...
- El bus de control donde quedan recogidos la posición de la palanca de gases, la orden de deflexión en alerones, timón de profundidad, timón de dirección...
- El bus de inicialización el cual será el encargado de obtener los valores iniciales de todas las variables que se vayan a integrar...

Quedarían pendientes de implementar buses como los de navegación, los hidráulicos, neumáticos, los cuales quedarían fuera de los objetivos de este primer mínimo producto viable.

A continuación, se introducirán los diferentes módulos que se llegaron a implementar.

3.1.1. Descripción de los distintos módulos

3.1.1.1. Bloque integrador

Todo el modelo dinámico gira alrededor de este bloque. Es el encargado de la integración numérica de las ecuaciones de Newton y Euler (ecuaciones 3 y 4 respectivamente), cambiar de coordenadas NED a coordenadas ECEF y de actualizar el bus de estados. Esta integración se puede hacer tanto en ejes cuerpo como en ejes NED, pero el hecho de utilizar los primeros da la ventaja de que la matriz de inercia es invariable, cosa que resulta muy útil matemáticamente. Por ello, se tomó como referencia ejes cuerpo en el desarrollo de este bloque.

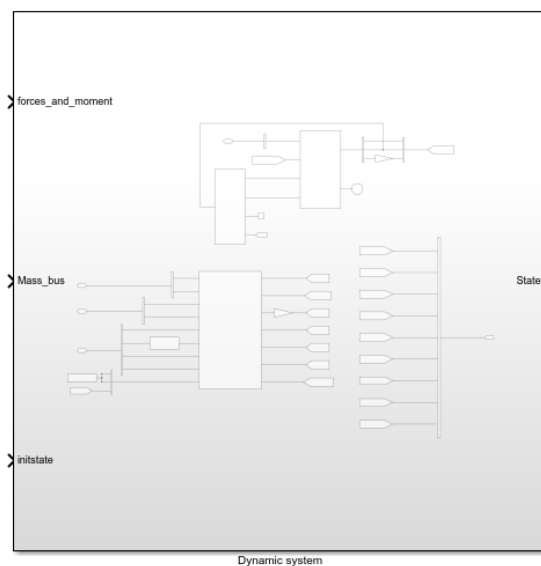


Ilustración 7 Bloque integrador

Dentro de este bloque se pueden diferenciar dos procesos diferentes, el primero es la integración de las velocidades y velocidades angulares mediante las fuerzas y momentos, y el segundo que hace el cambio de base y calcula las coordenadas ECEF.

Este primer bloque se alimenta del bus de fuerzas y momentos, el bus de datos de masas, y el bus de instante inicial. Dentro de este proceso, la integración de ambas ecuaciones está claramente diferenciada, y solo son retroalimentadas en el caso de la estimación de los ángulos de Euler, los cuales son necesarios en cada paso de iteración para la estimación de la gravedad.

Dentro del proceso de las ecuaciones de Euler, hay dos formas de hacer la integración numérica: mediante cálculo matricial o con cuaterniones. Ambos tienen ventajas e inconvenientes, básicamente el cálculo matricial es más intuitivo, pero en diferentes puntos puede dar discontinuidades, cosa que con los cuaterniones no sucede.

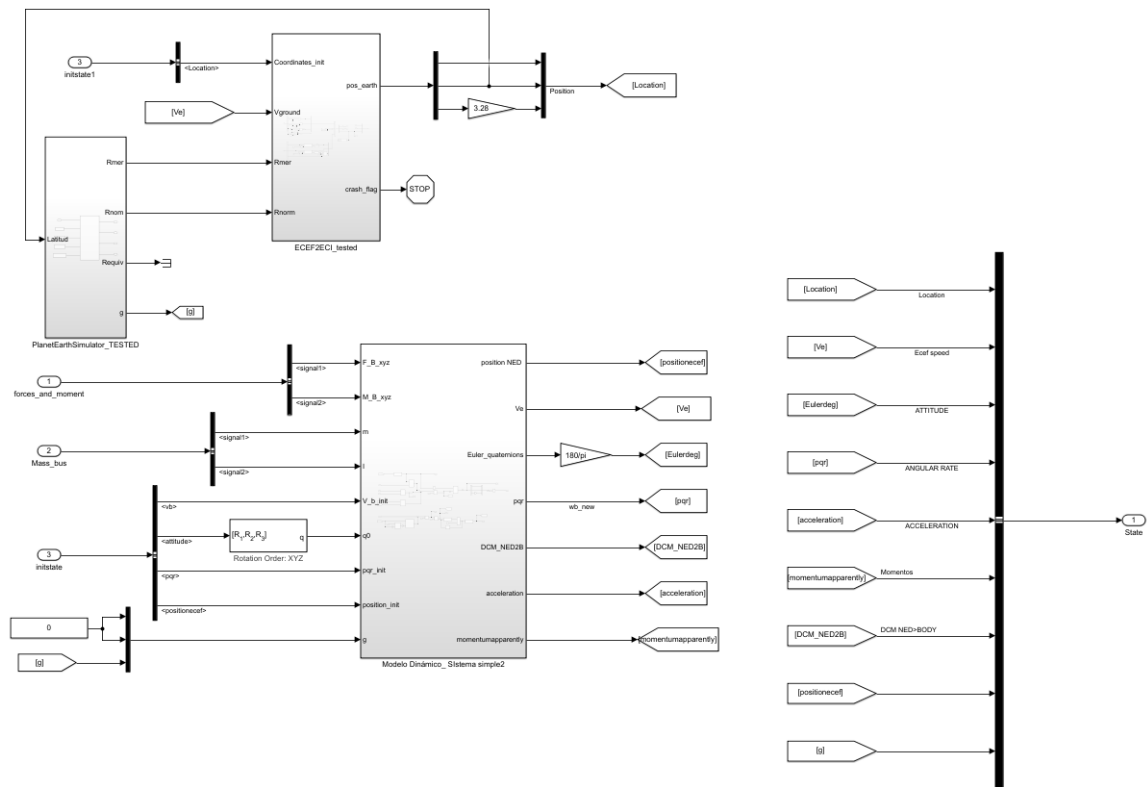


Ilustración 8 Detalle bloque integrador

En esta primera iteración del proceso se calcularon de las dos formas, para comparar el error entre las dos estimaciones, aunque en la versión final se optó finalmente por los cuaterniones, y su posterior transformación a ángulos de Euler. Esto se puede observar en la Ilustración 9.

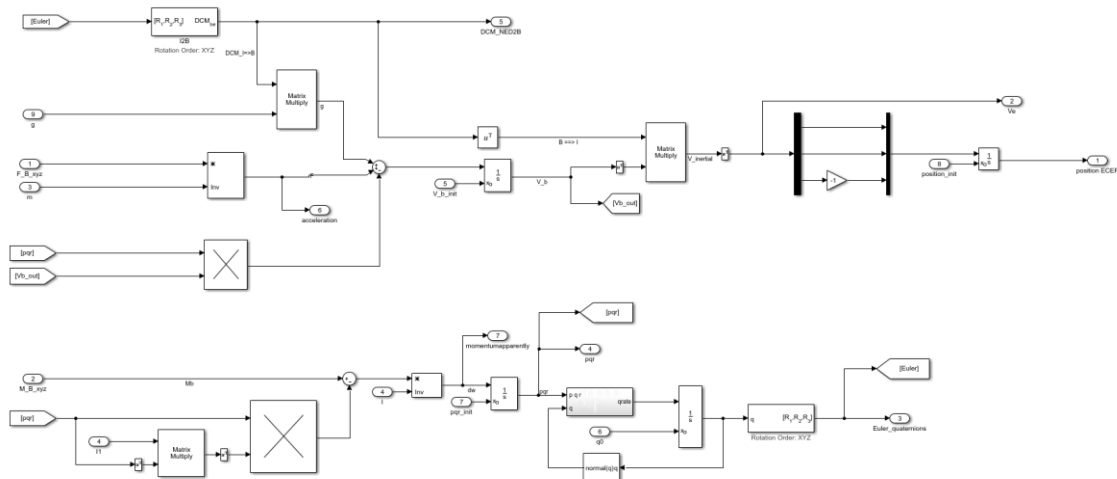


Ilustración 9 Detalle de la codificación de las ecuaciones dinámicas

El segundo proceso que se observa dentro de este bloque integrador es el módulo dedicado a la obtención de las coordenadas en el sistema de referencia ECEF a partir de las velocidades en sistema de referencia NED. Este está formado por dos procesos diferentes, los cuales serán introducidos más posteriormente.

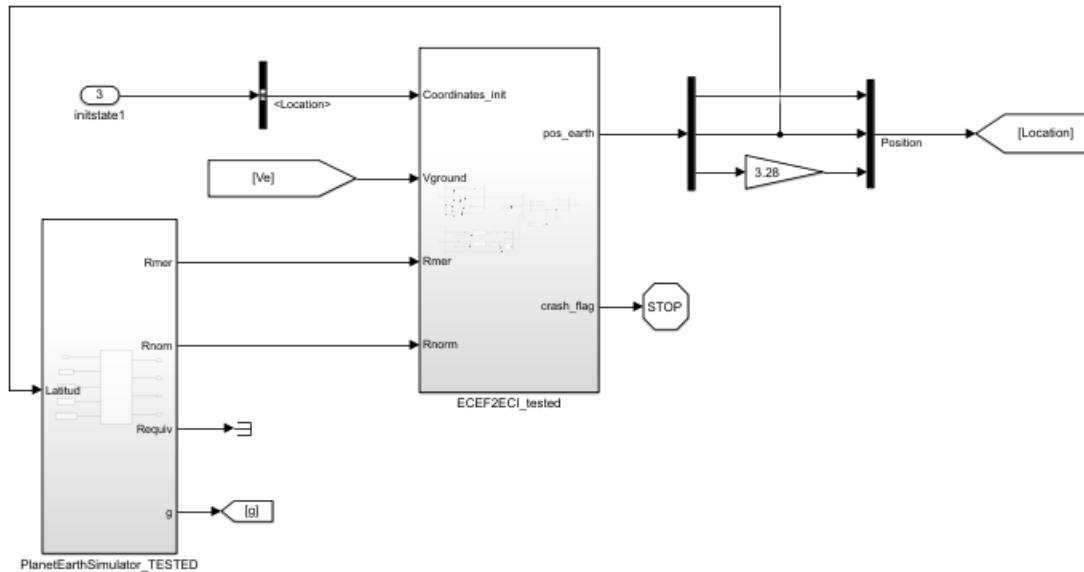


Ilustración 10 Detalle obtención coordenadas ejes ECEF

Se puede comentar que dentro de este bloque tenemos la bandera del impacto contra el suelo, el cual simplemente compara la velocidad z de la aeronave cuando la altura de la aeronave pasa a valer cero con un valor límite estructural, parando la simulación en el caso de que esta sea mayor.

3.1.1.2. Bloque aerodinámico

Este bloque es el encargado de calcular las fuerzas y momentos aerodinámicos, además de los ángulos de incidencia α y β (también conocidos como ángulo de ataque y de resbalamiento). Como se puede deducir, será el módulo encargado de alimentar el bus aerodinámico y contribuirá al bus de fuerzas y momentos (del cual también participan las fuerzas propulsivas y de reacción).

Se estimarán las fuerzas y momentos aerodinámicos mediante variables de estado y de control, características aerodinámicas, características físicas de la aeronave (como puede ser la superficie mojada o la cuerda), y una serie de coeficientes.

En esta primera iteración se tomó un avión de referencia desde el cual trabajar, en este caso el dron *Skywalker X8*, para hacer la caracterización. Se escogió esta aeronave debido a la gran cantidad de datos que están disponibles en la red.

Todas las entradas necesarias para esta estimación se pueden ver a la izquierda del diagrama mostrado en la Ilustración 11. Estas entradas pueden ser ampliadas a posterioridad, dependiendo esto del modelo aerodinámico. Cuanto más completo y mejor refleje la realidad, la cantidad potencial de entradas necesarias para el modelo será mayor.

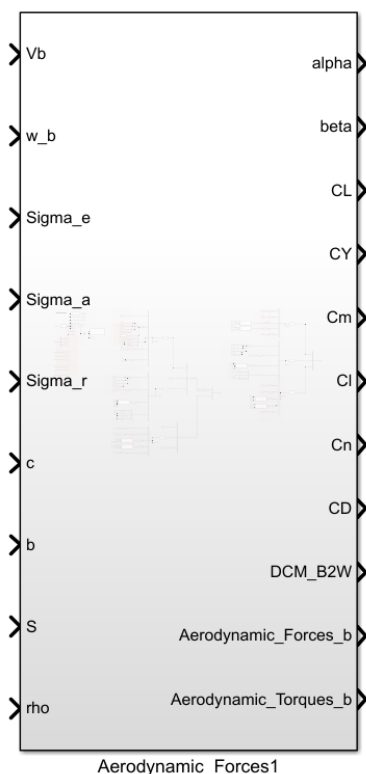


Ilustración 11 Bloque aerodinámico

Al igual que las entradas, se observan las salidas, en las cuales se identifican tanto como los ángulos de incidencia del viento, la matriz de cosenos directores para cambiar de base entre ejes cuerpo y ejes viento, un desglose de los diferentes coeficientes de cada una de las fuerzas y momentos, y finalmente los vectores de fuerzas y momentos aerodinámicos en ejes cuerpo.

Estos coeficientes pueden depender de una serie de variables externas, pero como primera aproximación se toman unos valores fijos para poder desarrollar este prototipo. Siendo así, estas fuerzas y momentos no serían más que un polinomio el cual se puede codificar mediante un diagrama de bloques, como se observa en la Ilustración 13.

Se observa tres partes bien diferenciadas dentro de este bloque, que son el bloque que computa los ángulos de incidencia, el bloque que calcula las fuerzas aerodinámicas y el que calcula los momentos. Se pueden observar sendos bloques en la Ilustración 12 e Ilustración 13.

Además de lo anteriormente comentado, para precargar los coeficientes aerodinámicos se utilizó un pulsador, el cual se observa en el borde superior izquierdo de la Ilustración 12, en el que estaban recogidos estos y desde el cual se modificaban.

En dicha imagen se puede ver un sub bloque encargado de calcular la función sigmoide, la función encargada de simular la entrada en pérdida. Dicha función depende solamente del ángulo de ataque y del ángulo teórico o experimental de entrada en pérdida. A grandes rasgos, genera una región donde el comportamiento de los coeficientes se mantiene lineal, y una segunda donde estos pierden esta característica, desplomándose y así simula dicha pérdida.

En la Ilustración 13 se observa cómo se ha codificado visualmente la función encargada de estimar la fuerza propulsiva, donde queda patente todas las variables, métrica, funciones y coeficientes que entran en juego.

3.1.1.3. Bloque propulsivo

Este es el bloque encargado de estimar las fuerzas y los momentos propulsivos. En esta ocasión, estando acorde con cómo funciona el bloque integrador, se darán los vectores en ejes cuerpo.

Aquí entra una de las grandes interrogantes de este prototipo, ya que el modelo propulsivo es de los sistemas más complejos y costosos computacionalmente a la hora de simular, y de los que menos datos de dominio público hay.

Además de esto anterior, existen varios tipos de motores con sus correspondientes modelos matemáticos asociados, por lo que una simulación estándar única para todos se antoja complicada y se deberían construir varios bloques para cada tipo de modelo y para cada tipo de modelaje.

En esta primera iteración, se va a simular un motor de hélice acoplado a un motor eléctrico. Para ello se utilizarán los llamados coeficientes de potencia y coeficientes propulsivos para simular el efecto de la hélice, y para simular el motor eléctrico se utilizará un modelo simplificado de circuito eléctrico.

Los coeficientes de potencia y propulsivos son datos experimentales usualmente dados por el fabricante. Para más información, se entrará en detalle posteriormente en el apartado [3.2.2.6](#).

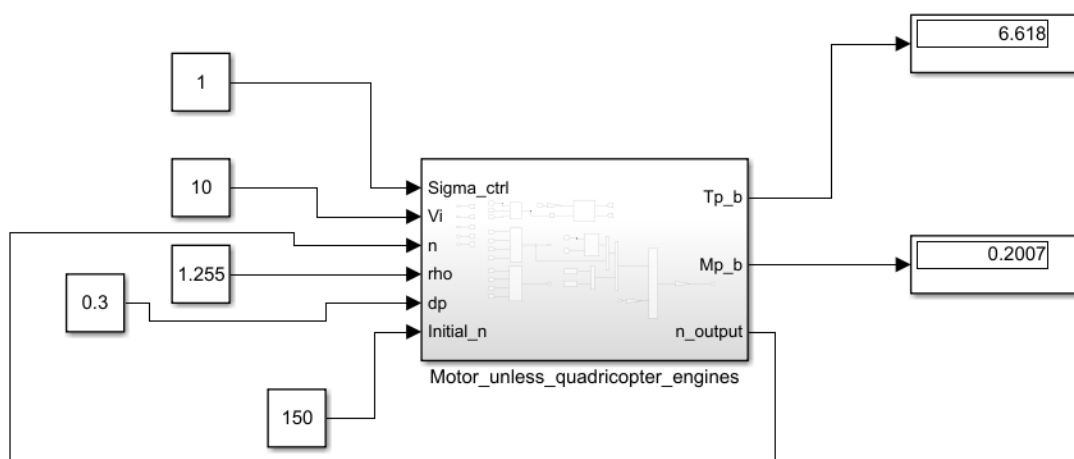


Ilustración 14 Motor eléctrico acoplado a hélice

En la Ilustración 14 se observa las entradas y salidas de este bloque. Entre las entradas, se puede observar la variable de control nombrada como *Sigma_ctrl*, la cual modela mediante una variable contenida entre el 0 y el 1 el mínimo o el máximo voltaje que se le aplica a la entrada del circuito eléctrico.

Además, el conjunto de las entradas lo completan una serie de características del vuelo como la velocidad incidente o la densidad del aire, el diámetro de la hélice o tanto la inicialización del número de vueltas del eje tanto como su valor actual.

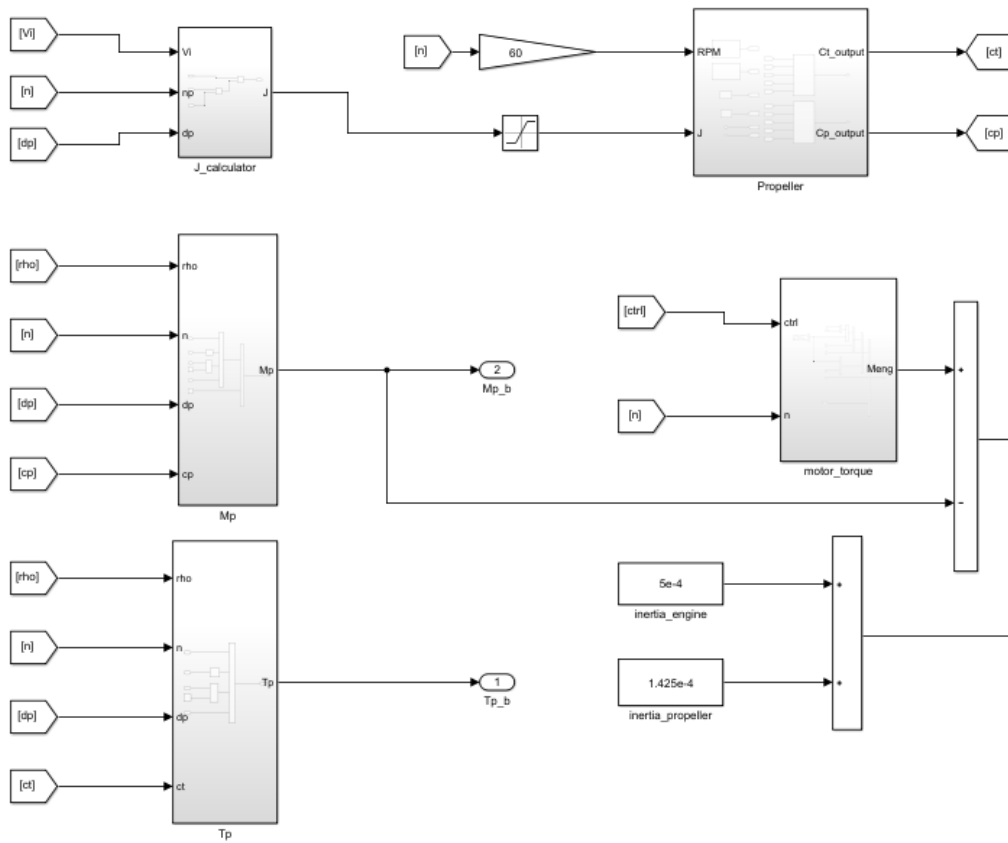


Ilustración 15 Detalle diagrama del bloque estimador motor eléctrico acoplado a hélice parte uno

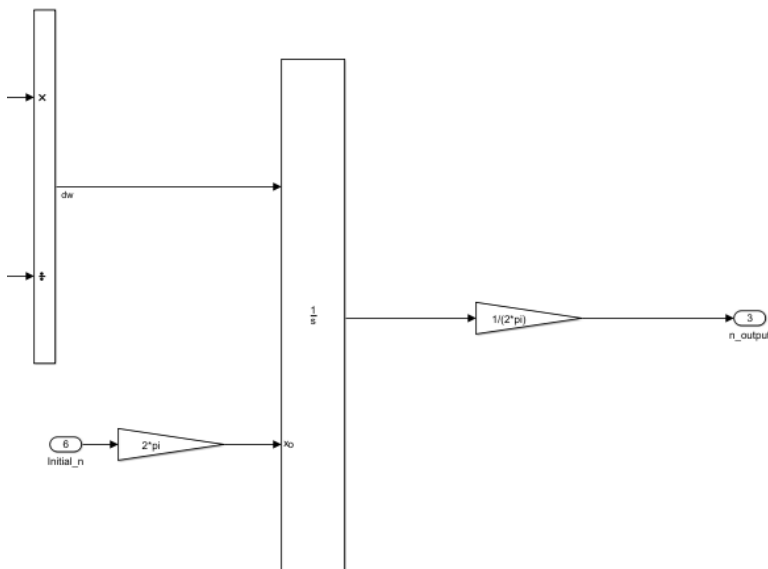


Ilustración 16 Detalle diagrama del bloque estimador motor eléctrico acoplado a hélice parte dos

Los coeficientes de potencia y propulsivos están íntimamente relacionados con el parámetro de avance J , el cual se computa en la parte superior del diagrama. A partir de ahí se computan el empuje y el momento

aerodinámico generado, el cual es necesario ya que será la principal resistencia que tenga que vencer el motor eléctrico y fijará el número de vueltas por minuto.

3.1.1.4. Bloque atmósfera

Este bloque recoge las ecuaciones de la atmosfera estándar internacional. Al contrario de su uso habitual en los vuelos reales que es una medida directa, en la simulación se obtiene la altitud como una resultante de la integración de las fuerzas, y desde la cual se transforma en la presión o temperatura, utilizando las ecuaciones de forma inversa.

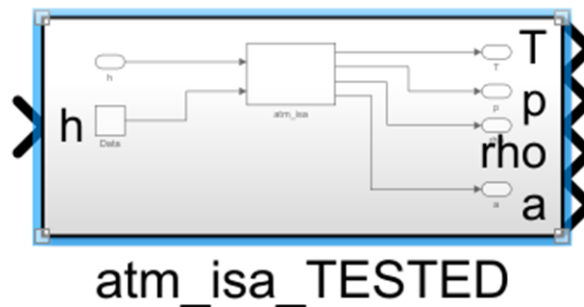


Ilustración 17 Bloque atmósfera estándar internacional

Como se observa en la Ilustración 17 se puede observar como la entrada es la altitud, y las salidas son tanto la temperatura como la presión, la densidad y la velocidad del sonido.

```
function [T,p,rho,a] = atm_isa(h,data)
    T0=data(1);
    alpha0=data(2);
    p0=data(3);
    rho0=data(4);
    Ra=data(5);
    g=data(6);
    Tl=data(7);
    pl=data(8);
    rho1=data(9);
    k=data(10);

    T=T0-alpha0*h;
    p=p0*(1-(alpha0*h/T0))^(g/(Ra*alpha0));
    rho=rho0*(1-(alpha0*h/T0))^(g/(Ra*alpha0)^-1);
    a=sqrt(k*Ra*T);

    if h>=11000
        T=Tl;
        p=pl*exp(-(g*(h-11000))/(Ra*Tl));
        rho=rho1*exp(-(g*(h-11000))/(Ra*Tl));
        a=sqrt(k*Ra*T);
    end
end
```

Ilustración 18 Detalle de las ecuaciones de la atmosfera estándar internacional

Además de esto, en la Ilustración 18 se observa el código implementado en esta función. Como dato, obsérvese que se ha tomado en cuenta el cambio de la atmosfera en la troposfera, cambio estimado a la altitud de 11000 metros.

3.1.1.5. Sub bloque modelo tierra

Tanto este sub bloque como el que se expone en el punto [3.1.1.6](#) han estado anteriormente representados en el bloque integrador.

Es un hecho que la tierra no es completamente esférica, sino que tiene una forma que se denomina geoide. Por ello, se deben utilizar modelos para poder estimar la elevación terrestre y la gravedad en cada punto. En este caso particular, se ha utilizado el sistema de coordenadas geográficas WGS84, el cual es un modelo de tierra ampliamente utilizado, cuyo bloque y ecuaciones que se puede observar en la Ilustración 19 e Ilustración 20 respectivamente.

Se observa que este modelo depende de la latitud y que otorga tanto la gravedad local como el radio meridiano o el radio meridional que serán necesario para la estimación de la latitud y longitud posteriormente.

Evidentemente, esto es un modelo tal como la atmosfera isa y no es capaz de predecir todas las irregularidades en el campo gravitatorio.

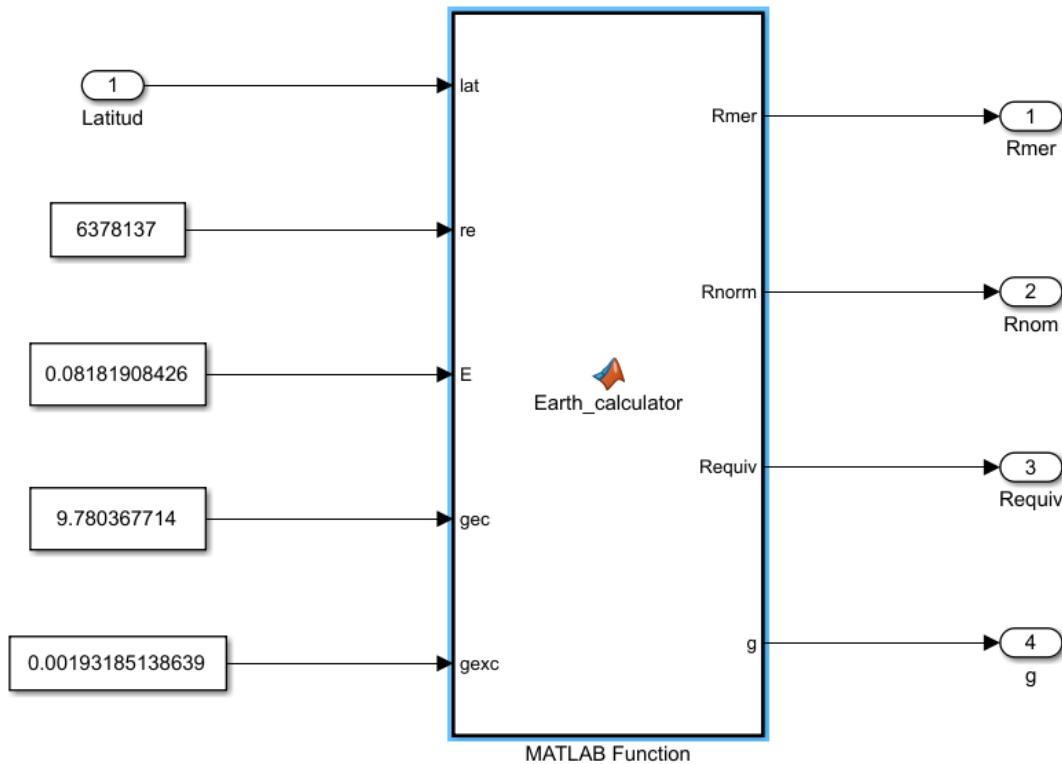


Ilustración 19 Sub bloque modelo de Tierra

```
function [Rmer,Rnorm,Requiv,g] = Earth_calculator(lat,re,E,gec,gexc)

Rmer=((re*(1-E^2))/((1-(E^2)*(sin(lat)^2))^(3/2)));

Rnorm=((re)/((1-(E^2)*(sin(lat)^2))^(1/2)));

Requiv=sqrt(Rmer*Rnorm);

g=gec*(1+gexc*sin(lat)^2)/((1-E^2*sin(lat)^2)^0.5)

end
```

Ilustración 20 Detalle ecuaciones de modelo de tierra WGS84

3.1.1.6. Sub bloque transformador sistema de referencia ECI a ECEF

Este sub bloque es el encargado de estimar las coordenadas ECEF (longitud, latitud y altura). Como se observa en la Ilustración 21, para estimar tanto la longitud como la latitud necesita un punto inicial desde el cual iniciar la integración, y el procedimiento a seguir es el de obtener las velocidades angulares a través de las velocidades horizontales locales.

Una vez estimadas ambas se integran, teniendo en cuenta las discontinuidades existentes por el hecho de utilizar ángulos mediante lógica. Para la altura, mediante una función interna de *Simulink*, en función de la longitud y latitud nos da como resultado la elevación, lo cual restado a la altitud nos da la altura.

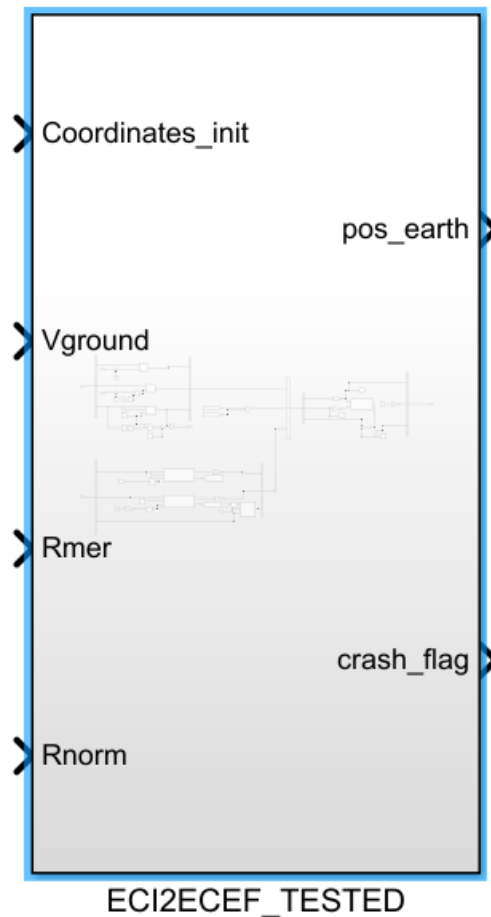


Ilustración 21 Sub bloque cálculo coordenadas ECEF

A continuación, se muestran las ecuaciones implementadas en el esquema mostrado en ilustración 21.

$$\dot{\varphi} = \frac{V_N}{R_{\text{meridiano}} + \text{Alt}} \quad (17)$$

$$\dot{\lambda} = \frac{V_E}{(R_{\text{normal}} + \text{Alt}) \cos(\varphi)} \quad (18)$$

$$\text{Alt} \begin{cases} -V_D & B_{\text{Ground}} = 0 \\ 0 & B_{\text{Ground}} = 1 \end{cases} \quad (19)$$

Donde *BGround* no es más que el booleano cuyo valor es la unidad cuando la aeronave está apoyada en tierra. Con dicho booleano se puede modelar los impactos, se evita que la aeronave traspase la tierra, y es el indicador para que las fuerzas de reacción empiecen a entrar en juego.

3.1.2. Implementación

Después del testeo de cada uno de los bloques individualmente, mediante la manipulación de las entradas y comprobando que las salidas tenían unos resultados dentro de lo esperado, se pasó al testeo del conjunto. Para ello, se realizaron simulaciones más complejas donde se involucraban varios de bloques para poder observar su respuesta.

Como ejemplo de este proceso, se va a exponer el último test que se llevó a cabo en este desarrollo de la parte del proyecto que involucra *Matalb/Simulink*. Este test en particular consistió en comprobar el funcionamiento del bloque aerodinámico y del bloque propulsivo en relación al bloque integrador cerrando el bucle mediante controladores. El fin de dichos controladores era el de implementar un autopiloto rudimentario, mediante controladores proporcionales, integrales y derivativos (PID) bajo la hipótesis de sensores perfectos, como se observa en la ilustración posterior (ampliación posterior en Ilustración 23 e Ilustración 24).

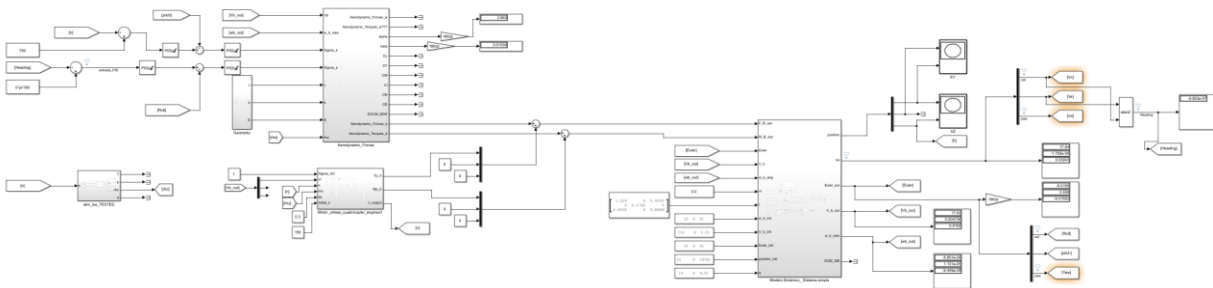


Ilustración 22 Detalle simulación de test para bloque aerodinámico y propulsivo

En dicha ilustración se observa una vista general de la integración, la cual alberga tanto los bloques aerodinámicos como el bloque propulsivo, además de un control orientado a mantener la altitud y el rumbo. Para gobernar la aeronave mediante estos controladores PID, simplemente hay que cambiar el valor de la referencia y los bloques de control actuarán sobre el timón de profundidad (en el caso de querer cambiar/mantener la altitud) y los alerones (en el caso de querer cambiar/mantener el el rumbo), con el fin de modificar la trayectoria de la aeronave.

Se puede observar que ambos controles han sido implementados mediante una estructura de doble controlador proporcional, integral y derivativo.

En el caso de la selección de la altitud, dicho el error en altitud (el valor de referencia menos el valor real estimado) se procesa para generar un ángulo de cabeceo objetivo, cuyo error es el que finalmente se procesa para enviar una orden al timón de profundidad.

En el caso de la selección del rumbo, el error en este se procesa para generar un ángulo de balance objetivo, cuyo error respecto al estimado se procesa para generar un orden a los alerones.

Para poder ver que los controles funcionaban, primero se tuneo la primera capa de los controladores PID los cuales utilizan el error respecto al ángulo de cabeceo y balance para controlar la aeronave y, una vez que se estuvo satisfecho con los resultados, se hizo el tuneo de la segunda capa de control.

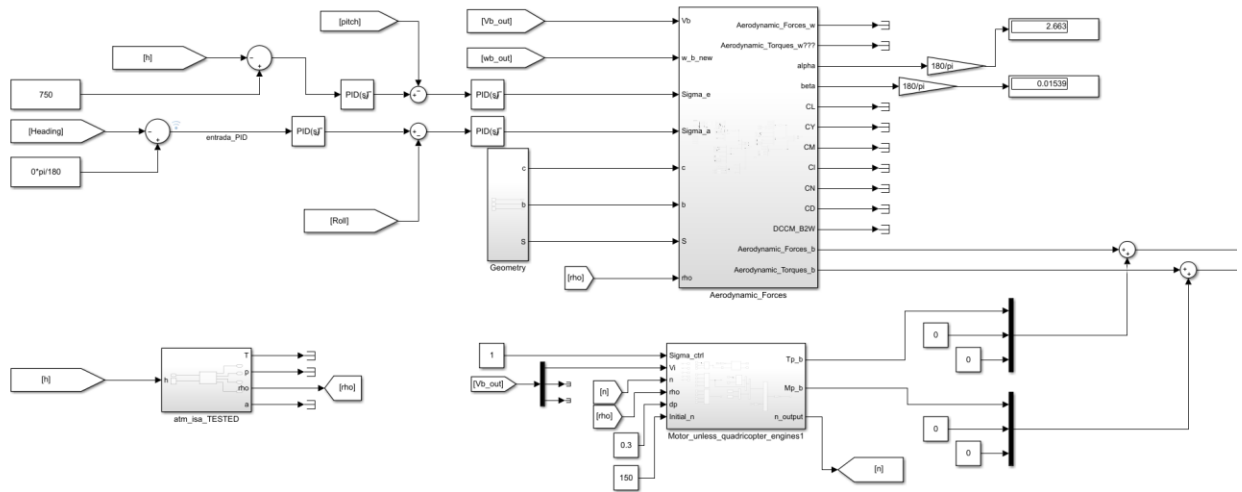


Ilustración 23 Detalle simulación completa parte uno

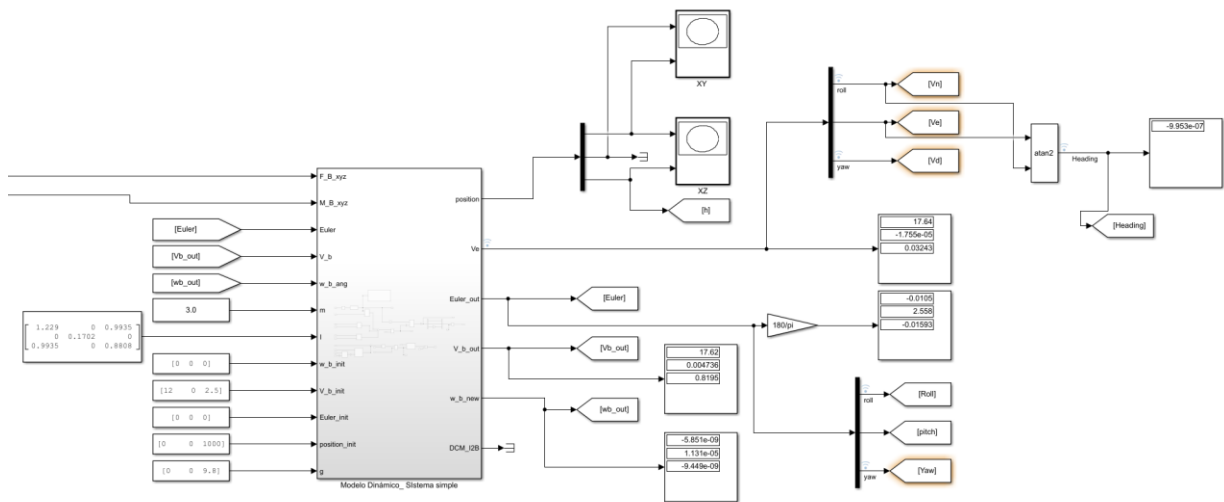


Ilustración 24 Detalle simulación completa parte dos

Como resultado, se obtuvo una simulación suave en la cual se fue capaz de controlar la trayectoria de la aeronave mediante valores de referencia, de forma similar a las aeronaves reales. Esto, por ejemplo, permite controlar el momento inducido por el motor mediante el objetivo en rumbo, ya que este a su vez controla el ángulo de balance.

En la Ilustración 25, Ilustración 26 e Ilustración 27 se puede observar la altitud frente a la altitud objetivo, el ángulo de balance respecto el tiempo y el rumbo frente al rumbo objetivo.

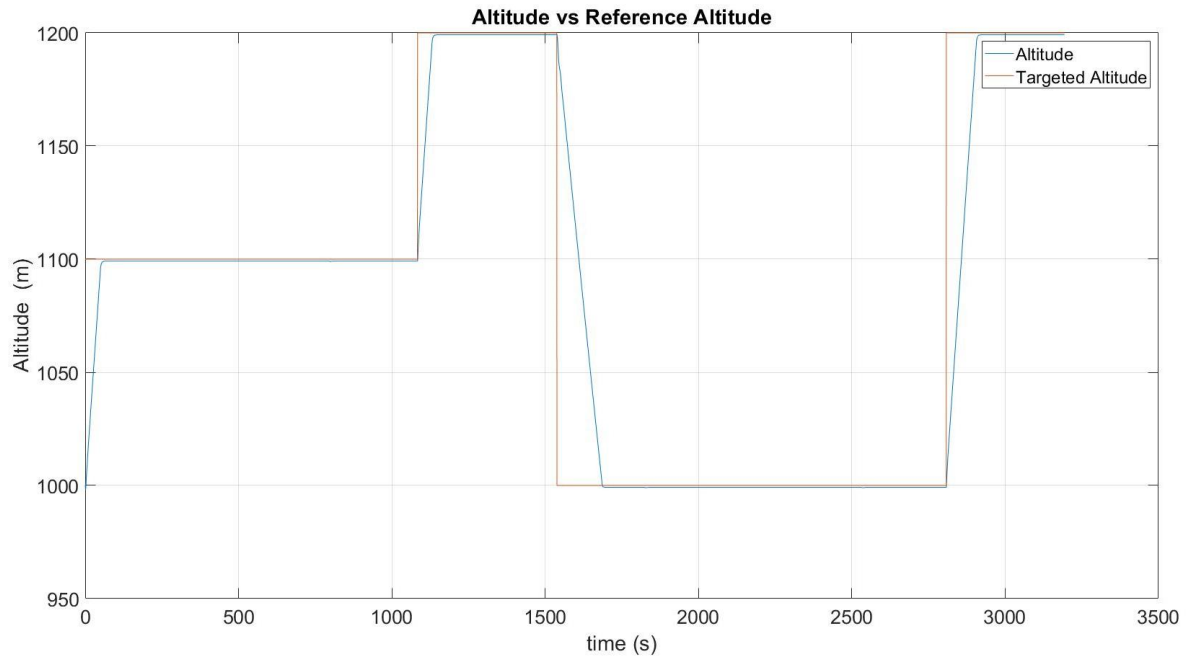


Ilustración 25 Detalle captura altitud

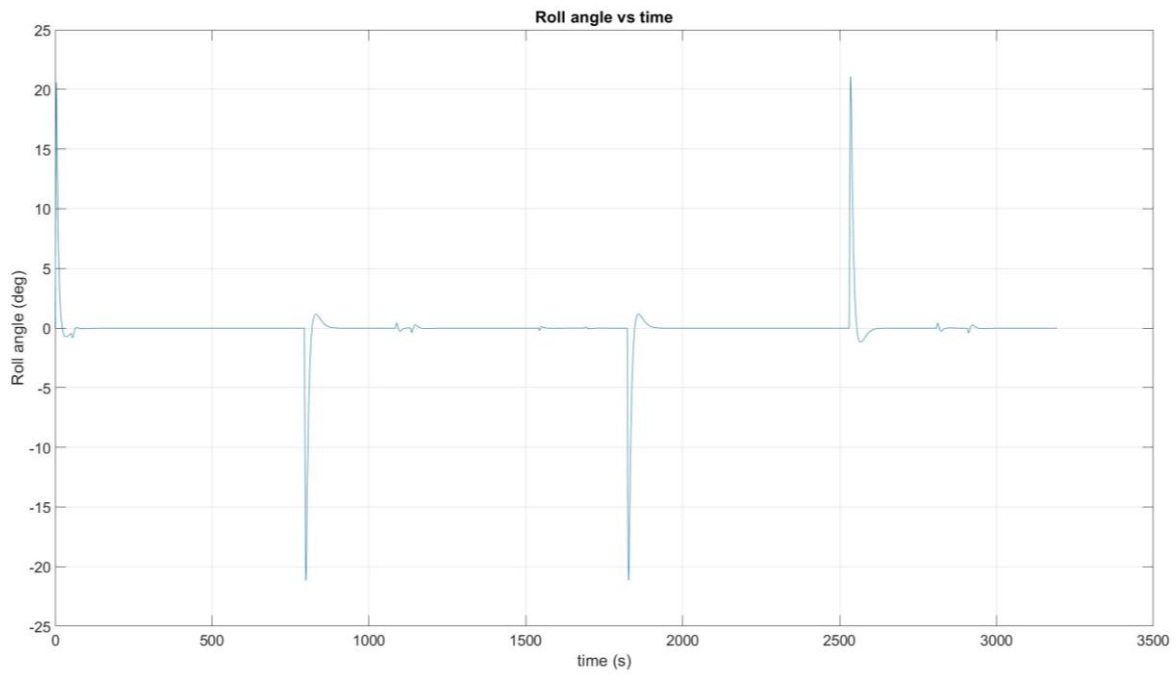


Ilustración 26 Detalle captura ángulo de balance

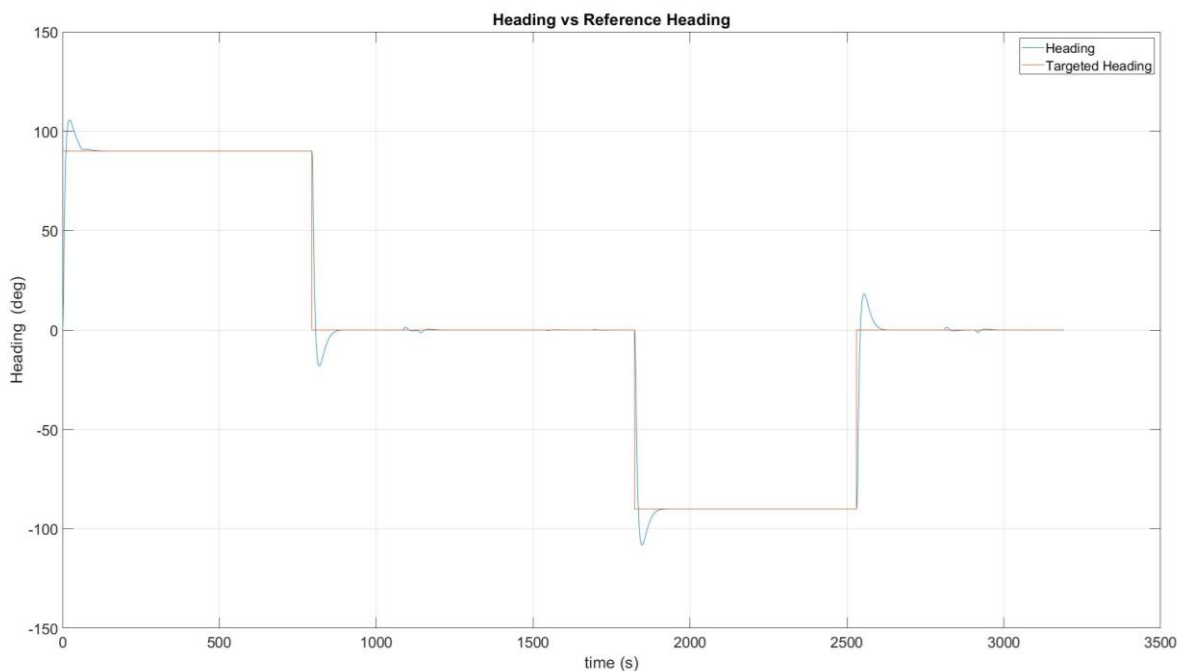


Ilustración 27 Detalle captura rumbo

De esta forma se dejaron las bases para poder testear los subsecuentes bloques, al poder utilizar la aeronave en un amplio rango de funcionamiento.

3.1.3. Ventajas e inconvenientes

Una vez implementados todos estos módulos y habiendo sido testeados, se empezaron a ver las diferentes ventajas e inconvenientes que tenía esta plataforma para la consecución del proyecto.

3.1.3.1. Ventajas

Versatilidad:

Al ser *Matlab/Simulink* realmente una herramienta matemática, se pueden implementar en ella diversos tipos de modelo sin importar su naturaleza, siempre y cuando estén modelados matemáticamente, cumpliendo así la primera premisa que se marcó en este proyecto.

A la hora de simular un avión donde hay tantos campos diferentes relacionados entre sí, desde la aerodinámica hasta el sistema eléctrico pasando por la rigidez de la estructura, esta característica es crucial.

Modularidad:

Cumpliendo otra premisa de este proyecto, la herramienta *Simulink* permite construir el simulador a base de diferentes bloques. Esta característica hace muy fácil la certificación del total ya que, una vez terminado un bloque, se le puede hacer una batería de test para ver si este funciona a priori como debería, antes de integrarlos en estructuras de más alto nivel.

A esto se le suma que es muy fácil la reutilización de código ya testado, pudiendo programar de lo más sencillo a lo más complejo por adición.

Interfaz visual:

Esta característica es bastante útil a la hora de que un usuario empiece a codificar por primera vez. El hecho de que no sea código y que sea un diagrama de bloques, hace que una persona totalmente ajena al proyecto pueda llegar a comprender como funciona y como se podría modificar y/o añadir nuevas funcionalidades.

Además, a la hora de corregir comportamientos indeseables permite atacar de una forma fácil a los bloques y buses en los que creemos que esta el fallo, teniendo varias herramientas de visualización de datos. Estas son usadas para comprobar unidades, discontinuidades, coherencia etc.

3.1.3.2. Inconvenientes

Plataforma de pago:

Esto va en contra de una de las premisas marcadas para el proyecto, haciendo que la distribución del mismo y su alcance sea más difícil para los alumnos, profesorado y terceros.

Consumo de recursos:

Uno de los grandes problemas de la *Simulink* es la cantidad de recursos computacionales que consume. Esto hace que a la hora de extender el proyecto con más bloques y hacer el sistema más complejo, se podría llegar a necesitar ordenadores cada vez más potentes, haciendo su accesibilidad aún más limitada.

Rigidez:

Una vez implementados todos estos bloques, se empezaron a desarrollar dos nuevos bloques, los cuales estarían encargados de la entrada y salida de datos. La idea principal es que un avión fuera caracterizado mediante un archivo XML y que mediante una serie de *sockets* (puntos virtuales de intercambio de datos) se pudiera verter la información hacia el exterior.

Si bien es cierto que lo segundo (la apertura de *sockets*) se puede hacer de una forma relativamente sencilla, lo primero sí que se antojaba complicado, sobre todo desde el punto de vista del tiempo que supondría su implementación.

3.1.4. Conclusión

Matlab/Simulink es una herramienta potente capaz de simular casi cualquier sistema físico debido a sus amplios recursos, siempre desde un lenguaje de alto nivel. Esto cumple con creces el carácter docente que se le quiere dar al proyecto, ya que cualquier persona con ciertos conocimientos podría entender y trabajar sobre el simulador en poco tiempo sin ser un experto en la herramienta. Esto, junto a sus interfaces ya implementadas, hace que sea una herramienta perfecta para el análisis de la simulación.

No obstante, el hecho de empezar desde cero hace que mucho de los bloques que se han implementado ya se hayan realizados anteriormente por otros alumnos, o incluso que haya librerías dentro de *Simulink* con el mismo fin, y esto le da poco valor añadido a esta vía.

Además, si se recuerda lo mencionado en el apartado [1.3](#), la cual muestra ambas partes del simulador teórico, la estación y la simulación, con este desarrollo se habría tocado la segunda parte, pero prácticamente nada de la primera.

Teniendo esto en cuenta, implementar una interfaz de control con *Matlab/Simulink* para desarrollar esta estación podría ser costoso en términos temporales, teniendo en cuenta las facilidades que ofrecen otras plataformas.

Por ello, se decidió explorar otras posibilidades antes de elegir por qué camino continuar, llegando así a *JSBSim*.

3.2. JSBSim

3.2.1. Introducción a JSBSim

JSBSim no es un simulador por sí mismo, sino que se trata de un modelo dinámico de vuelo (*FDM: Flight Dynamic Model*) como se observa en la Ilustración 4, en el apartado [1.3](#).

Este modelo dinámico está basado en un sistema sólido rígido con seis grados de libertad. A priori, con este sistema se podría reproducir el movimiento de cualquier objeto en el universo, siempre y cuando se sea capaz de modelar y computar las fuerzas aplicadas en él. En el caso del vuelo de una aeronave, se deben identificar las fuerzas y momentos aerodinámicos relativos a la aeronave (dependiendo de su índole) además de las fuerzas propulsivas y de reacción, con el fin de propagar su dinámica, orientación y posición a través del tiempo en toda la envolvente de vuelo.

Con dicho fin se deben hacer caracterizaciones de diferentes elementos ya introducidos, tales como la aerodinámica del vehículo, el entorno, la atmósfera, el geóide WGS84, las reacciones de contacto y la gravedad. Dentro de este último bloque entraría el modelo de la planta propulsiva.

Ya en una capa más compleja, las características del control de vuelo deben ser caracterizadas también, así como los sistemas de navegación, control y guiado.

Debido a todos estos modelos matemáticos, será necesario encontrar una forma sencilla para extraer e introducir datos al modelo dinámico, tanto como para poder gobernar la aeronave como para alimentar sistemas propios de la simulación.

Este tipo de modelos pueden ser ejecutados para una simulación cinemática y dinámica por sí mismas o pueden formar parte de una estructura superior. Esto es realmente útil para el diseño, ya que se pueden programar baterías de test para certificar sistemas, o iterar a la hora de dimensionar las superficies de las aeronaves con un objetivo basado en actuaciones. Esto es posible ya que puede ser ejecutado en tiempo real (interesante, ya que permitiría visualizar en tiempo real el comportamiento del avión, e incluso introducir a un piloto humano en el bucle de simulación) o a una frecuencia superior, solamente limitada por la potencia del ordenador anfitrión (interesante para la obtención de datos).

Este modelo está desarrollado en el lenguaje C++, básicamente por ser un lenguaje enfocado a entornos orientados a objetos, los cuales están basados en cuatro conceptos: polimorfismo, herencia, encapsulación y abstracción. Estos objetos están definidos por clases. Por ejemplo, un coche es un objeto, y cada coche tiene unos atributos (como el peso, motor y color) y unas funciones (como guiar o frenar). Todos esos atributos y funciones conforman una plantilla de objeto genérico, lo cual serían las clases.

En la Ilustración 28 se observa cómo está estructurada la librería *JSBSim* para modelar dinámicas de vuelo, mediante una estructura de árbol. Las distintas clases que conforman dicho árbol están divididas en función de su objetivo, como pueden ser clases orientadas a la matemática, al envío y lectura de entradas y salidas, modelos y clases básicas, etc.

Por ejemplo, *FGmodel*, la clase que modela la aeronave, necesita de otras clases más específicas, las cuales aportan distintas características como la aerodinámica (*FGAerodynamics*) o la masa y el centrado de la aeronave (*FGMassBalance*), creando una jerarquía de clases, que van de lo concreto a lo general.

Otro ejemplo interesante es *FGEngine*. Debido a que no todos los motores se modelan de la misma forma como ya se ha introducido anteriormente, existen diferentes clases para cada uno de ellos. Esto será realmente interesante ya que las variables de control de los mismos también cambiarán, teniéndose que tener en cuenta para el diseño de futuros sistemas e interfaces.

Se puede ver también que en lo más alto de este libro de dependencias encontramos *FGJSBBase*, clase que se encarga de gestionar la comunicación entre las distintas clases dependientes.

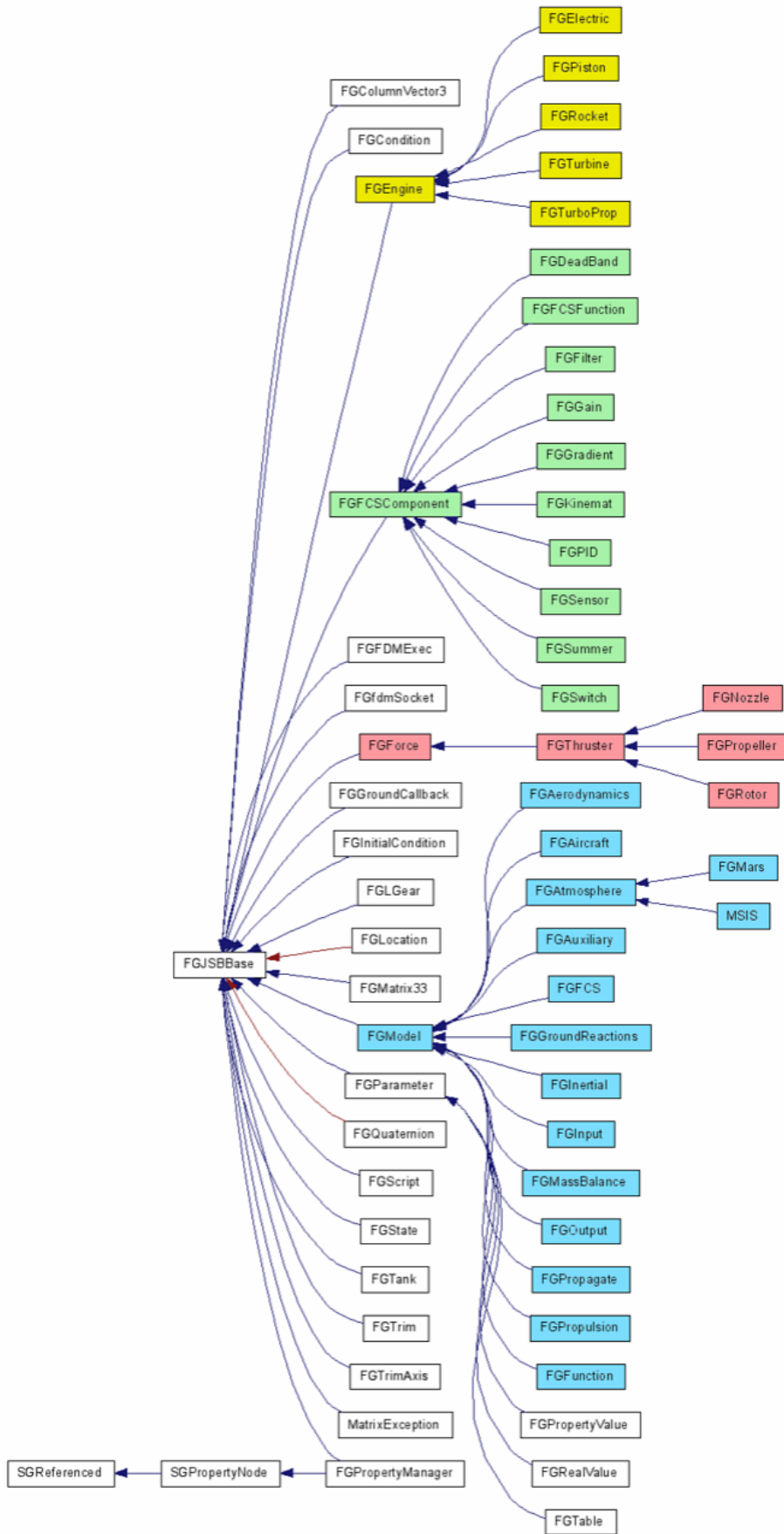


Ilustración 28 Jerarquía de clases en JSBSim

De esta forma y con los ejemplos anteriores, se puede observar el concepto de jerarquía, la reutilización de código y la modelización de las funciones.

Entre todas estas clases, se encuentra la clase de ejecución (FGFDMExec). Esta clase es utilizada para crear un objeto FGFDMExec, cargar los archivos de inicialización y de la ejecución de la simulación hasta que esta sea completada.

Estos archivos son del tipo XML (*Extensivle Markup Language*) y en ellos están recogidos datos que caracterizan a la propia aeronave, como son sus superficies de control o el tipo de motor que este utiliza, además de iniciar todos los sistemas, posición y actitud inicial.

La manera de ejecutar una simulación sigue el patrón que se indica en la Ilustración 29.

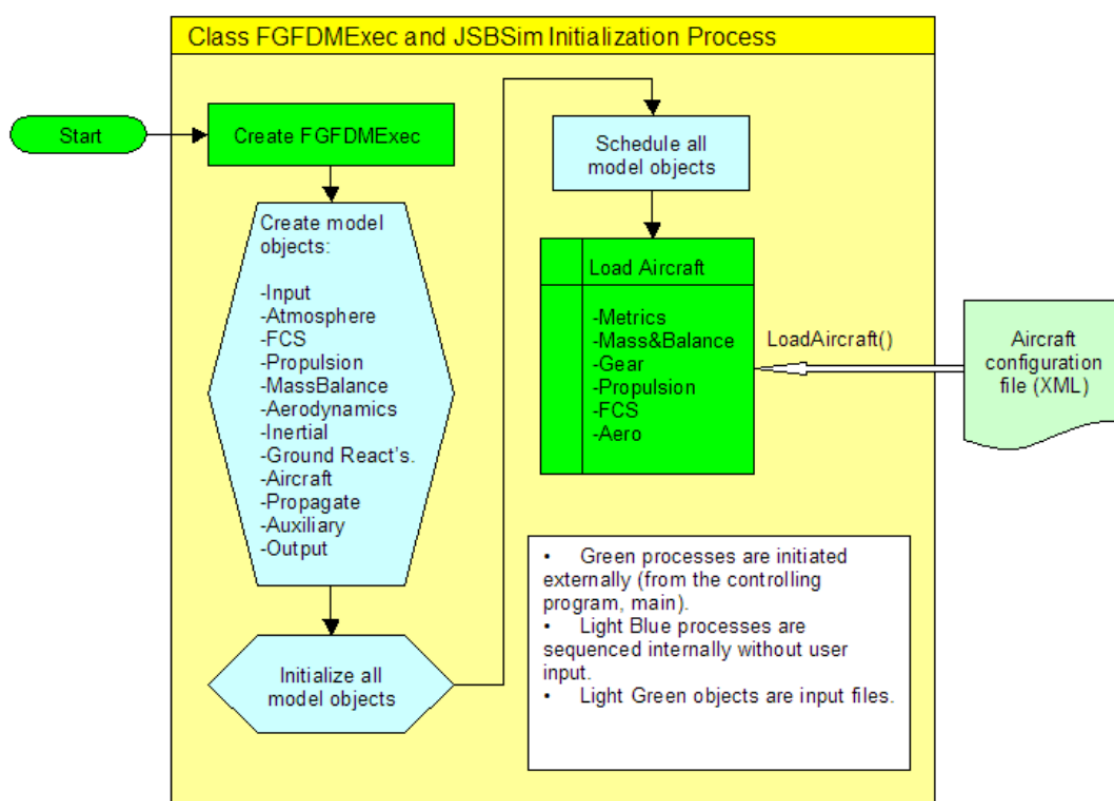


Ilustración 29 Proceso de inicialización y utilización de la clase FGFDMExec

Debido a la importancia que tienen estos archivos XML y la utilidad que tienen, se va a proseguir a su introducción.

3.2.2. Archivo de configuración de avión

Aircraft Configuration File o Archivo de Configuración de Avión es un archivo XML donde quedan recogidas todas las características de un avión a simular. Estos siguen una forma estándar a la hora de introducir los datos, por lo que se podría generar un archivo de cualquier aeronave real o en vía de diseño, de una forma sencilla.

Los apartados que se van a introducir a continuación son:

- Métrica
- Masa y centrado
- Fuerzas de flotación
- Reacciones en tierra
- Reacciones externas

- Planta propulsiva
- Aerodinámica
- Sistemas, autopiloto y control de vuelo
- Entradas
- Salidas

Para su mejor comprensión, se introducirá en cada uno de ellos un ejemplo, todos ellos extraídos de [1] *JSBSim: AN open source, platform-independent, flight dynamics model in C++*, en el cual se modela a un B747-100. Para más información, acudir a la fuente.

3.2.2.1. Métrica

En esta sección se definen las medidas y superficies principales del vehículo, así como la localización de puntos clave. Se puede observar un ejemplo a continuación.

```
<wingarea unit="FT2"> 174.0 </wingarea>
<wingspan unit="FT"> 35.8 </wingspan>
<chord unit="FT"> 4.9 </chord>
<htailarea unit="FT2"> 21.9 </htailarea>
<htailarm unit="FT"> 15.7 </htailarm>
<vtailarea unit="FT2"> 16.5 </vtailarea>
<vtailarm unit="FT"> 15.7 </vtailarm>
<location name="AERORP" unit="IN">
  <x> 43.2 </x>
  <y> 0.0 </y>
  <z> 59.4 </z>
</location>
<location name="EYEPOINT" unit="IN">
  <x> 37.0 </x>
  <y> 0.0 </y>
  <z> 48.0 </z>
</location>
<location name="VRP" unit="IN">
  <x> 42.6 </x>
  <y> 0.0 </y>
  <z> 38.5 </z>
</location>
```

Ilustración 30 Detalle codificación métrica

En la Ilustración 30 se observan datos tales como la superficie mojada, la envergadura del avión o las medidas de las estabilizadores horizontales y verticales. Es más, se deben introducir las unidades, ya que *JSBSim* puede trabajar tanto en sistema métrico como en el sistema imperial.

Posteriormente introduce tres puntos importantes, como el punto de referencia aerodinámico, el punto donde está localizado el piloto (*EYEPOINT*) desde el cual se computarán las fuerzas que vería este, y el VRP o *Vehicule Reference Point* (Punto de Referencia del Vehículo), el cual es un punto geométrico ligado al sistema de referencia ejes cuerpo desde el cual se suele tomar como referencia (el centro de gravedad se mueve a medida que se consume combustible). Usualmente este es tomado como la nariz del avión.

3.2.2.2. Masa y centrado

En esta sección se especifican las propiedades másicas del avión objetivo. En este archivo están recogidos datos como el peso en vacío de la aeronave, los momentos de inercia, localización del centro de gravedad y definiciones de todos los puntos de masa potenciales. Se puede observar un ejemplo a continuación.

En la Ilustración 31 se pueden observar características como el centro de gravedad, los momentos de inercia o el peso en vacío de la aeronave. Además de esto introduce un punto de masa donde quedaría recogida la carga de pago. Esto es muy importante a la hora de los estudios de estabilidad o de simular extracciones.

```
<mass_balance>
  <documentation>
    The Center of Gravity location, empty weight, in aircraft's own
    structural coord system.
  </documentation>
  <ixx unit="SLUG*FT2"> 2.31442e+06 </ixx>
  <iyy unit="SLUG*FT2"> 1.34649e+06 </iyy>
  <izz unit="SLUG*FT2"> 3.59608e+06 </izz>
  <ixy unit="SLUG*FT2"> 0 </ixy>
  <emptywt unit="LBS"> 48400 </emptywt>
  <location name="CG" unit="IN">
    <x> 459.2 </x>
    <y> 0 </y>
    <z> -31.8 </z>
  </location>
  <pointmass name="payload">
    <weight unit="LBS"> 1500 </weight>
    <location name="POINTMASS" unit="IN">
      <x> 460.0 </x>
      <y> 0 </y>
      <z> -31.8 </z>
    </location>
  </pointmass>
</mass_balance>
```

Ilustración 31 Detalle codificación masa y centrado

3.2.2.3. Fuerzas de flotación

JSBSim permite simular globos aerostáticos o zepelines mediante la modelación de las fuerzas de las fuerzas de flotación, usando el principio de Arquímedes. En este apartado se pueden caracterizar variables tales como el tipo de gas usado, su localización o incluso la presión tarada de la válvula de seguridad, cómo se observa en la Ilustración 32.

```

<buoyant_forces>
  <gas_cell type="{HYDROGEN | HELIUM | AIR}">
    <location unit="{M | IN}">
      <x> {number} </x>
      <y> {number} </y>
      <z> {number} </z>
    </location>
    <x_{radius|width} unit="{M | IN}"> {number} </x_{radius|width}>
    <y_{radius|width} unit="{M | IN}"> {number} </y_{radius|width}>
    <z_{radius|width} unit="{M | IN}"> {number} </z_{radius|width}>
    <max_overpressure unit="{PA | PSI}"> {number} </max_overpressure>
    [<valve_coefficient unit="{M4*SEC/KG | FT4*SEC/SLUG}"> {number}
      </valve_coefficient>]
    [<fullness> {number} </fullness>]
    [<heat>
      {heat transfer coefficients} [lbs ft / sec]
    </heat>]
    [<ballonet>
      <location unit="{M | IN}">
        <x> {number} </x>
        <y> {number} </y>
        <z> {number} </z>
      </location>
      <x_{radius|width} unit="{M | IN}"> {number} </x_{radius|width}>
      <y_{radius|width} unit="{M | IN}"> {number} </y_{radius|width}>
      <z_{radius|width} unit="{M | IN}"> {number} </z_{radius|width}>
      <max_overpressure unit="{PA | PSI}"> {number} </max_overpressure>
      <valve_coefficient unit="{M4*SEC/KG | FT4*SEC/SLUG}"> {number}
        </valve_coefficient>
      [<fullness> {number} </fullness>]
      [<heat>
        {heat transfer coefficients} [lb ft / (sec R)]
      </heat>]
      [<blower_input>
        {input air flow function} [ft^3 / sec]
      </blower_input>]
    </ballonet>]
  </gas_cell>
</buoyant_forces>

```

Ilustración 32 Detalle fuerzas de flotación

3.2.2.4. Reacciones en tierra

Los contactos del tren de aterrizaje entre el avión y el suelo deben ser modelados para un despegue y aterrizaje realista. Además de esto se puede marcar puntos de contacto, puntos discretos de la estructura capaces de colisionar con tierra, tales como la punta de las alas, nariz o cola.

Esta modelización es interesante ya que, durante situaciones de despegue u aterrizaje anómalas, ya sea por vientos cruzados fuertes, inputs de los pilotos u otros posibles fallos, se pueden dar lugar a impactos de la aeronave con la pista y suelen ser ampliamente estudiados a la hora de diseñar las leyes de control de vuelo.

En la Ilustración 33 se puede observar dichos dos tipos de contacto, el tipo *BOGEY* donde quedan caracterizados los contactos del tren de aterrizaje con el suelo, y el tipo *STRUCTURE*, que caracteriza los puntos del fuselaje capaz de impactar con el suelo.

```

<ground_reactions>

  <contact type="BOGEY" name="NOSE_GEAR">
    { nose gear description ... }
  </contact>

  <contact type="BOGEY" name="LEFT_MAIN">
    { left main gear description ... }
  </contact>

  <contact type="BOGEY" name="RIGHT_MAIN">
    { right main gear description ... }
  </contact>

  <contact type="STRUCTURE" name="LEFT_WING_TIP">
    { left wing tip description ... }
  </contact>

  <contact type="STRUCTURE" name="RIGHT_WING_TIP">
    { right wing tip description ... }
  </contact>

  <contact type="STRUCTURE" name="TAIL_SKID">
    { tail skid description ... }
  </contact>

  ... additional contacts ...

</ground_reactions>

```

Ilustración 33 Detalle reacciones con tierra

La diferencia principal entre ambos tipos es la forma en la que se computan las fuerzas de reacción. Aunque en ambas básicamente la fuerza es una resultante que se opone a la penetración del avión dentro de la tierra, los *BOGEY* incluyen características como frenos, dirección en tierra, o si es un tren retráctil. La última característica solo implica tener en cuenta el tren a la hora de un contacto, su aportación aerodinámica tendrá que ser modelada en el apartado aerodinámico. Esto se puede observar, junto a otras características en la Ilustración 34.

```

<contact type="BOGEY" name="NOSE_GEAR">
  <location unit="IN">
    <x> -6.8 </x>
    <y> 0 </y>
    <z> -20 </z>
  </location>
  <spring_coeff unit="LBS/FT"> 1800 </spring_coeff>
  <damping_coeff unit="LBS/FT/SEC"> 600 </damping_coeff>
  <static_friction> 0.8 </static_friction>
  <dynamic_friction> 0.5 </dynamic_friction>
  <rolling_friction> 0.02 </rolling_friction>
  <max_steer unit="DEG"> 10 </max_steer>
  <brake_group> NONE </brake_group>
  <retractable>0</retractable>
</contact>

```

Ilustración 34 Detalle reacción tipo BOGEY

3.2.2.5. Reacciones externa

JSBSim presenta este módulo para poder modelar reacciones externas que añadan momentos o fuerzas a la aeronave, tales como catapultas o paracaídas. En estos se debe declarar características tales como el punto de aplicación de la fuerza, la dirección de aplicación de esta, y el valor o función que sigue dicho valor. Esto se

puede observar en la Ilustración 35, donde se da como ejemplo un paracaídas.

```

<external_reactions>
  <!-- "Declare" the reefing term -->
  <property>fcs/parachute_reef_pos_norm</property>

  <force name="parachute" frame="WIND">
    <function>
      <product>
        <p> aero/qbar-psf </p>
        <p> fcs/parachute_reef_pos_norm </p>
        <v> 1.0 </v> <!-- Full drag coefficient -->
        <v> 20.0 </v> <!-- Full parachute area -->
      </product>
    </function>
    <!-- The location below is in structural frame (x positive
         aft), so this location describes a point 1 foot aft
         of the origin. In this case, the origin is the center. -->
    <location unit="FT">
      <x>1</x>
      <y>0</y>
      <z>0</z>
    </location>
    <!-- The direction describes a unit vector. In this case, since
         the selected frame is the WIND frame, the "-1" x component
         describes a direction exactly opposite of the direction
         into the wind vector. That is, the direction specified below
         is the direction that the drag force acts in. -->
    <direction>
      <x>-1</x>
      <y>0</y>
      <z>0</z>
    </direction>
  </force>
</external_reactions>

```

Ilustración 35 Detalle ejemplo reacciones externas

3.2.2.6. Planta propulsiva

La sección de la planta propulsiva está dividida en dos partes bien diferenciadas.

El objetivo de la primera es definir la localización y orientación de los diferentes motores de la aeronave. El modelo de motor y el tipo de este viene dado en otro archivo de configuración específico que será introducido posteriormente en esta misma sección.

El objetivo de la segunda parte es la de modelar los tanques de combustible, define el tipo de combustible que se usa y la prioridad a la hora de consumirlo. Este apartado de los más importantes de *JSBSim*, ya que permite el cálculo en tiempo real del centro de gravedad, característica importante a la hora de estudiar la estabilidad tanto dinámica como estática para diferentes configuraciones de vuelo.

```

<propulsion>
  <engine file="filename">
    <location unit="IN">
      <x> number </x>
      <y> number </y>
      <z> number </z>
    </location>
    <orient unit="DEG">
      <roll> number </roll>
      <pitch> number </pitch>
      <yaw> number </yaw>
    </orient>
    <feed> number </feed>
    [<feed> number </feed>
    ...]
    <thruster file="filename">
      <location unit="IN">
        <x> number </x>
        <y> number </y>
        <z> number </z>
      </location>
      <orient unit="DEG">
        <roll> number </roll>
        <pitch> number </pitch>
        <yaw> number </yaw>
      </orient>
      <sense> +/-1 </sense>
      <p_factor> number </p_factor>
    </thruster>
  </engine>
  [... additional engines if present ...]
  <tank type="type name">
    <location unit="IN">
      <x> number </x>
      <y> number </y>
      <z> number </z>
    </location>
    <capacity unit="LBS"> number </capacity>
    <contents unit="LBS"> number </contents>
  </tank>

```

Ilustración 36 Detalle planta propulsiva

En la Ilustración 36 se puede observar un esquema de un archivo de configuración de la propulsión. Se puede distinguir la localización del motor, la orientación de este y propulsor (en el caso que lo hubiera) el cual transformara la potencia del motor en fuerza. Para ello necesitara un archivo externo el cual sirva para modelar la física del proceso, como en el caso de las hélices.

En la Ilustración 37 se puede observar un ejemplo del archivo de configuración específico del modelo de planta motora, en este caso una turbina CFM56, usualmente montada en aviones A320. En este caso, como ya se había adelantado, para modelar el empuje se utilizan interpolaciones de tablas. Estos datos serán utilizados por la función *FGEngine* para computar el empuje, el régimen del motor y el consumo.

Se observa que hay una triple interpolación en estas tablas, dependiendo de la posición de la palanca de gases, el Mach de la aeronave y su altitud. En función de ellos se obtiene un multiplicador el cual se aplica al empuje máximo de la aeronave a nivel del mar, estimándose así el empuje.

```

1  <?xml version="1.0"?>
2  <!--
3  File:      CFM56_5.xml
4  Author:   Aero-Matic v 0.8
5
6  Inputs:
7  name:      CFM56_5
8  type:      turbine
9  thrust:    25000 lb
10 augmented? no
11 injected?  no
12 -->
13
14 <turbine_engine name="CFM56_5">
15   <milthrust> 25000.0 </milthrust>
16   <bypassratio> 5.9 </bypassratio>
17   <tsfc> 0.657 </tsfc>
18   <idlen1> 30.0 </idlen1>
19   <idlen2> 60.0 </idlen2>
20   <maxn1> 100.0 </maxn1>
21   <maxn2> 100.0 </maxn2>
22   <augmented> 0 </augmented>
23   <injected> 0 </injected>
24
25   <function name="IdleThrust">
26     <table>
27       <independentVar lookup="row">velocities/mach</independentVar>
28       <independentVar lookup="column">atmosphere/density-altitude</independentVar>
29       <tableData>
30         .....
31         -10000    0    10000    20000    30000    40000    50000    60000
32         0.0  0.0420  0.0436  0.0528  0.0694  0.0899  0.1183  0.1467  0.0
33         0.2  0.0500  0.0501  0.0335  0.0544  0.0797  0.1049  0.1342  0.0
34         0.4  0.0040  0.0047  0.0020  0.0272  0.0595  0.0891  0.1203  0.0
35         0.6  0.0     0.0     0.0     0.0     0.0276  0.0718  0.1073  0.0
36         0.8  0.0     0.0     0.0     0.0     0.0174  0.0468  0.0900  0.0
37         1.0  0.0     0.0     0.0     0.0     0.0     0.0422  0.0700  0.0
38       </tableData>
39     </table>
40   </function>
41
42   <function name="MilThrust">
43     <table>
44       <independentVar lookup="row">velocities/mach</independentVar>
45       <independentVar lookup="column">atmosphere/density-altitude</independentVar>
46       <tableData>
47         .....
48         -10000    0    10000    20000    30000    40000    50000    60000
49         0.0  1.2600  1.0000  0.7400  0.5340  0.3720  0.2410  0.1490  0.0
50         0.2  1.1710  0.9340  0.6970  0.5060  0.3550  0.2310  0.1430  0.0
51         0.4  1.1500  0.9210  0.6920  0.5060  0.3570  0.2330  0.1450  0.0
52         0.6  1.1810  0.9510  0.7210  0.5320  0.3780  0.2480  0.1540  0.0
53         0.8  1.2580  1.0200  0.7820  0.5820  0.4170  0.2750  0.1700  0.0
54         1.0  1.3690  1.1200  0.8710  0.6510  0.4750  0.3150  0.1950  0.0
55         1.2  0.0000  0.0000  0.0000  0.0000  0.0000  0.0000  0.0000  0.0
56       </tableData>
57     </table>
58   </function>

```

Ilustración 37 Detalle ejemplo planta motora turbofán

En la Ilustración 38 e Ilustración 39 se observa un ejemplo de modelo de hélice. Al igual que en el ejemplo anterior, estos datos se utilizarán dentro de la función *FGPropeller* para estimar el empuje y el momento acoplado.

Se observa que al igual que en *Matlab/Simulink* utiliza los anteriormente nombrados coeficientes de potencia y propulsivo para calcular las fuerzas y momentos.

$$T = C_T(J, M, Re)\rho n^2 D^4 \quad (20)$$

$$P = C_P(J, M, Re)2\pi\rho n^3 D^4 \quad (21)$$

$$J = \frac{u_0}{nD} \quad (22)$$

En las ecuaciones anteriores se puede distinguir parámetros como J , el cual es denominado como el parámetro de avance, del cual serán dependientes ambos coeficientes anteriormente introducidos. También serán dependientes del Reynolds y del Mach. Se distinguen igualmente n , que es la velocidad angular de la hélice medida en revoluciones por segundo, y D que sería el diámetro de esta en metros.

```

1  <?xml version="1.0"?>
2
3  <propeller name="Fixed-Pitch 75-inch Two-Blade Propeller">
4    <ixx> 1.67 </ixx>
5    <diameter unit="IN"> 75.0 </diameter>
6    <numblades> 2 </numblades>
7    <minpitch> 22 </minpitch>
8    <maxpitch> 22 </maxpitch>
9
10   <table name="C_THRUST" type="internal">
11     <tableData>
12       0.0  0.073
13       0.1  0.073
14       0.2  0.072
15       0.3  0.071
16       0.4  0.069
17       0.5  0.066
18       0.6  0.062
19       0.7  0.055
20       0.8  0.045
21       0.9  0.034
22       1.0  0.024
23       1.1  0.013
24       1.2  -0.006
25       1.3  -0.013
26       1.4  -0.024
27       1.5  -0.034
28       1.6  -0.045
29       1.7  -0.055
30       1.8  -0.062
31       1.9  -0.066
32       2.0  -0.069
33       2.1  -0.071
34       2.2  -0.072
35       2.3  -0.073
36       5.0  -0.073
37     </tableData>
38   </table>
39   <table name="C_POWER" type = "internal">
40     <tableData>

```

Ilustración 38 Detalle ejemplo hélice nº1

```

39 <table name="C_POWER" type = "internal">
40 <tableData>
41     0.0  0.0660
42     0.1  0.0700
43     0.2  0.0700
44     0.3  0.0660
45     0.4  0.0600
46     0.5  0.0530
47     0.6  0.0501
48     0.7  0.0469
49     0.8  0.0426
50     0.9  0.0360
51     1.0  0.0282
52     1.1  0.0191
53     1.2  0.0155
54     1.3  0.0191
55     1.4  0.0282
56     1.5  0.0360
57     1.6  0.0426
58     1.7  0.0469
59     1.8  0.0501
60     1.9  0.0516
61     2.0  0.0525
62     2.1  0.0525
63     2.2  0.0522
64     2.3  0.0511
65     2.4  0.0504
66     5.0  0.0493
67 </tableData>
68 </table>
69
70 <!-- thrust effects of helical tip Mach -->
71 <table name="CT_MACH" type="internal">
72 <tableData>
73     0.85  1.0
74     1.05  0.8
75 </tableData>
76 </table>
77
78 <!-- power-required effects of helical tip Mach -->
79 <table name="CP_MACH" type="internal">
80 <tableData>
81     0.85  1.0
82     1.05  1.8
83     2.00  1.4
84 </tableData>
85 </table>
86
87 </propeller>
88

```

Ilustración 39 Detalle ejemplo hélice n°2

Una vez abordados diferentes tipos de planta propulsiva se procede a entrar en más profundidad en el modelado de los depósitos y combustible. Como se observa en la Ilustración 36, cada motor tiene asociados varios *feed* o alimentadores, que no son más que los tanques de los cuales beben estos motores.

A estos se les puede asociar una prioridad de consumo, siendo uno la más alta. También se puede asignar un 0 en prioridad a un tanque, lo cual serviría para cerrarlo, pudiendo así diseñar patrones de consumo.

El repostaje y la vertida de combustible se iniciará en todos los tanques por igual sin importar la anteriormente comentada prioridad.

En la Ilustración 40 se puede observar un ejemplo genérico de un tanque de combustible.

```
<tank type="{FUEL | OXIDIZER}">
  [<grain_config type="{CYLINDRICAL | ENDBURNING}">
    <length unit="{IN | FT | M}"> {number} </radius>
  </grain_config>]
  <location unit="{FT | M | IN}">
    <x> {number} </x>
    <y> {number} </y>
    <z> {number} </z>
  </location>
  <drain_location unit="{FT | M | IN}">
    <x> {number} </x>
    <y> {number} </y>
    <z> {number} </z>
  </drain_location>
  <radius unit="{IN | FT | M}"> {number} </radius>
  <capacity unit="{LBS | KG}"> {number} </capacity>
  <contents unit="{LBS | KG}"> {number} </contents>
  <temperature> {number} </temperature> <!-- must be deg Fahrenheit -->
  <standpipe unit="{LBS | KG}"> {number} </standpipe>
  <priority> {integer} </priority>
  <density unit="{KG/L | LBS/GAL}"> {number} </density>
  <type> {string} </type> <!-- overrides previous density setting -->
</tank>
```

Ilustración 40 Detalle modelo tanque genérico

En la anterior ilustración se puede distinguir que se puede definir la forma del tanque, la localización tanto del depósito como de las válvulas de repostaje, la temperatura inicial del combustible y el combustible utilizado. Obsérvese algunos de los combustibles disponibles en la Ilustración 41.

FUEL	DENSITY (LBS/GAL)	YEARS USED	NOTES
AVGAS	6.02		
JET-A	6.74		
JET-A1	6.74		a.k.a. F-35
JET-B	6.48		designed for cold climates, similar to JP-4
JP-1	6.76	1944-1945	
JP-2	6.38	1945-1947	experimental
JP-3	6.34	1947-1951	designed for aircraft carrier use
JP-4	6.48	1951-1995	a.k.a. F-40, AVTAG
JP-5	6.81	1952-	a.k.a. F-44, AVCAT, designed for aircraft carrier use
JP-6	6.55	1956	used only in XB-70 Valkyrie
JP-7	6.61		used only in SR-71
JP-8	6.66	1996-	a.k.a. F-34, used since 1978 in NATO
JP-8+100		1994-	
JP-9		1979-	designed for cruise missiles

Ilustración 41 Detalle tipo de combustible

3.2.2.7. Aerodinámica

Las fuerzas y momentos aerodinámicos son modelados en este archivo de configuración. Dentro de esta sección existen seis subsecciones correspondiéndose a los tres ejes de movimiento y a los ejes de rotación. En cada una de esas subsecciones se definen diferentes aportaciones a la sustentación como se ya se introdujo en secciones anteriores como en el punto [2.5](#).

Anteriormente vimos que esta estimación por coeficientes, estos se podían dar como constantes o que fueran dependientes de diferentes variables, en función del error que se quiera asumir en función de la eficiencia. Como ya se vio en cómo se modela la planta propulsiva, estos coeficientes también pueden ser tabulados en tablas.

```

1551 <axis name="LIFT">
1552 <documentation>
1553 Add Mach 0.0 and 0.6 :
1554 - decrease (0 and x1/3 at 0 radian) landing lift to get 10 pitch;
1555 - decrease climb lift to get 10 pitch;
1556 - at low speed, lift is better (x0.75) at high AOA, because of vortex over the wing.
1557
1558 Add Mach above 0.8 :
1559 - transonic (Mach 1.06) shifts the curve vertically (XB-70).
1560 - supersonic (above Mach 1.18) shifts the curve vertically (XB-70).
1561 - adjust consumption during supersonic cruise : drag increases with alpha.
1562 </documentation>
1563 <function name="aero/coefficient/CLalpha">
1564 <description>Lift_due_to_alpha</description>
1565 <product>
1566 <property>aero/qbar-psf</property>
1567 <property>metrics/Sw-sqft</property>
1568 <property>aero/function/kCLge</property>
1569 <table>
1570 <independentVar lookup="row">aero/alpha-rad</independentVar>
1571 <independentVar lookup="column">velocities/mach</independentVar>
1572 <tableData>
1573      0.0      0.6      0.8      1.0      1.06      1.18      1.65      2.10
1574      -0.20    -0.227  -0.227  -0.680  -0.680  -0.700  -0.680  -0.780  -0.880
1575      0.00     0.000   0.067   0.200   0.200   0.180   0.200   0.100   0.000
1576      0.23     0.900   0.900   1.200   1.200   1.180   1.200   1.100   1.000
1577      0.60     0.200   0.200   0.600   0.600   0.580   0.600   0.500   0.400
1578 </tableData>
1579 </table>
1580 </product>
1581 </function>
    
```

Ilustración 42 Detalle modelo Concorde

En la Ilustración 42 se observa un detalle de un modelo aerodinámico del *Concorde*, en el cual se observa la dependencia del coeficiente de sustentación en función de dos factores, el ángulo de ataque y el Mach de vuelo.

Pero en este modelo también tiene en cuenta los cambios de configuración, es decir la extensión de flaps y slats y la posición del tren de aterrizaje. Como se observa en la Ilustración 43, el cual es un extracto de un modelo aerodinámico de la avioneta *CI72P*, la posición de flaps se puede tener en cuenta tanto como para modelar el coeficiente de resistencia constante, como para el dependiente del ángulo. En este segundo caso se puede observar una tabla de doble entrada en la cual se tiene en cuenta el ángulo de ataque y la posición de los flaps, y pudiéndose interpolar cualquier valor intermedio.

Esto se puede trasladar a todos los ejes y a todas las funciones que se puedan modelar. Cabe destacar que en función del propósito de la simulación estos modelos serán más o menos complejos, o la forma de atacarlos será completamente distinta. Esta es uno de los puntos fuertes de *JSBSim*, la total flexibilidad a la hora de modelar estas fuerzas aerodinámicas.

```

493
494
495
496
497
498
499
500
501
502
503
504
505
506
507
508
509
510
511
512
513
514
515
516
517
518
519
520
521
522
523
524
525
526
527
528
529
530
531
532
533
534
535
536
537
538
539
540
541
542
543
544
545
546
547
548
549
550

```

```

<function name="aero/coefficient/CDDf">
  <description>Delta_drag_due_to_flap_deflection</description>
  <product>
    <property>aero/qbar-psf</property>
    <property>metrics/Sw-sqft</property>
    <property>aero/function/kCDge</property>
    <table>
      <independentVar>fcs/flap-pos-deg</independentVar>
      <tableData>
        0.0000  0.0000
        10.0000 0.0070
        20.0000 0.0120
        30.0000 0.0180
      </tableData>
    </table>
  </product>
</function>
<function name="aero/coefficient/CDwbh">
  <description>Drag_due_to_alpha</description>
  <product>
    <property>aero/qbar-psf</property>
    <property>metrics/Sw-sqft</property>
    <property>aero/function/kCDge</property>
    <table>
      <independentVar lookup="row">aero/alpha-rad</independentVar>
      <independentVar lookup="column">fcs/flap-pos-deg</independentVar>
      <tableData>
        0.0000  10.0000  20.0000  30.0000
        -0.0873  0.0041  0.0000  0.0005  0.0014
        -0.0698  0.0013  0.0004  0.0025  0.0041
        -0.0524  0.0001  0.0023  0.0059  0.0084
        -0.0349  0.0003  0.0057  0.0108  0.0141
        -0.0175  0.0020  0.0105  0.0172  0.0212
        0.0000  0.0052  0.0168  0.0251  0.0299
        0.0175  0.0099  0.0248  0.0346  0.0402
        0.0349  0.0162  0.0342  0.0457  0.0521
        0.0524  0.0240  0.0452  0.0583  0.0655
        0.0698  0.0334  0.0577  0.0724  0.0804
        0.0873  0.0442  0.0718  0.0881  0.0968
        0.1047  0.0566  0.0874  0.1053  0.1148
        0.1222  0.0706  0.1045  0.1240  0.1343
        0.1396  0.0860  0.1232  0.1442  0.1554
        0.1571  0.0962  0.1353  0.1573  0.1690
        0.1745  0.1069  0.1479  0.1708  0.1830
        0.1920  0.1180  0.1610  0.1849  0.1975
        0.2094  0.1298  0.1746  0.1995  0.2126
        0.2269  0.1424  0.1892  0.2151  0.2286
        0.2443  0.1565  0.2054  0.2323  0.2464
        0.2618  0.1727  0.2240  0.2521  0.2667
        0.2793  0.1782  0.2302  0.2587  0.2735
        0.2967  0.1716  0.2227  0.2507  0.2653
        0.3142  0.1618  0.2115  0.2388  0.2531
        0.3316  0.1475  0.1951  0.2214  0.2351
        0.3491  0.1097  0.1512  0.1744  0.1866
      </tableData>
    </table>
  </product>
</function>

```

Ilustración 43 Detalle modelo aerodinámico

3.2.2.8. Sistemas, autopiloto y control de vuelo

Dentro de esta parte del código quedan recogidos varios modelos que se pueden utilizar a la hora de simular sistemas, como pueden ser simular sensores, actuadores, interruptores, PID o deflexiones de las superficies aerodinámicas.

Al ser un catálogo muy amplio, se va a mostrar un ejemplo de cómo se modela la deflexión de las superficies aerodinámicas y como esto se traduce en cambios las fuerzas y momentos inducidos, debido a que esto será crucial a la hora de simular el vuelo.

En la Ilustración 44 se puede observar la variable de control *elevator-cmd-norm*, que es la variable de control para el timón de profundidad y *pitch-trim-cmd-norm*, que es la variable de control para el trimado en pitch.

Estas están acotadas entre los valores -1 y 1, los cuales suelen ser valores típicos de entradas para *joysticks* comerciales.

Asociadas a estas variables de control, está asociada una deflexión de las superficies aerodinámicas asociadas, como se puede observar en la Ilustración 44. En este ejemplo estas estarían taradas en un rango de -28° a 23° . Estas deflexiones estarían recogidas en las salidas *elevator-pos-rad*, la cual recoge el valor real de la deflexión del timón de profundidad y la salida *elevator-pos-norm*, la cual recoge el valor normalizado.

```

249 <flight_control name="FCS: c172">
250   <channel name="Pitch">
251     <summer name="Pitch Trim Sum">
252       <input>fcs/elevator-cmd-norm</input>
253       <input>fcs/pitch-trim-cmd-norm</input>
254       <clipto>
255         <min>-1</min>
256         <max>1</max>
257       </clipto>
258     </summer>
259
260     <aerosurface_scale name="Elevator Control">
261       <input>fcs/pitch-trim-sum</input>
262       <gain>0.01745</gain>
263       <range>
264         <min>-28</min>
265         <max>23</max>
266       </range>
267       <output>fcs/elevator-pos-rad</output>
268     </aerosurface_scale>
269
270     <aerosurface_scale name="Elevator Position Normalized">
271       <input>fcs/elevator-pos-deg</input>
272       <domain>
273         <min>-28</min>
274         <max>23</max>
275       </domain>
276       <range>
277         <min>-1</min>
278         <max>1</max>
279       </range>
280       <output>fcs/elevator-pos-norm</output>
281     </aerosurface_scale>
282   </channel>

```

Ilustración 44 Detalle variables de control

En la Ilustración 45 se puede ver como se ha modelado la deflexión de los flaps en el modelo de avioneta *C172P*. Se observa que en este caso la variable de control sobre la deflexión comandada, la cual tiene cuatro posibles posiciones de palanca, que son las deflexiones de 0° , 10° , 20° o 30° , contando además con un tiempo de deflexión fijo. Posteriormente, existe otra variable que modela la deflexión real de los flaps, mediante unos límites en la deflexión mínima y máxima y una variable de deflexión normalizada entre -1 y 1.

Esta distinción entre deflexión comandada y deflexión real se hace para poder codificar diferentes discrepancias entre ellas, como por ejemplo un pasible atasco de los flaps en una cierta posición (en la literatura anglosajona *flat jamming*) o la retracción automática mediante leyes de control de vuelo debido en función de las velocidades.

```

372
373
374
375
376
377
378
379
380
381
382
383
384
385
386
387
388
389
390
391
392
393
394
395
396
397
398
399
400
401
402
403
404
405
406
407
<channel name="Flaps">
  <kinematic name="Flaps Control">
    <input>fcs/flap-cmd-norm</input>
    <traverse>
      <setting>
        <position>0</position>
        <time>0</time>
      </setting>
      <setting>
        <position>10</position>
        <time>2</time>
      </setting>
      <setting>
        <position>20</position>
        <time>1</time>
      </setting>
      <setting>
        <position>30</position>
        <time>1</time>
      </setting>
    </traverse>
    <output>fcs/flap-pos-deg</output>
  </kinematic>
  <aerosurface_scale name="Flap Position Normalizer">
    <input>fcs/flap-pos-deg</input>
    <domain>
      <min>0</min> <!-- Flaps actual minimum position -->
      <max>30</max> <!-- Flaps actual maximum position -->
    </domain>
    <range>
      <min>0</min> <!-- Flaps normalized minimum position -->
      <max>1</max> <!-- Flaps normalized maximum position -->
    </range>
    <output>fcs/flap-pos-norm</output>
  </aerosurface_scale>
</channel>

```

Ilustración 45 Detalle de deflexión de flaps

Estos parámetros tendrán posteriormente su uso dentro del modelo aerodinámico, como se puede observar en la Ilustración 46. Aunque no sea el caso específico de los flaps, se puede observar cómo tiene en cuenta la variable de control *fcs/elevator-pos-rad* a la hora de estimar la resistencia inducida, en este caso debida a la deflexión del timón de profundidad.

```

695
696
697
698
699
700
701
702
703
<function name="aero/coefficient/CLDe">
  <description>Lift_due_to_Elevator_Deflection</description>
  <product>
    <property>aero/qbar-psf</property>
    <property>metrics/Sw-sqft</property>
    <property>fcs/elevator-pos-rad</property>
    <value>0.4300</value>
  </product>
</function>

```

Ilustración 46 Detalle resistencia inducida por deflexión timón de profundidad

Todas estas variables no están predefinidas y son introducidas por el usuario, lo cual permite que se puedan implementar cualquier tipo de aeronave, como por ejemplo aviones con cola en V, drones gobernados por flaperones, etc.

3.2.2.9. Entradas y salidas: sockets

A la hora de introducirse en el bucle de simulación o de la lectura de datos para poder procesarlos o mostrarlos, la simulación precisa de un sistema de intercambio de datos entre las aplicaciones que componen la simulación. Y es donde entran en juego los *sockets*.

Un *socket* es un concepto abstracto por el cual dos programas pueden intercambiar cualquier flujo de datos en red, generalmente de manera fiable y ordenada.

Para que dos programas puedan intercambiar información, es necesario dos cosas:

- Que ambos procesos sean localizables dentro de la maquina
- Que ambos procesos sean capaces de enviar paquetes de información, codificados mediante un cierto protocolo

Para cumplir la primera condición se recurre a las direcciones IP y a los puertos, los cuales indican a que red se debe atacar y dentro de esa red en que punto de entrega o en que nodos se deposita la información. Por ejemplo, si se quiere iniciar una comunicación dentro del mismo ordenador, se recurre a la IP 127.0.0.1, la cual sería la red interna del ordenador en uso, y a un puerto que no esté siendo utilizado por otros procesos, como por ejemplo el 1137. Con ello, se habría definido un nodo virtual para intercambio de información.

Para cumplir la segunda condición, hay principalmente dos tipos de protocolos usados para el intercambio de información:

- Interfaz *socket* de flujo (SOCK_STREAM)

Es un protocolo orientado a una conexión confiable y altamente utilizado en páginas web. Por ejemplo, cuando se abre un navegador web y se demanda una dirección web, por ejemplo <https://www.us.es/>, el ordenador lanza paquetes TCP al servidor de la web requerida, con la intención de que dicho servidor envíe de vuelta la página en cuestión.

El servidor al recibir la petición, envía una serie numerada de paquetes TCP, que el navegador junta para poder abrir la página web. Para asegurar que estos paquetes TCP han sido entregados, cada vez que el demandante recibe un paquete envía una confirmación. Si esta no es recibida por el servidor, el susodicho paquete será enviado todas las veces que sea necesaria hasta que se reciba la confirmación.

- Interfaz *socket* de datagrama (SOCK_DGRAM)

Este servicio no está orientado a la conexión, sino que un servidor emite ininterrumpidamente paquetes UDP, los cuales no cuentan con confirmación de llegada. Si un paquete de información es perdido por el camino, estos simplemente se no llegarán a su destino al no ser reenviados.

Este tipo de protocolo se suele utilizar cuando la velocidad de envío es primordial y el error de la conexión no es crítica, como pueden ser los servicios de emisión en directo o juegos en línea.

Una vez definidos tanto la dirección como el protocolo, dos programas podrían enviar información entre ellos, y *JSBSim* cuenta con funciones específicas para dicho fin, pudiendo abrir *sockets* de entrada y de salida.

Los *sockets* de entrada son usados principalmente para interactuar u obtener información con *JSBSim* en mitad de un proceso. Este proceso siempre se abrirá en red local, y solamente se tendrá que definir el puerto seleccionado para el intercambio de información, como se puede ver en la Ilustración 47.

```
909 <input port="1137" />
```

Ilustración 47 Detalle definición *socket* de entrada

Esta opción se suele utilizar cuando se utiliza *JSBSim* en modo *Standalone* (concepto que se abordara en profundidad posteriormente), y en el cual se suele simular una maniobra en particular en ciclo cerrado, es decir, todas las acciones son definidas con anterioridad mediante unos archivos de configuración que se introducirán más adelante. Una vez establecida la comunicación, se puede interactuar con el proceso utilizando diferentes comandos, como los usados de ejemplo en la Ilustración 48.


```

get 'property_name'
set 'property_name'
hold
resume
info
help
quit

```

Ilustración 48 Detalle comandos en los *sockets* de entrada

En la imagen anterior se pueden observar comandos de lectura, *get 'property_name'*, el cual dará como respuesta el valor actual del parámetro demandado, o comandos de escritura, *set 'property_name'*, en el cual se podrán modificar parámetros internos del proceso. Esto se puede observar en los ejemplos propuestos en la Ilustración 49 e Ilustración 50.

```

JSBSim> get inertia/cg-x-ft
inertia/cg-x-ft = 43.886700

```

Ilustración 49 Detalle comando de lectura

```

JSBSim> set ap/elevator_cmd 1

```

Ilustración 50 Detalle comando de escritura

También existe un comando para recibir información general del proceso, *info*, el cual dará información importante de la simulación que se está dando lugar. Datos como el paso de simulación o la aeronave que se está utilizando se retroalimentan, como se observa en la Ilustración 51.

```

JSBSim> info
JSBSim version: 0.9.13
Config File version: 2.0
Aircraft simulated: Cessna C-172
Simulation time: 0.008

```

Ilustración 51 Detalle comando de información general

Al contrario que para los *sockets* de entrada, los *sockets* de salida pueden ser configurados con diferentes tipos de objetivos, aunque todos ellos con la finalidad de poder leer datos de la simulación en proceso.

```

<output name="name" type="type" [port="int" protocol="UDP|TCP"]
rate="number">
  [<group> OFF|ON </group>]
  [<property> name </property>]
</output>

```

Ilustración 52 Detalle configuración estándar de un *socket* de salida

En la Ilustración 52 se observa la estructura genérica para definir un *socket* de salida. Para definir dicho *socket* se necesitaría asignarle un nombre o una dirección IP como se introducirá posteriormente (*name='name'*), un tipo de output (*type='type'*), una velocidad de vertido de información (*rate='rate'*). Siempre y cuando sean necesarios, también se deberán definir un puerto dentro de esa dirección IP (*port='port'*) y un protocolo de comunicación (*protocol='UDP/TCP'*).

También se pueden observar los parámetros que se quieren seleccionar, pudiendo ser estos elegidos por grupos, los cuales son seleccionables mediante los comandos *ON/OFF* como se observa en la Ilustración 53, o parámetro a parámetro, como se puede observar posteriormente en la Ilustración 54.

```

921 <output name="localhost" type="SOCKET" port="1140" rate="40" protocol='udp'>
922   <simulation> OFF </simulation>
923   <atmosphere> OFF </atmosphere>
924   <massprops> OFF</massprops>
925   <aerosurfaces> OFF </aerosurfaces>
926   <rates> OFF </rates>
927   <velocities> ON </velocities>
928   <forces> OFF </forces>
929   <moments> OFF </moments>
930   <position> OFF </position>
931   <propulsion> OFF </propulsion>
932   <fcs> OFF </fcs>
933   <ground_reactions> OFF </ground_reactions>
934   <coefficients> OFF </coefficients>
935 </output>

```

Ilustración 53 Detalle grupos de datos

Una vez introducidos los *sockets*, se abordarán a continuación los tipos de salida seleccionables:

- *CSV Comma separated data* (o datos separados por comas)

Este tipo de salida generara un archivo de datos al final de la simulación, en el cual se encontrarán los datos que se hayan configurado con anterioridad. Un ejemplo de cómo debe ser configurada esta salida esta mostrada en la Ilustración 54.

```

919 <output name="B737_datalog.csv" type="CSV" rate="20">
920   <property> velocities/vc-kts </property>
921   <velocities> ON </velocities>
922 </output>

```

Ilustración 54 Detalle salida tipo CSV

En la Ilustración 55 se puede observar el archivo CSV. En él se aprecian varias columnas iniciadas en el tiempo de obtención del dato, y asociados a este tiempo, los valores de cada uno de los parámetros que fueron requeridos.

1	Time	q bar (psf)	Reynolds Nu	V_{Total} (ft)	V_{Inertial}	UBody	VBody	WBody
2	0	250,506547	26467679,3	750	737,040336	750	0	5,68E-14
3	0,049998	250,40567	26462354,9	749,84873	737,039797	749,848406	-0,00078488	0,69681561
4	0,099996	250,30619	26457113,7	749,69904	737,039952	749,697833	-0,00149903	1,34521194
5	0,149994	250,207674	26451932,5	749,550294	737,040747	749,547772	-0,00214021	1,94429592
6	0,199992	250,110122	26446811,2	749,402502	737,042143	749,398354	-0,00270696	2,49331619
7	0,24999	250,013535	26441749,7	749,255678	737,044108	749,249704	-0,00319861	2,99189497
8	0,299988	249,917919	26436748,1	749,10984	737,046617	749,101941	-0,00361529	3,44000779
9	0,349986	249,823283	26431806,7	748,965009	737,049645	748,955175	-0,00395784	3,83796208
10	0,399984	249,729637	26426925,9	748,82121	737,053178	748,809508	-0,0042278	4,18637477
11	0,449982	249,636994	26422106,4	748,67847	737,057201	748,665029	-0,0044273	4,48614915
12	0,49998	249,545372	26417349	748,536819	737,061707	748,521821	-0,00455905	4,73845113

Ilustración 55 Detalle archivo CSV

Este tipo de archivos son muy útiles a la hora de analizar maniobras y actuaciones de las aeronaves, y son ampliamente utilizados en la industria. Gracias a ellos se pueden testear sistemas, leyes de control de vuelo, dimensionar las superficies aerodinámicas, y analizar fácilmente usando técnicas de *BigData*.

- *Socket*

Este tipo de salida configura al proceso para que se envíen datos a un puerto en particular. Como se observa en la Ilustración 56, la manera de configurar los datos de salida es la misma, pero requiere especificar la dirección IP y el puerto a usar.

```
<output name="192.168.0.10" type="SOCKET" port="6055" rate="20">
  <simulation> OFF </simulation>
  <atmosphere> OFF </atmosphere>
  <massprops> OFF</massprops>
  <rates> ON </rates>
  <velocities> ON </velocities>
  <forces> OFF </forces>
  <moments> OFF </moments>
  <position> OFF </position>
  <propulsion> OFF </propulsion>
  <aerosurfaces> OFF </aerosurfaces>
  <fcs> OFF </fcs>
  <ground_reactions> OFF </ground_reactions>
  <coefficients> OFF </coefficients>
</output>
```

Ilustración 56 Detalle salida tipo *socket*

Como ya se han explicado antes, en función del protocolo de comunicación que se use este afectará al proceso o no. Por ejemplo, si utilizamos un protocolo TCP, los paquetes necesitarán confirmación de recibimiento, haciendo que el proceso sufra un retraso. En cambio, el protocolo UDP enviará datos de forma secuencial sin importar si hay un retraso. Este protocolo es el que usualmente se utiliza para las simulaciones en tiempo real.

- *Flightgear*

Básicamente es un *socket* especialmente diseñado para enviar los datos necesarios para la representación visual vía *Flightgear* mediante el envío de una serie de parámetros específicos, un simulador código libre introducido en la sección [1.2](#). La forma de configurarlo es parecida al caso anterior, aunque en este caso no hace falta definir los parámetros deseados, como se puede observar en la Ilustración 57.

```
937 <output name="127.0.0.1" type="FLIGHTGEAR" protocol="UDP" port="5500" rate="60">
938 </output>
```

Ilustración 57 Detalle salida tipo *Flightgear*

- *Tabular*

Caso idéntico al CSV en el cual, en vez de generar un archivo de datos separados por comas, estos son separados por tabulaciones. Es simplemente una cuestión de formato.

3.2.3. Archivo de inicialización de la aeronave

Previamente se ha mostrado como se carga el modelo de aeronave, pero posteriormente se deben introducir una serie de valores iniciales para iniciar la simulación. Esto se hace mediante otro archivo XML, en el cual se indican los parámetros y su valor.

En la Ilustración 58 se puede observar un detalle de un archivo de inicialización. En él, se distinguen datos básicos para iniciar el proceso de simulación, como lo son las velocidades en ejes cuerpo, la posición en coordenadas geodésicas o los ángulos de Euler para representar la actitud.

Los datos que no sean inicializados específicamente, empezarán con un valor por defecto, en muchos casos 0.

```

1  <?xml version="1.0"?>
2  <initialize name="reset01">
3  <!--
4      This file sets up the aircraft to start off
5      from the runway in preparation for takeoff.
6  -->
7  <ubody>      0.0 </ubody>
8  <vbody>      0.0 </vbody>
9  <wbody>      0.0 </wbody>
10 <latitude unit="DEG"> 47.0 </latitude>
11 <longitude unit="DEG"> 122.0 </longitude>
12 <phi>        0.0 </phi>
13 <theta>      0.0 </theta>
14 <psi unit="DEG"> 150.0 </psi>
15 <altitude>   2000.00 </altitude>
16 <hwind>     0.0 </hwind>
17 <xwind>     0.0 </xwind>
18 <vc>        90.0 </vc>
19 <gamma unit="DEG"> 3.0 </gamma>
20 </initialize>

```

Ilustración 58 Detalle archivo de inicialización

3.2.4. Modos de funcionamiento de JSBSim

Una vez llegados a este punto, *JSBSim* permite de hacer la simulación desde dos puntos de vista, en función del objetivo que esté buscando el usuario.

La primera de las opciones, ejecutar el código en *Standalone*, la cual permite una ejecución de una simulación en bucle cerrado donde las entradas están predefinidas, y el código corre a la velocidad por defecto o deseada dentro de los límites de potencia del ordenador anfitrión. Esta funcionalidad está orientada al análisis de las actuaciones de un avión, enfocado al diseño de este o de los sistemas que lo componen.

En problemas de optimización o de diseño de sistemas, se pueden ejecutar baterías de test variando las condiciones iniciales y almacenar los datos resultantes de cada uno de ellos, para poder analizar finalmente los resultados, y decantarse por un diseño o por otro.

Por ejemplo, se pueden hacer pruebas con diferentes modelos de un autopiloto y comprobar las desviaciones dentro de maniobras específicas, para poder así comprobar su robustez.

La segunda de ellas, se trataría de una simulación en tiempo real, en la cual *JSBSim* suene pertenecer a un programa más grande el cual se sirve de este modelo dinámico de vuelo para estimar la cinemática y dinámica. Esto se puede observar en el esquema de la Ilustración 4 y es lo que sucede en el caso de *Flightgear*.

El final de ambos ciclos de simulación puede ser algo arbitrario, como puede ser que el usuario cambie de posición o que acabe la lección en el caso de simulaciones en tiempo real, o tener algún tipo de condición de tipo condicional, como el descenso a una altura determinada, siendo esto último especialmente útil para bucles de optimización.

A continuación, con el objetivo de arrojar algo de luz sobre estos dos modos de funcionamiento, se va a entrar en profundidad en cada uno de ellos y se expondrá un ejemplo de funcionamiento.

3.2.4.1. Modo *Standalone*

3.2.4.1.1. Introducción

Como ya se introdujo anteriormente, al ser una simulación en bucle cerrado, todas las acciones y configuración de la aeronave tienen que ser predefinidas con anterioridad. Con dicho objeto, existe otro archivo de configuración llamado *Script*, en el cual estarán recogidos los eventos que sucederán durante la simulación.

Se puede distinguir este archivo en dos partes bien diferenciadas. Una primera de inicialización, en la cual se define los archivos de configuración de avión y el archivo de inicialización de la aeronave a utilizar, definiéndose además el tiempo de simulación. Este cuenta con un valor inicial, uno final y un paso. La segunda parte está definida con una serie de eventos, los cuales son definidos una vez que se cumplan una

serie de condiciones temporales o lógicas. Posterior a este cumplimiento, se cambian una serie de propiedades, como por ejemplo diferentes cambios en los comandos, o en la configuración de la aeronave. Dichos eventos además pueden ser persistentes, lo cual implica que cada vez que el proceso evalúe que se cumple dicha condición se volverá a ejecutar dicha acción.

```

1 <?xml version="1.0" encoding="UTF-8"?>
2 <?xml-stylesheet type="text/xsl" href="http://jsbsim.sourceforge.net/JSBSimScript.xsl"?>
3 <runscript xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4   xsi:noNamespaceSchemaLocation="http://jsbsim.sf.net/JSBSimScript.xsd"
5   name="C172 cruise at 4K, 100% power">
6   <description>This run is for testing the C172 altitude hold autopilot and cruise performance</description>
7   <use aircraft="c172x" initialize="reset01"/>
8
9   <run start="0.0" end="50" dt="0.0083333">
10
11     <property value="0" persistent="true"> simulation/run_id </property>
12
13     <event name="Set Temperature">
14       <condition>
15         simulation/sim-time-sec ge 0.0
16       </condition>
17       <set name="atmosphere/delta-T">
18         <function>
19           <switch>
20             <p> simulation/run_id </p>
21             <v> 0.0 </v>
22             <v> -36.0 </v>
23             <v> 36.0 </v>
24           </switch>
25         </function>
26       </set>
27       <set name="ap/altitude_setpoint" value="4000.0"/>
28     </event>
29
30     <event name="Start engine">
31       <condition>
32         simulation/sim-time-sec ge 0.01
33       </condition>
34       <set name="fcs/throttle-cmd-norm" action="FG_RAMP" value="0.50" tc="0.05"/>
35       <set name="fcs/mixture-cmd-norm" action="FG_RAMP" value="1.0" tc="0.05"/>
36       <set name="propulsion/magneto_cmd" value="3"/>
37       <set name="propulsion/starter_cmd" value="1"/>
38       <notify/>
39     </event>

```

Ilustración 59 Detalle script para ejecutar *JSBSim* en bucle cerrado

En la Ilustración 59 se puede observar el detalle de un *script*, nuevamente en formato XML. En él, se puede ver como se inicializan ambos archivos de configuración, seguidos posteriormente de la configuración del tiempo de simulación.

Seguidamente se definen los eventos que tendrán lugar en el ciclo de simulación, observándose que el primero tendrá lugar en el instante cero de simulación configura la atmósfera y la temperatura. Posteriormente, le seguirá el segundo evento el cual pasados 0.01 segundos, se aplican cambios en los comandos para encender el motor en cuestión.

```

27 <event name="Set heading hold">
28   <description>Set Heading when 5 ft AGL is reached</description>
29   <notify/>
30   <condition>
31     position/h-agl-ft >= 5
32   </condition>
33   <set name="ap/heading_setpoint" value="200"/>
34   <set name="ap/attitude_hold" value="0"/>
35   <set name="ap/heading_hold" value="1"/>
36 </event>
37 <event name="Set autopilot for 20 ft.">
38   <description>Set Autopilot for 20 ft</description>
39   <notify/>
40   <condition>
41     aero/qbar-psf >= 4
42   </condition>
43   <set name="ap/altitude_setpoint" value="100.0"/>
44   <set name="ap/altitude_hold" value="1"/>
45   <set name="fcs/flap-cmd-norm" value=".33"/>
46 </event>

```

Ilustración 60 Detalle eventos disparados por condiciones

En contraposición de los ejemplos anteriores, en la Ilustración 60 se observa que en este caso ambos eventos tomarán lugar en el momento en el que se cumpla una condición lógica. En el primer lugar que la altitud llegue a cierto valor, y en el segundo cuando la presión dinámica alcance otro valor.

```

<event name="text" [[persistent="true|false"] |
                    [continuous="true|false"]]>
  [<description> text </description>]
  <condition [logic="AND|OR"]>
    text
    [<condition [logic="AND|OR"]> ... </condition>]
  </condition>

  <!-- Form 1 for the set element -->
  [<set name="text" value="number"
    [type="value|delta"]
    [action="step|ramp|exp" [tc="number"]]]/>

  <!-- Form 2 for the set element -->
  [<set name="text"
    [type="value|delta"]
    [action="step|ramp|exp" [tc="number"]]]>
    <function>
      ... <!-- Function definition --> ...
    </function>
  </set>

  [<delay> number </delay>]
  [<notify>
    [<property> name </property>]
    [ ... ]
  </notify>]
</event>

```

Ilustración 61 Detalle ejemplo genérico *Script*

En la Ilustración 61, se puede observar un ejemplo genérico de *Script* en la cual están recogido todas las opciones que se pueden aplicar. La característica *persistent* hará que una vez la condición se dé la primera vez, esta se seguirá aplicando todas las veces que se cumpla la condición si dicha variable se configura como True, como ya se adelantó anteriormente. La característica *continuous* hace que el evento se repita en cada paso del ciclo de simulación. Por defecto, los eventos solo se aplicarán la primera vez que se cumpla la condición de disparo.

Después se ve la característica condición, en la cual se puede configurar para que se cumplan varias o diferentes condiciones, mediante lógicas *AND* y *OR*. Además de esto se puede ver que se pueden configurar dichas condiciones en serie.

Las condiciones se pueden configurar tanto con símbolos matemáticos, como con símbolos, siendo estos los siguientes:

- EQ (*equal to* o igual o el símbolo =)
- NE (*not equal to* o desigual o el símbolo ~)
- LT (*less than* o menor de o el símbolo <)
- LE (*less than or equal to* o menor o igual que o el símbolo <=)
- GT (*greater than* o mayor que o el símbolo >)
- GE (*greater than or equal to* o mayor o igual que o el símbolo >=)

Posteriormente, se puede observar cómo se configurar los parámetros deseados. Estos pueden ser modificados mediante cambios en ellos valores, *value*, o un incremento desde el valor anterior, *delta*.

A parte de esto, se encuentran las acciones, que configuran como se lleva a cabo este cambio. Este valor o incremento puede ser aplicado mediante un escalón, *step*, la cual será la opción por defecto, una rampa, *ramp*,

o una función exponencial, *exp*. Para estas dos últimas se necesita configurar un tiempo de establecimiento regido por segundos, y configurado mediante el parámetro *tc*.

Dichas acciones se pueden configurar con un cierto retraso para el inicio de la aplicación de la acción, regido por la opción *delay*. Fijando los segundos deseados mediante este parámetro, se configurarán los segundos deseados a partir de los cuales empezará la acción.

Finalmente, se pueden utilizar los eventos simplemente para dar información, mediante la opción de *notify*. Con ella, se pueden especificar los valores que se escribirán por la consola de ejecución cuando se cumpla cierta condición.

3.2.4.1.2. Ejemplo práctico

Una vez introducido los elementos necesarios para ejecutar el modo bucle cerrado se va a poner como ejemplo un caso particular. Para ejecutar ‘JSBSim.exe’ primeramente habrá que compilarlo, ya sea en *C++* o en *Python*, o descargar el ejecutable directamente.

En este apartado se ejecutará el programa en el sistema operativo Windows, pero se puede hacer análogamente en distribuciones Linux si se quisiese.

Una vez se tenga el ejecutable, habrá que ir a la consola de Windows dentro de la carpeta contenedora del programa. Una vez en ella se puede ejecutar el programa escribiendo “./JSBSim”, lo cual nos dará información genérica de cómo usar el programa en este modo bucle cerrado, como se puede observar en la Ilustración 62.

Se puede observar en la Ilustración 62 que también se podría configurar la simulación en bucle cerrado desde la propia consola, además de configurar otra serie de opciones como podrían ser suspender la ejecución, marcar el ritmo de la simulación fijando la frecuencia o marcar la opción de que la ejecución sea en tiempo real del proceso, como ya se adelantó.

En este caso se cargará un script de un crucero para un *B737*, en cuyo archivo de configuración del avión está marcado una salida del tipo CSV. La orden de ejecución del programa se puede observar en la Ilustración 63.

```

Windows PowerShell
PS D:\Nueva carpeta\flightgear-autopilot-proyect\JSBSim> ./JSBSim
JSBSim version 1.0.0 Jun  3 2020 17:45:28
Usage: jsbsim [script file name] [output file names] <options>
options:
--help returns this message
--version returns the version number
--outputlogfile=<filename> sets (overrides) the name of a data output file
--logdirectivefile=<filename> specifies the name of a data logging directives file
                             (can appear multiple times)
--root=<path> specifies the JSBSim root directory (where aircraft/, engine/, etc. reside)
--aircraft=<filename> specifies the name of the aircraft to be modeled
--script=<filename> specifies a script to run
--realtime specifies to run in actual real world time
--nice specifies to run at lower CPU usage
--nohighlight specifies that console output should be pure text only (no color)
--suspend specifies to suspend the simulation after initialization
--initfile=<filename> specifies an initialization file
--catalog specifies that all properties for this aircraft model should be printed
                (catalog=aircraftname is an optional format)
--property=<name=value> e.g. --property=simulation/integrator/rate/rotational=1
--simulation-rate=<rate (double)> specifies the sim dt time or frequency
                If rate specified is less than 1, it is interpreted as
                a time step size, otherwise it is assumed to be a rate in Hertz.
--end=<time (double)> specifies the sim end time

NOTE: There can be no spaces around the = sign when
      an option is followed by a filename
PS D:\Nueva carpeta\flightgear-autopilot-proyect\JSBSim>

```

Ilustración 62 Detalle ejecución *JSBSim* consola Windows

```

PS D:\Nueva carpeta\flightgear-autopilot-proyect\JSBSim> ./JSBSim --script=scripts//3/_cruise

```

Ilustración 63 Detalle ejecución *JSBSim*

Como se observa en la Ilustración 64 y como ya se ha comentado anteriormente, lo primero que se hace es introducir los datos de la aeronave. Mientras esto sucede, el ejecutable saca por consola una serie de datos característicos a medida que se van cargando.

```
PS D:\Nueva carpeta\flightgear-autopilot-proyect\JSBSim> ./JSBSim --script=scripts/737_cruise

JSBSim Flight Dynamics Model v1.0.0 Jun  3 2020 17:45:28
[JSBSim-ML v2.0]

JSBSim startup beginning ...

Reading Aircraft Configuration File: 737
Version: 2.0

This aircraft model is a BETA release!!!

This aircraft model probably will not fly as expected.

Use this model for development purposes ONLY!!!

Description:   Models a Boeing 737.
Model Author:  Dave Culp
Creation Date: 2006-01-04
Version:       $Revision: 1.43 $

Aircraft Metrics:
WingArea: 1171.000000
WingSpan: 94.700000
Incidence: 0.000000
Chord: 12.310000
H. Tail Area: 348.000000
H. Tail Arm: 48.040000
V. Tail Area: 297.000000
V. Tail Arm: 44.500000
Eyepoint (x, y, z): 80.000000 , -30.000000 , 70.000000
Ref Pt (x, y, z): 625.000000 , 0.000000 , 24.000000
Visual Ref Pt (x, y, z): 0.000000 , 0.000000 , 0.000000

Mass and Balance:
baseIxx: 562000.000000 slug-ft2
baseIyy: 1473000.000000 slug-ft2
baseIzz: 1004000.000000 slug-ft2
```

Ilustración 64 Detalle ejecución de *JSBSim* en consola

```
Script: "Cruise flight in 737."
begins at 0.0000 seconds and runs to 100.0082 seconds with dt = 0.008333 (121.000000 Hz)

Local property: notify-time-trigger = 0.000000
Event 0 (Set engines running):
  When first triggered, executes once if all of the following are true: {
    sim-time-sec le 0.100000
  }
  Actions taken:
  {
    set propulsion/engine/set-running to 1.000000 (constant via step)
    set propulsion/engine[1]/set-running to 1.000000 (constant via step)
  }

Event 1 (Trim):
  When first triggered, executes once if all of the following are true: {
    sim-time-sec gt 0.100000
  }
  Actions taken (after a delay of 5.000000 secs):
  {
    set simulation/do_simple_trim to 1.000000 (constant via step)
  }
  Notifications:
  {
    n2
    n2
    thrust-lbs
    thrust-lbs
    vc-kts
    vc-fps
    vt-fps
    phi-rad
    theta-rad
    psi-rad
  }

Event 2 (Repeating Notify):
  Whenever triggered, executes once if all of the following are true: {
    sim-time-sec > notify-time-trigger
```

Ilustración 65 Detalle ejecución de *JSBSim* en consola, inicio del script

Si hay algún error de carga, estos aparecerían a continuación del dato que se ha tratado de introducir y serviría para depurar el script a ejecutar.

Posteriormente, el programa lee el script que tiene que ejecutar, para guardar los eventos y las condiciones que tiene que ir chequeando para que la simulación se ejecute sin fallos, como se puede observar en la Ilustración 59.

Una vez terminada dicha carga, el programa está preparado para iniciar la simulación dinámica del vehículo, solo quedando por inicializar los valores iniciales, como se puede ver en la Ilustración 60.

```

End of vehicle configuration loading.
-----
Simulation Configuration
-----
Mass Properties Model:
Ground Reactions Model:
Aerodynamics Model:
Propulsion Model:
-----
State Report at sim time: 0.000000 seconds
Position
ECI: -7573455.85416249, 12120062.90758432, 15325989.50994734 (x,y,z, in ft)
ECEf: -7573455.854162, 12120062.907584, 15325989.509947 (x,y,z, in ft)
Local: 47.191291, 122.000000, 30000.000000 (geodetic lat, lon, alt ASL in deg and ft)

Orientation
ECI: 146.5996313734458, 28.83217294227462, 68.18000934691847 (phi, theta, psi in deg)
Local: 3.566420668128901e-15, 4.112117579729561e-15, 225 (phi, theta, psi in deg)

Velocity
ECI: -639.5973933762564, 57.68999602017038, -361.684248868723 (x,y,z in ft/s)
ECEf: 244.2115319171359, 609.9551063799315, -361.6842488687231 (x,y,z in ft/s)
Local: -530.330086, -530.330086, -0.000000 (n,e,d in ft/sec)
Body: 750.000000, 0.000000, 0.000000 (u,v,w in ft/sec)

Body Rates (relative to given frame, expressed in body frame)
ECI: -0.002014858138982949, -3.575075025962312e-05, -0.001500715941292636 (p,q,r in deg/s)
ECEf: -3.882513038895894e-19, -0.002050608889242573, 0.001554934040960687 (p,q,r in deg/s)

---- JSBSim Execution beginning ... -----

Set engines running (Event 0) executed at time: 0.008333

Repeating Notify (Event 2) executed at time: 0.008333
    
```

Ilustración 66 Detalle ejecución JSBSim, inicialización

En la ilustración anterior se pueden ver los datos iniciales en diferentes tipos de ejes para caracterizar la posición de la aeronave. Se observan datos como la posición, orientación, velocidad e incluso las velocidades angulares iniciales.

	A	B	C	D	E
1	Time	propulsion/engine[0]/thrust-N	propulsion/engine[1]/thrust-N	/fdm/jsbsim/attitude/theta-rad	attitude/theta-deg
2	0	1686,930052	1686,930052	7,18E-17	4,11E-15
3	0,099996	1688,236822	1688,236822	-7,44E-05	-0,004261188
4	0,199992	1689,50302	1689,50302	-0,000327509	-0,018764868
5	0,299988	1690,731888	1690,731888	-0,000779393	-0,044655906
6	0,399984	1691,923428	1691,923428	-0,001442582	-0,082653887
7	0,49998	1693,077268	1693,077268	-0,002322774	-0,133085164
8	0,599976	1694,192728	1694,192728	-0,003419472	-0,195921323
9	0,699972	1695,268886	1695,268886	-0,004726746	-0,27082261
10	0,799968	1696,304638	1696,304638	-0,006234047	-0,35718457
11	0,899964	1697,298759	1697,298759	-0,007927045	-0,454186211
12	0,99996	1698,249954	1698,249954	-0,009788472	-0,560838141
13	1,099956	1699,156905	1699,156905	-0,011798937	-0,676029301
14	1,199952	1700,018309	1700,018309	-0,013937696	-0,798571133
15	1,299948	1700,832912	1700,832912	-0,01618336	-0,927238232
16	1,399944	1701,599531	1701,599531	-0,018514536	-1,060804792
17	1,49994	1702,317081	1702,317081	-0,020910377	-1,198076347
18	1,599936	1702,984588	1702,984588	-0,023351048	-1,337916475
19	1,699932	1703,601183	1703,601183	-0,025818108	-1,479268616
20	1,799928	1704,166119	1704,166119	-0,028294806	-1,621172951
21	1,899924	1704,67877	1704,67877	-0,030766289	-1,762778485
22	1,99992	1705,138632	1705,138632	-0,033219737	-1,903350709
23	2,099916	1705,545319	1705,545319	-0,035644428	-2,042275262

Ilustración 67 CSV resultante de la ejecución de JSBSim

Y a partir de este momento la simulación se ejecutará hasta el final prefijado, dando por consola los datos que se hayan configurado mediante los eventos.

En la Ilustración 67 estarían los datos resultantes de la ejecución del script, donde se mostrarán los datos que hayan sido fijados en los archivos de configuración. Estos datos ya estarían listos para hacer análisis de la actuación y testear diferentes tipos de casos mediante cambios en los archivos de configuración.

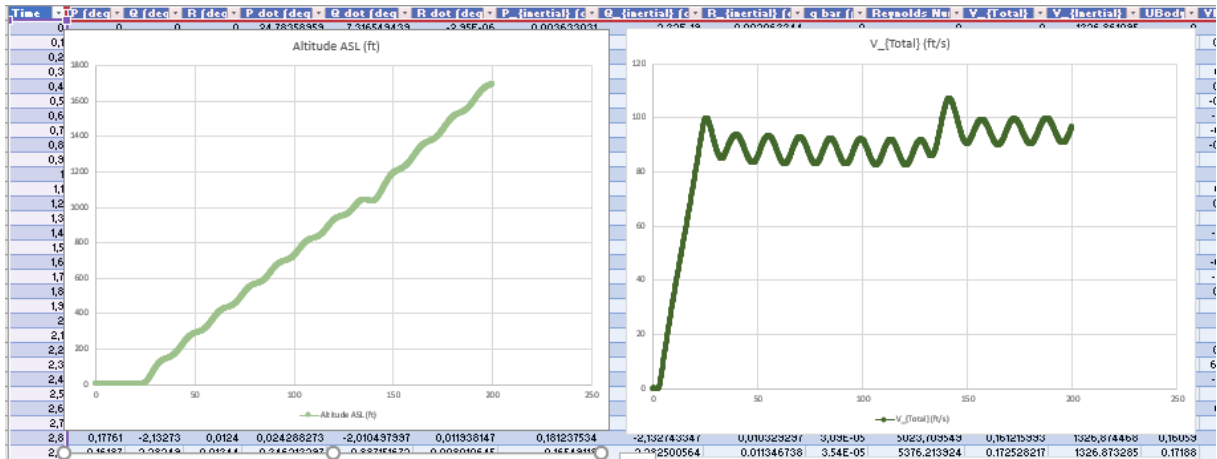


Ilustración 68 Detalle parámetros Altitude ASL (ft) y V_{total} (ft/s) respecto al tiempo obtenidos mediante JSBSim

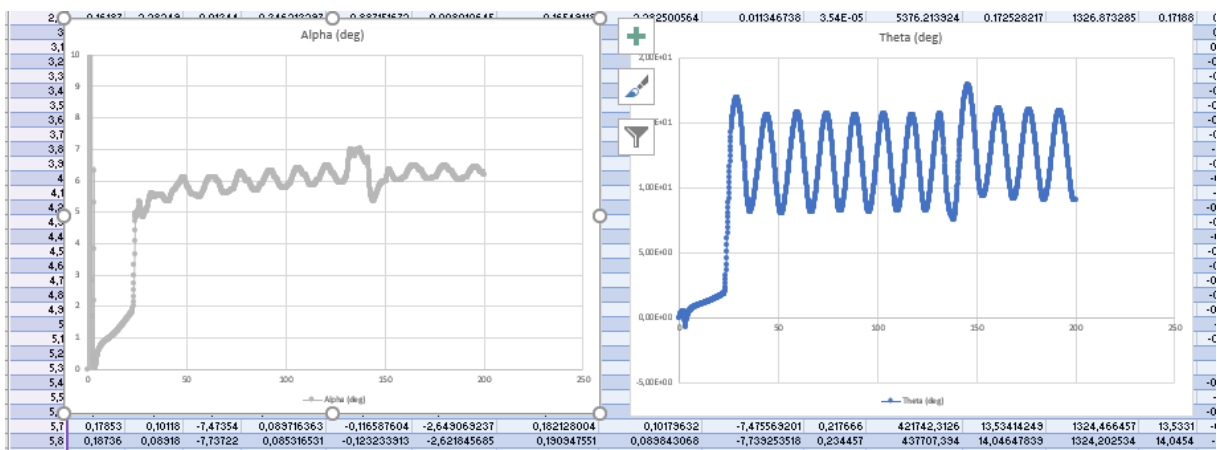


Ilustración 69 Detalle parámetros Alpha (deg) y Theta (deg) respecto al tiempo obtenidos mediante JSBSim

En las Ilustración 68 e Ilustración 69 se puede observar las gráficas resultantes de una simulación que se ha efectuado en JSBSim. En dicho caso se trata de un despegue, y se pueden observar la evolución respecto al tiempo de la altitud, del ángulo de ataque, de la velocidad, del ángulo theta, del ángulo phi y de la deflexión de los elevadores. Se puede observar tanto la potencia de la herramienta, ya no solo se han propagado datos dinámicos asociados a los 6 grados de libertad, sino también todos los parámetros al modelo aerodinámico y al modelo propulsivo.

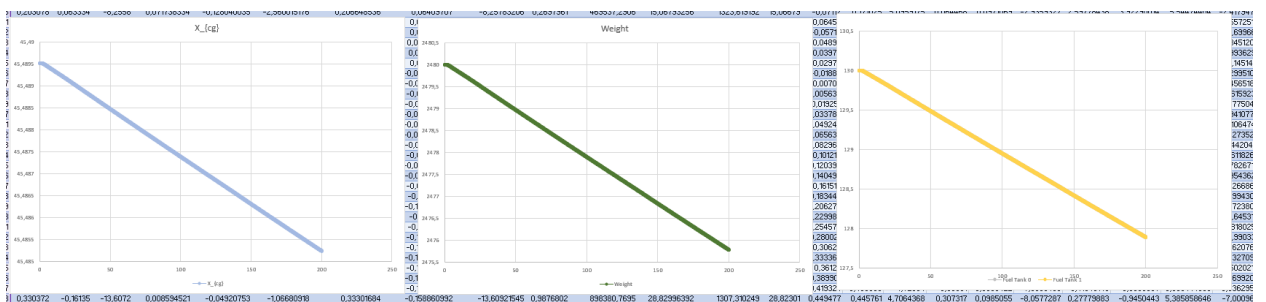


Ilustración 70 Detalle parámetros másicos obtenidos de JSBSim

En la Ilustración 70, se observa la propagación del centro de gravedad (eje x), del peso total y del peso de combustible (para ambos tanques) a lo largo de esta simulación. Así se observa cómo funciona el modelo de consumo en *JSBSim*.

3.2.4.2. Modo simulación tiempo real

Como se ha adelantado anteriormente, también se pueden utilizar *JSBSim* para hacer una simulación en tiempo real. Al contrario del modo *Standalone*, para ejecutar esta simulación hay que basarse en las librerías *JSBSim* y programar un código que las contenga, en otras palabras, no es solo activar un ejecutable. Esto requiere un conocimiento en profundidad de las librerías.

Existen dos lenguajes de programación en los cuales está implementado *JSBSim*, *C++* y *Python*. Aunque si bien es cierto que *C++* es un lenguaje más eficiente a la hora de la ejecución y de que esto es crítico para los simuladores, *Python* ofrece una programación más amigable y entendible.

Después de probar ambos métodos, con conocimientos básicos de ambos, se decide optar por *Python*, ya que por el carácter lectivo que se le quiere dar al proyecto es el que tiene la curva de aprendizaje más generosa y ofrece recursos más accesibles y de fácil implementación, como ya se verá posteriormente.

Antes de entrar en el código en sí, se introducirá la librería *pygame*, la cual ha hecho posible este prototipo.

3.2.4.2.1. *Pygame*, e introduciendo variables de control

Para realizar esta parte del proyecto, se necesitaba un modo de introducir las variables de control en el bucle de simulación. Aunque esto podría haber sido implementado usando las teclas del teclado, *Python* nos ofrece un amplio catálogo de librerías código libre que se pueden usar con diferentes objetivos.

Al igual que sucede en aviones reales y en función de la complejidad de estos, una posición de pedales, de los cuernos de mando o del *joystick* se transduce, directa o indirectamente según las leyes de vuelo, en una deflexión de una superficie aerodinámica. Estas entradas son continuas, y varían entre valores mínimo y máximo, pudiéndose seleccionar cualquier deflexión intermedia. Esto se puede observar en la Ilustración 44, en el apartado [3.2.2.8](#) en el cual se ve esa traducción de una señal de rango [-1,1] en una deflexión de una superficie.

Para ello, la forma más fácil de reproducir este comportamiento era utilizar un *joystick* para simular las entradas que actúen sobre el timón de profundidad y alerones, pero al no tener físicamente uno específico a la simulación aérea, se optó por un *gamepad*, o mando de consola en castellano. Ya que era para un prototipo, este ayuda al testeado sin hacer un desembolso extra, dando una sensación de control similar.

En este tipo de controladores existen dos tipos básicos de entradas, unas entradas progresivas que varían entre dos variables, usualmente -1 y 1, en función de la posición de unas palancas, los llamados *joystick*, y otros los llamados pulsadores o botones, los cuales tienen asociados dos posiciones, arriba y abajo.

Una vez se toma esta decisión, se deben captar y dar un valor a estas entradas del usuario y llevarlas a *JSBSim*, y es por ello que se implementó a través de *pygame*. *Pygame* es una librería *Python* centrada en captar señales provenientes de controladores y asociarles ciertos valores. Esto lo hace a través de eventos, los cuales tienen una serie de parámetros asociados para hacerlos únicos respecto a otros. Estos eventos pueden ser tanto provenientes de periféricos comunes, como el ratón de un ordenador, teclas del teclado, o provenientes de periféricos más específicos para la simulación.

Para explorar esta librería, se implementó un código específico, el cual está recogido en el anexo 3. En él, se explora esta librería y se muestran por pantalla los eventos generados al utilizar el mando. Se propone un juego muy sencillo, en el cual se muestra una ventana en blanco, función interna de esta librería, y se asocian eventos a una salida de texto descriptivo por pantalla, y en algunos casos al cambio de color de esta.

Primeramente, se inicializa el sistema y se precargan las funciones en la memoria. Posteriormente y al inicio de la función *main()*, se inicializa la pantalla y se le da forma, además de inicializar variables como una bandera que marca el fin de la ejecución, la inicialización del tiempo de juego o una lista vacía donde se

introdujeran los datos de los *joysticks*. Además, antes de iniciar el bucle de juego, se buscan y numeran todos los mandos que están conectados al ordenador, para poder distinguirlos entre si (esta opción estaría más enfocada al desarrollo de juegos multijugador). Todo esto se puede observar en la Ilustración 71.

```

1 import pygame
2 pygame.init()
3
4
5
6 def main():
7     screen = pygame.display.set_mode((640, 480))
8     pygame.display.set_caption("Joystick Testing")
9
10    background = pygame.Surface(screen.get_size())
11    background = background.convert()
12    background.fill((255, 255, 255))
13
14    joysticks = []
15    clock = pygame.time.Clock()
16    keepPlaying = True
17
18    # for al the connected joysticks
19    for i in range(0, pygame.joystick.get_count()):
20        # create an Joystick object in our list
21        joysticks.append(pygame.joystick.Joystick(i))
22        # initialize them all (-1 means loop forever)
23        joysticks[-1].init()
24        # print a statement telling what the name of the controller is
25        print(f"Detected joystick {joysticks[-1].get_name()}")

```

Ilustración 71 Detalle inicialización *Pygame*

En la Ilustración 72, se observa el texto de respuesta mostrado en la consola de *Python*, el cual ha detectado el controlador *Thrustmaster dual analog 3.2*.

```

Python 3.8.3 (default, Jul 2 2020, 17:30:36) [MSC v.1916 64 bit (AMD64)]
Type "copyright", "credits" or "license" for more information.

IPython 7.16.1 -- An enhanced Interactive Python.

In [1]: runfile('G:/portatil_31_08_2020/desktop/Pruebas_python/teclas_test.py', wdir='G:/
portatil_31_08_2020/desktop/Pruebas_python')
pygame 1.9.6
Hello from the pygame community. https://www.pygame.org/contribute.html
Detected joystick Thrustmaster dual analog 3.2

```

Ilustración 72 Detalle detección del mando

Una vez inicializado, el código ejecuta con una frecuencia de 60 Hz un código que muestra por pantalla los diferentes eventos que han sucedido. Por ejemplo, al pulsar un botón, se tendrán por pantalla los siguientes datos característicos mostrados en la Ilustración 73.

```

pygame 1.9.6
Hello from the pygame community. https://www.pygame.org/contribute.html
Detected joystick Thrustmaster dual analog 3.2
Joystick Thrustmaster dual analog 3.2 button 1 down.
<Event(10-JoyButtonDown {'joy': 0, 'button': 1})>
Joystick Thrustmaster dual analog 3.2 button 1 up.
<Event(11-JoyButtonUp {'joy': 0, 'button': 1})>

```

Ilustración 73 Detalle evento botón

Se puede ver como el evento nos dice que ha pasado (*button 1 down*), el mando utilizado (*'joy': 0*) y que botón ha disparado el evento (*'button': 1*). A estos eventos se le pueden asignar acciones de tipo lógica, por ejemplo, al encendido de luces, la deflexión de un tren de aterrizaje o el cambio de la posición de la palanca de *slat/flap*.

A parte de esto se tienen los movimientos de los *joysticks*, los cuales nos darán información similar al caso de los botones por pantalla, añadiendo la posición de la palanca, como se puede ver en la Ilustración 74.


```

<Event(7-JoyAxisMotion {'joy': 0, 'axis': 1, 'value': -0.8828394421216468})>
Joystick Thrustmaster dual analog 3.2 axis 1 motion.
-1.000030518509476
<Event(7-JoyAxisMotion {'joy': 0, 'axis': 1, 'value': -1.000030518509476})>
Joystick Thrustmaster dual analog 3.2 axis 1 motion.
-0.8750267036957915
<Event(7-JoyAxisMotion {'joy': 0, 'axis': 1, 'value': -0.8750267036957915})>
Joystick Thrustmaster dual analog 3.2 axis 1 motion.
-0.0390942106387524
<Event(7-JoyAxisMotion {'joy': 0, 'axis': 1, 'value': -0.0390942106387524})>
Joystick Thrustmaster dual analog 3.2 axis 1 motion.
-3.051850947599719e-05
<Event(7-JoyAxisMotion {'joy': 0, 'axis': 1, 'value': -3.051850947599719e-05})>

```

Ilustración 74 Detalle evento *joystick*

En la ilustración anterior se pueden distinguir este tipo de evento (*Joystick Thrustmaster dual analog 3.2 axis 1 motion*), y los datos característicos al evento. Además de los anteriores, se encuentra un nuevo parámetro 'value' el cual asignará un valor numérico en función de la posición de palanca. Se observa que ha habido un paso desde una posición tope (-1) hasta la posición neutra (-3e-5). El hecho que no sean números redondos es debido a una mala calibración de estos.

Este tipo de valores son muy útiles a la hora de introducir variables de control que gobiernan las superficies aerodinámicas, e incluso pudiéndose utilizar para otras variables como la configuración del trimado, siempre y cuando en vez de tener un *joystick* se tuviera una palanca estática.

Gracias a esta librería y este pequeño código, se exploraron todas las posibilidades y se pasó a la implementación del prototipo asociando estos eventos a parámetros de control internos de *JSBSIM*.

3.2.4.2.2. Prototipo simulador tiempo real

Una vez resuelto la problemática de la introducción de variables de control, se inició la programación del prototipo. El objetivo de este prototipo era ver cuan fácil es implementar con *JSBSim* una simulación en tiempo real de manera satisfactoria y realista, haciéndose un vuelo manual iniciado en un crucero para comprobar si se puede efectuar un control efectivo, que a priori no está depurado.

En este caso no hubo un ajuste entre el valor del *joystick* y el valor comandado de la deflexión, siendo una ley de control directa. Todo ello tenía como objetivo saber si era una herramienta potente desde la cual seguir implementando.

Para ello se implementó un bucle parecido al que se implementó para *pygame*, en el cual se tiene una fase preliminar de inicialización del sistema, y una segunda parte que consiste en un bucle infinito en el cual se propagará la simulación.

El paso inicial es el mismo que en el caso del modo bucle cerrado, y se debe ejecutar la clase un *FGFDMEx* para inicializar todas las funciones, y siguiendo con la carga del archivo de configuración de avión y el archivo de inicialización de la aeronave. Esto se introdujo con más profundidad en el apartado 4.2.1 [Introducción a JSBSim](#), en la Ilustración 28.

Como se ha visto anteriormente, se puede conectar *JSBSim* con *Flightgear* para utilizar este último como interfaz visual. Para llevar esto a cabo, es necesario abrir un *socket* de salida, como se observa en la Ilustración 75. Posteriormente se ejecutará *Flightgear* en la consola de Windows, usando la configuración mostrada en la Ilustración 76.

```

952 <output name="127.0.0.1" type="FLIGHTGEAR" protocol="UDP" port="5500" rate="60">
953 </output>

```

Ilustración 75 Detalle salida tipo *Flightgear*

```
Windows PowerShell
PS D:\Program Files\FlightGear 2018.3.6\bin> ./fgfs --fdm=null --native-fdm=socket,in,60,0,5500,udp --httpd=8080
```

Ilustración 76 Detalle sincronización *Flightgear* con *JSBSim*

En la ilustración anterior se observa que se ejecuta *Flightgear* usando la opción `--fdm=null`, lo cual hace que los datos dinámicos del propio programa queden desactivados y estos sean leído de una fuente externa. Para ello, será necesario un *socket* tipo *native*, donde *socket* indica el tipo de conexión, *in* si se trata de entrada o de salida, 60 los hertzios de entrega de datos, 0 estaría reservado a la dirección IP (donde 0 indica la dirección propia del ordenador 127.0.0.1), 5500 sería el puerto utilizado para el intercambio de información y UDP el tipo de conexión que se establecerá. Todo esto está explicado más en detalle en el apartado [3.2.2.9](#))

En este caso concreto, se ha tomado como aeronave la avioneta Cessna 172P, ya que es de los modelos más completos ofrecidos por la librería interna de *JSBSim*, además de ser una aeronave perfecta para la inicialización de pilotos.

Lo anteriormente descrito fue ejecutado sin mayores problemas, pudiéndose conectar el prototipo con *Flightgear*, donde este un *feedback* visual de las entradas, siendo un completo éxito. Se puede observar un *frame* de esta simulación la Ilustración 77.

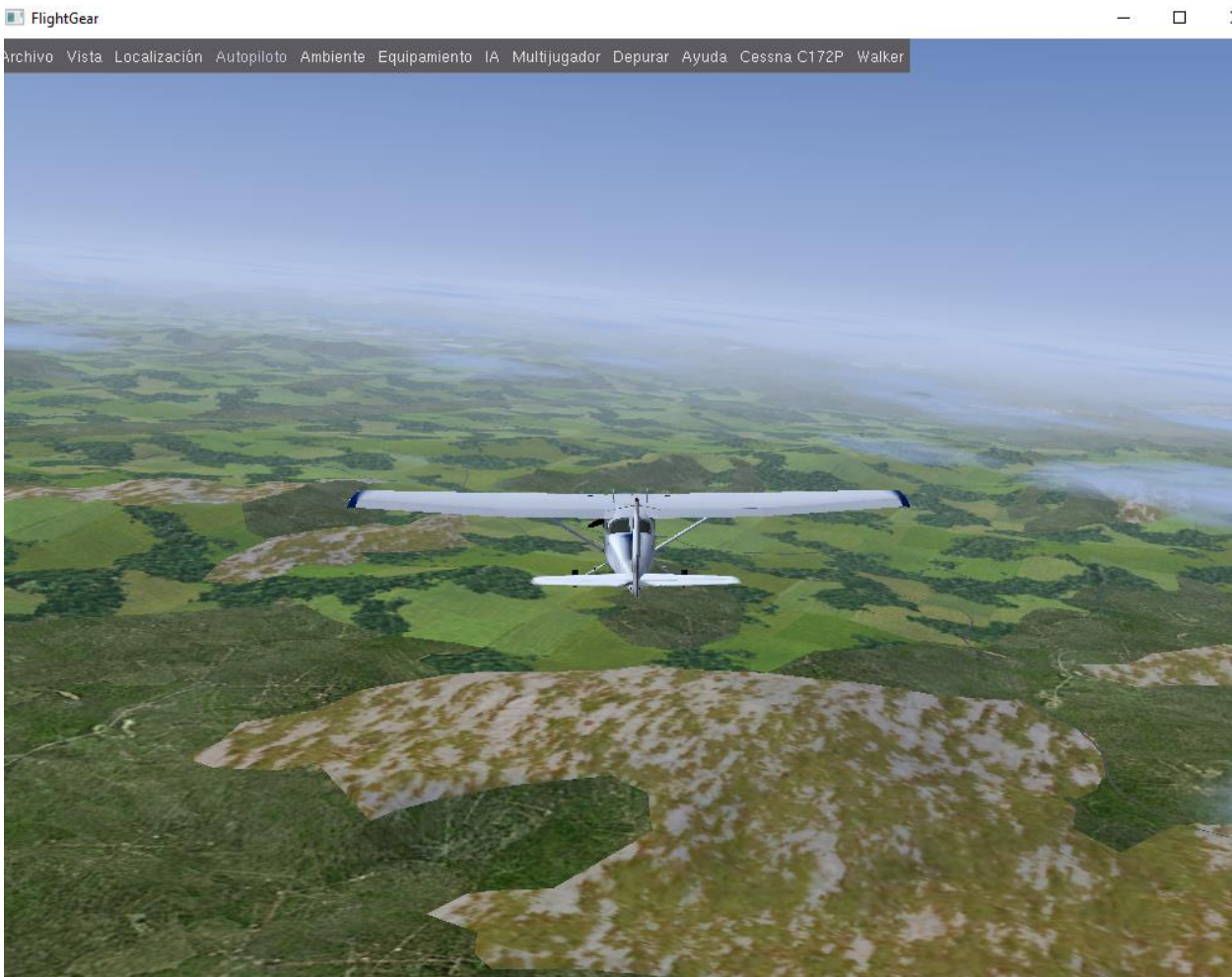


Ilustración 77 *Flightgear* usado como interfaz gráfica.

Además de introducir las condiciones dinámicas iniciales a partir de las cuales se empezará a propagar la simulación se deben introducir otros parámetros iniciales, más centrados en el estado interno de la aeronave, como pueden ser trimado de referencia, posición de los slat/flap, o la posición del tren de aterrizaje, en función de cómo este definido el archivo de configuración de avión y modelo propulsivo utilizado. Todo esto se puede

observar y partes anteriormente descritas se pueden observar en la Ilustración 78.

```

2   import pygame
3   # Importante para la simulacion dinamica
4   import jsbsim
5   # Tiempo de simulacion
6   import time
7   from controller import controller
8
9
10
11  # Inicializamos y creamos objeto joystick
12  pygame.init()
13  joysticks = []
14  # clock = pygame.time.Clock()
15  keepPlaying = 1
16
17  for i in range(0, pygame.joystick.get_count()):
18      # create an Joystick object in our list
19      joysticks.append(pygame.joystick.Joystick(i))
20      # initialize them all (-1 means loop forever)
21      joysticks[-1].init()
22      # print a statement telling what the name of the controller is
23      print(f"Detected joystick {joysticks[-1].get_name()}")
24
25
26
27  # Init jsbsim
28  fdm = jsbsim.FGFDMEExec('.', None)
29  fdm.load_model('c172p')
30  fdm.load_ic('reset00.xml', True)
31
32
33
34
35  fdm['propulsion/set-running'] = -1
36  fdm['ic/alpha-deg'] = 3.2758
37  fdm['ic/theta-deg'] = 3.2787
38  fdm['ic/psi-true-deg'] = 307
39  # fdm['gear/gear-cmd-norm'] = 0
40  fdm['ic/q-rad_sec'] = 0.0
41  fdm['ic/p-rad_sec'] = 0.0
42  fdm['ic/r-rad_sec'] = 0.0
43  fdm['ic/roc-fps'] = 0.00001
44  fdm['fcs/pitch-trim-cmd-norm'] = -0.2
45  fdm['fcs/roll-trim-cmd-norm'] = 0.0
46  fdm['fcs/yaw-trim-cmd-norm'] = 0.0
47  fdm['fcs/flap-cmd-norm'] = 0
48  fdm['fcs/elevator-cmd-norm'] = 0.0
49  fdm['fcs/roll-cmd-norm'] = 0.0
50  fdm['fcs/aileron-cmd-norm'] = 0.0
51  fdm['fcs/throttle-cmd-norm'][-1] = 0.8

```

Ilustración 78 Detalle inicialización prototipo simulación tiempo real

En una segunda parte del código se hace un bucle que mientras la bandera de ejecución no sea configurada como cero. Debido a que el tiempo de propagación no es el mismo para cada paso de iteración, al principio de cada bucle se calcula la diferencia entre el tiempo real de la simulación y el último instante en el cual fue propagada la dinámica, para estimar el número de veces que hay que iterar para llegar al tiempo real. Esto hace que siempre haya un retraso entre el tiempo de simulación y el tiempo real, pero tiende a cero a mayor potencia de computación.

Una vez se ha estimado el número de ciclos que se tiene que propagar la simulación, se comprueba que haya habido algún evento. Si efectivamente alguno ha sucedido, se actualiza el nuevo valor de este. Dentro de esta parte se podrían relacionar diferentes eventos con diversos parámetros de control, e incluso ejecutar leyes de control de vuelo, ofreciendo una gran versatilidad. Una vez hecho esto se propaga la dinámica, mediante la

función `fdm.run()`. La implementación final está plasmada en la Ilustración 79.

```

82     while keepPlaying:
83         # clock.tick(60)
84         current_seconds = time.time()
85         actual_elapsed_time = current_seconds - initial_seconds
86         sim_lag_time = actual_elapsed_time - fdm.get_sim_time()
87         iterations = int(sim_lag_time / frame_duration)
88
89
90     for _ in range(int(sim_lag_time / frame_duration)):
91         event = controller(joysticks)
92         if event != None:
93             if event.type == pygame.JOYAXISMOTION:
94                 if event.axis == 1:
95                     # elevator_cmd = float(event.value)*0.5 + 0.5
96                     elevator_cmd = float(event.value)
97                 elif event.axis == 2:
98                     throttle_cmd = float(event.value)*float(0.5) + 0.5
99                     # throttle_cmd = float(event.value)*2 + 0.5
100                    # throttle_cmd = float(event.value)
101                 elif event.axis == 0:
102                     # aileron_cmd = float(event.value)*0.5 + 0.5
103                     aileron_cmd = float(event.value)
104                 elif event.axis == 3:
105                     # roll_cmd = float(event.value)*0.5 + 0.5
106                     rudder_cmd = float(event.value)
107
108
109         # throttle_cmd = 1
110         #
111         fdm['fcs/elevator-cmd-norm'] = elevator_cmd
112         fdm['fcs/rudder-cmd-norm'] = rudder_cmd
113         fdm['fcs/aileron-cmd-norm'] = aileron_cmd
114         fdm['fcs/throttle-cmd-norm'][-1] = throttle_cmd
115         # print(elevator_cmd)
116         print(throttle_cmd)
117         # fdm['fcs/mixture-cmd-norm'][-1] = throttle_cmd
118         # print(fdm.get_aircraft())
119         # fdm['simulation/dt'] = 2
120         result = fdm.run()
121         print(int(result))
122         # fdm.output()
123         # print(result)
124         current_seconds = time.time()
125         actual_elapsed_time = current_seconds - initial_seconds

```

Ilustración 79 Detalle bucle de simulación *JSBSim* en *Python*

Una vez escrito el código y tras varios intentos la simulación fue un éxito. El retraso entre la simulación y el tiempo real es casi imperceptible, como se observa en el parámetro `sim_lag_time` en la Ilustración 80.

joysticks	list	1	[Joystick]
keepPlaying	int	1	1
result	bool	1	True
rudder_cmd	int	1	0
sim_lag_time	float	1	0.006667709350590911

Ilustración 80 Extracto datos de la simulación

El control basado en los *joysticks*, aun siendo rudimentarios debido a la ley directa implementada, son adecuados para comprobar el comportamiento de la aeronave de una forma intuitiva, gracias además a la ayuda visual que ofrece *Flightgear*.

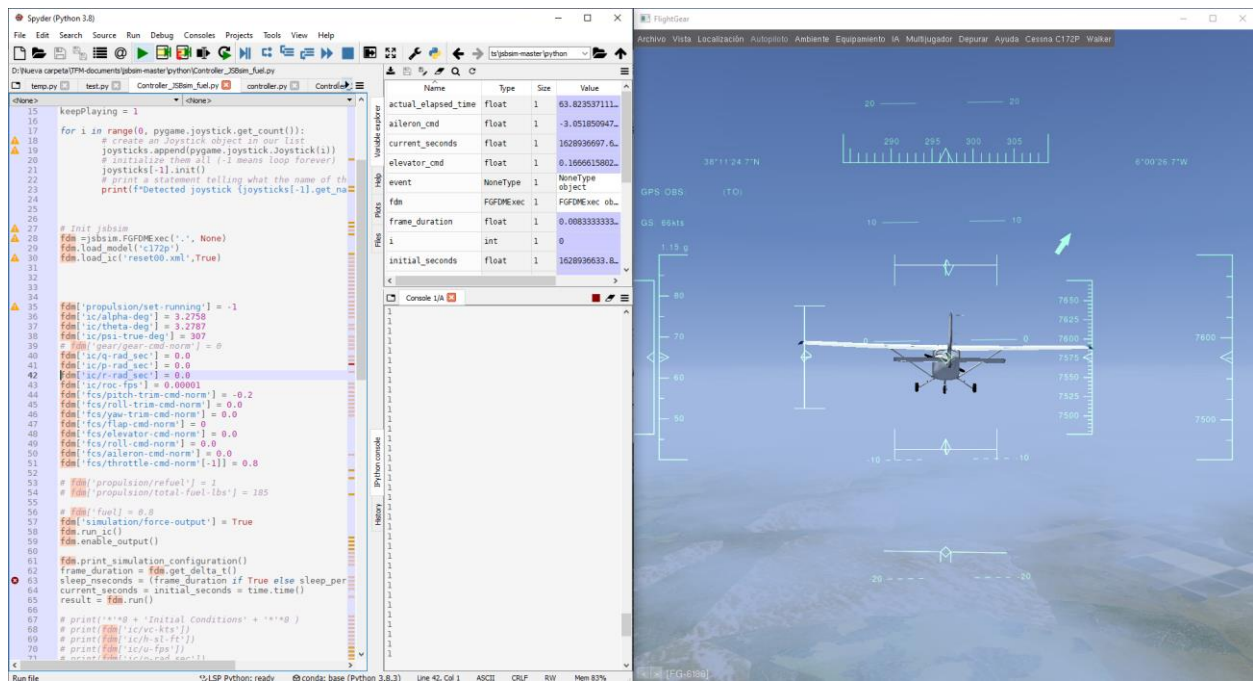


Ilustración 81 Detalle prototipo más interfaz gráfica

En la Ilustración 81 se puede observar tanto el código como la consola, además de *Flightgear* como interfaz gráfica. En la consola, se observan las ordenes comandadas provenientes del *gamepad*, pudiéndose así gobernar la aeronave.

3.2.5. Ventajas e inconvenientes

Una vez estudiado a fondo *JSBSim* y exploradas muchas de sus posibilidades, se va a analizar ventajas e inconvenientes encontrados para tomar una decisión en la continuidad por esta vía, o seguir investigando.

3.2.5.1. Ventajas

Código libre

Esta es una de las premisas del proyecto, y hace que la accesibilidad del proyecto sea mayor, pudiendo ser su uso escalable y sin problemas de licencias.

En el caso de *Python*, hace que se puedan utilizar múltiples librerías externas, como *pygame*, que ayudan a la implementación de nuevas funcionalidades, ya que es un lenguaje de programación muy activo y con una gran comunidad.

Librería robusta y flexible

Es una librería muy utilizada y con una comunidad bastante activa (sobretudo para el diseño y testeo de RPAS). La forma de codificar la información es intuitiva, y sin saber el lenguaje de programación se puede implementar fácilmente un archivo de configuración. Además, permite una total libertad a la hora de introducir variables de control, controladores, motores etc, siendo esto perfecto a la hora de introducirse en este mundo y tener múltiples referencias de como se implementan los diferentes modelos.

Versatilidad

El hecho que tenga dos posibles modos de funcionamiento la hace perfecta desde el punto de vista académico, sirviendo tanto para asignaturas cuyo objetivo sea el diseño de ciertos sistemas, como otras asignaturas mayormente centradas en el análisis de la dinámica y maniobras.

El modo *Standalone* sería perfecto a la hora de comprobar como pequeños cambios en el diseño o masa y

centrado pueden afectar a las diferentes maniobras, y el modo de simulación en tiempo real es perfecto a la hora de implementar diferentes sistemas de protección o leyes de control de vuelo.

Ligereza

Es una herramienta muy potente que se puede ejecutar rápidamente y consume pocos recursos, perfecta para la simulación aérea. Esta puede ser ejecutada desde incluso controladores potentes y así tener una gran accesibilidad.

3.2.5.2. Desventajas

Curva de aprendizaje lenta y documentación incompleta

Aunque posiblemente sea una herramienta perfecta para los objetivos del proyecto, la curva de aprendizaje puede ser frustrante al principio. Al estar ya todo hecho, se tiene que hacer una lectura en detalle del manual e incluso así, una vez en la implementación del código se debe seguir una estructura rígida para organizar la información.

Además, cada clase tiene sus propias variables y funciones que funcionan de maneras únicas, así que para poder implementarlos hay que hacer un estudio previo y posiblemente pasar un proceso de depuración del código. Eso se junta que a veces ciertas funcionalidades no están expuestas en el manual, y se requiere de ingeniería inversa para su entendimiento.

Instalación relativamente compleja

Para alguien que no esté familiarizado con el mundo de la programación, la instalación de las librerías puede ser frustrante, ya que depende de múltiples factores. La casuística varía entre que la versión de Python instalada no sea compatible con la última actualización de *JSBSim* y se tenga que investigar, hasta fallos propios de la configuración del ordenador anfitrión. Por ello, se debe hacer una lectura en profundidad de los pasos a realizar.

Modelos complejos

Para entender los modelos se debe hacer un estudio previo, por lo que hace que la curva de aprendizaje sea lenta y requiera de experiencia previa para obtener todo el potencial de la herramienta.

3.2.6. Conclusiones

JSBSim es una herramienta perfecta para la simulación dinámica de aeronaves, ya que muchas de las funcionalidades que, por ejemplo, se codificaron para este proyecto en *Matlab/Simulink*, están ya implementadas, con la diferencia que ya han pasado por muchas revisiones y diversos proyectos, por lo que se tiene una herramienta robusta y eficientes desde el punto de partida.

La manera que tiene de organizar la información es muy útil, ya que permite caracterizar las aeronaves de una forma sencilla y reproducible, pudiéndose almacenar la información y utilizarse posteriormente cuando se necesite. Además, consta de una base de datos bastante amplia, llena de referencias y ejemplos que pueden ser utilizados a la hora de diseñar proyectos propios.

El modo *Standalone* da la capacidad de poder diseñar y testar una aeronave de forma muy sencilla gracias a los archivos de configuración, pudiéndose introducir en la docencia de una forma simple. Esto permite orientar la docencia no solamente a la resolución de problemas teóricos, muchas veces alejados del mundo real, sino a abordar maniobras reales e implementar sistemas sencillos.

Además, gracias a un código relativamente sencillo, se puede interactuar directamente con el bucle de simulación, añadiendo datos entrantes del *gamepad*. Este precedente implica que se pueden utilizar réplicas de mandos o panalería más cercanas a la realidad, y con todas estas entradas externas, se abre la posibilidad de implementar diseño de leyes de control de vuelo, malfunctions, protecciones o modos de autopiloto mediante código *Python*.

La única pega que se le encontró a la plataforma es que no contaba con una interfaz externa con la cual interactuar, lo que en el apartado [1.3](#) se denominó como estación. Para suplir esta deficiencia, se utilizó

Flightgear, un simulador ya implementado, introduciéndose datos provenientes de la simulación que ya se estaban corriendo en el código *Python*.

Por ello, y aun estando satisfechos con los resultados de mostrados en esta librería, se decide estudiar más en profundidad *Flightgear* para poder explorar sus posibilidades, con el fin de poder implementar el simulador más completo con los recursos disponibles.

3.3. *Flightgear*

3.3.1. Introducción a *Flightgear*

Realmente *Flightgear* es una herramienta que se conocía previamente al estudio de *JSBSim*, a la cual no se quería recurrir para poder tener mayor rango de actuación y poder desarrollar el proyecto con mayor libertad. Sin embargo, una vez visto el coste y el conocimiento necesario para desarrollar una estación que acompañe la simulación, se decidió estudiar la viabilidad de este software en términos de simulador académico. Cabe decir que su uso ya era extendido, pudiéndose vincular con programas como *Matlab/Simulink* y habiéndose usado previamente para otros proyectos dentro de la Universidad de Sevilla.

Este es un simulador de vuelo, en principio para ordenadores personales, desarrollado en código libre. Está enfocado para ser una herramienta para entornos académicos y de investigación, a la par que para la instrucción de pilotos o incluso la industria, abierto a la colaboración de cualquier persona que quiera/pueda ayudar. Este ha sido desarrollado desde 1996, por voluntarios a lo largo de años de trabajo, fruto de una falta de funcionalidades de los simuladores comerciales. Estos eran bastante rígidos, no pudiéndose adentrarse en los sistemas y módulos que lo forman, y suelen estar desarrollados con objetivos concretos, careciendo de la posibilidad de expandirse una vez cerrados. Además, el hecho que este desarrollado por voluntarios, crea una comunidad viva, capaz de ayudarse mutuamente y exponer proyectos y dudas.

Bajo esta premisa se ha estado desarrollando un simulador que permite la simulación de un amplio rango de aeronaves, desde planeadores hasta helicópteros, así como aviones comerciales o cazas. Para ellos, se usan hasta tres modelos dinámicos diferentes, siendo estos los que dictan los movimientos que hacen los modelos 3D de las aeronaves, que por sí solas no son más que un conjunto de polígonos. Estos son:

- *JSBSim*: Modelo anteriormente comentado, y él que es usado por defecto
- *YASim*: otra opción en *Flightgear* y la cual destaca por utilizar la propia geometría de la aeronave para estimar las derivadas de estabilidad, muy útil para entornos académicos y diseños preliminares.
- *UIUC* otra opción en *Flightgear* y destacada por su simulación más compleja, nacido para simulaciones de eventos de englamamiento, y de una simulación más precisa de la entrada en pérdida.

Además de esto, se pueden introducir modelos dinámicos mediante programas externos, como es el caso introducido anteriormente (*Matlab/Simulink*), y este consta de otros modelos más específicos para casos especiales, como es el caso de aeronaves menos densas que el aire.

A diferencia de los programas comerciales, *Flightgear* es el resultado de una colección de código, que requiere de diferentes librerías para poder ser compilado. La principal es *SimGear*, que es el motor de simulación principal. Además, esta *TerraGear*, la cual es el programa predeterminado de datos de terreno, *OpenAL* que es utilizado para como software de sonido/audio, y *PLIB* o *OpenSceneGraph* son utilizados para la representación de gráficos 3D.

El principal atractivo que tiene *Flightgear* es la librería de modelos 3D de aeronave ya disponibles (con sus respectivos modelos dinámicos enfocados a *JSBSim* o *YASim*) o los escenarios terrestres. Cabe destacar de estos últimos que, al estar hechos por voluntarios, no todos ellos están desarrollados con el mismo nivel de detalle, destacando sobretodo escenarios que toman lugar en los Estados Unidos de América.

Aún así, aunque haya lugares que antojen falta de fidelidad a la hora de representar fielmente ciudades o aeropuertos, tantos las pistas como los accidentes geograficos tienen la suficiente fidelidad, y se pueden obtener para casi cualquier punto del globo terrestre, como se observa en la Ilustración 82.

Otro de los grandes atractivos que ofrece *Flightgear* es su protocolo multijugador, el cual permite el envío y carga de datos de otras entidades para la carga en tiempo real de estas, pudiendo haber varias aeronaves corriendo a la vez en la misma simulación. Primeramente esto fue implementado solo para red local, y posteriormente ampliado para lanzarlo en internet. Esto es un escenario realmente interesante para poder simular situaciones donde el control aéreo es necesario.

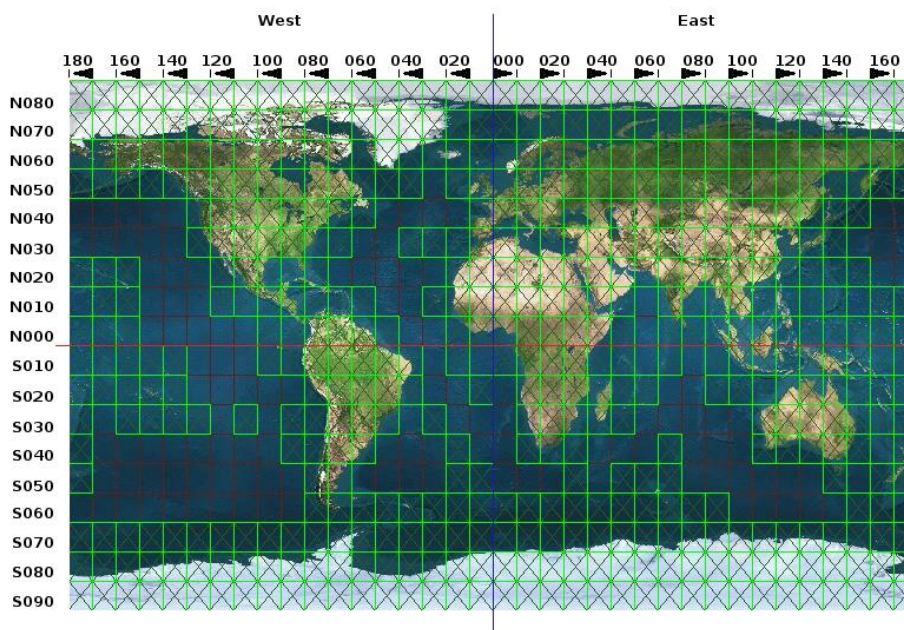


Ilustración 82 Interfaz de descarga de escenarios

Cabe destacar que de todos los aviones incluidos en las librerías de *Flightgear*, el avión por defecto y de referencia es la Cessna 172, la cual será utilizada en toda esta sección.

El que sea un proyecto bastante robusto a día de hoy ha sido gracia al interés que ha despertado por parte de la comunidad académica e industrial, que ha usado esta herramienta en multitud de diferentes proyectos. Algunos de ellos son:

- La creación de un *Full Flight Simulator* para investigación y entrenamiento, por parte de la *Università di Napoli*
- La creación de un simulador del 737NG con cabina tamaño real por parte de *NASA/Ames Human Centered System Lab*
- El uso extensivo de simuladores para entrenamiento por parte de la empresa *Simuladores Guarani*
- El estudio de las calidades de vuelo de un avión con forma de caja con alas usando *Flightgear* junto a *JSBSim*, por parte de la *HAW Hamburg*

Existe tantos otros, pero se han escogido estos por interés propio de este proyecto. En todos ellos, se ha utilizado esta herramienta con un enfoque de entrenamiento, y solo ha variado los medios utilizados para su implementación. Teniendo en cuenta todo lo mostrado, se puede considerar *Flightgear* como un candidato por derecho propio a ser usado como base para este proyecto.

3.3.2. Modos de lanzamiento

Hay dos formas para lanzar la simulación basada en *Flightgear*, teniendo como referencia la versión utilizada para este estudio, la cual es *Flightgear 2018.3.6*, y que a partir de ahora será la versión de referencia para este proyecto. La primera forma es mediante el denominado *Launcher*, más enfocada a un uso recreativo de la herramienta, y la segunda de ellas es mediante comandos desde la consola, desde la cual se pueden configurar diversas funcionalidades y parámetros más específicos.

En la Ilustración 83 se muestra el *Launcher*. Este ha cambiado a lo largo de las diferentes versiones con las que ha contado *Flightgear*. Para dar un ejemplo concreto, simplemente se profundizará sobre la versión de referencia de este proyecto.

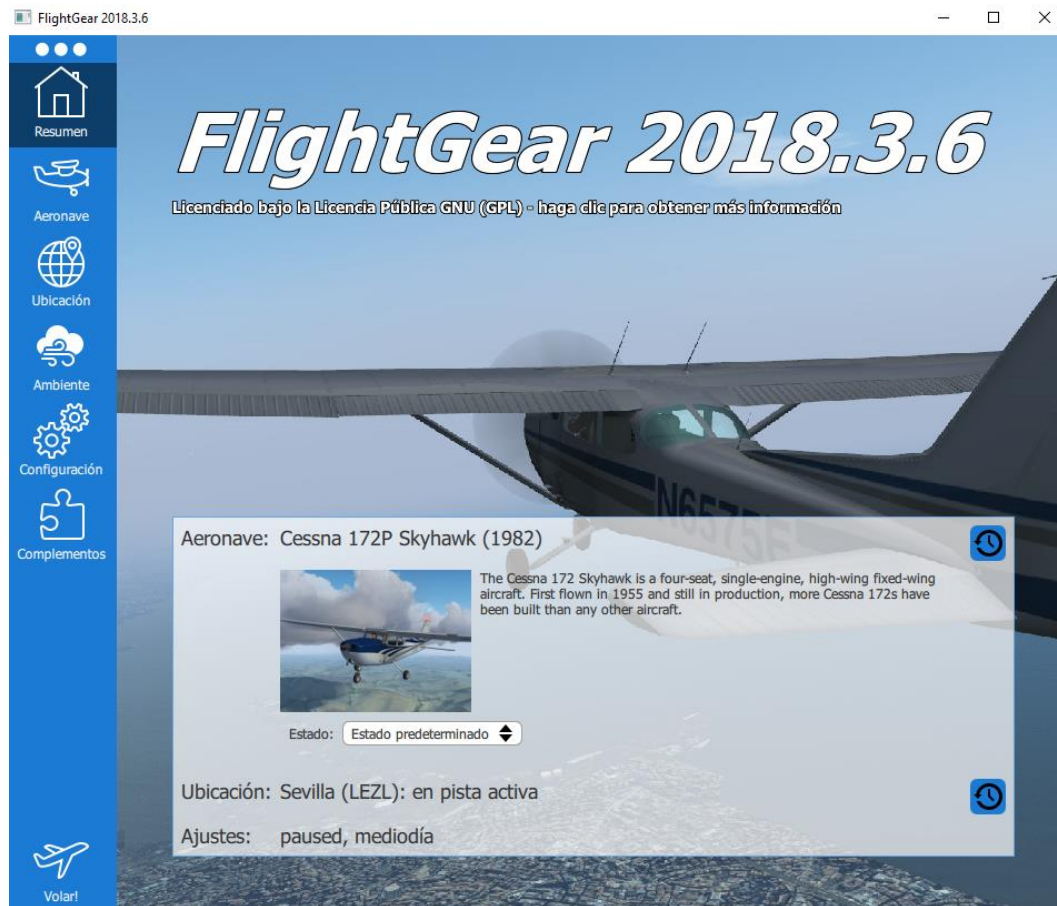


Ilustración 83 Detalle Launcher *Flightgear 2018.3.6*

En la ilustración anterior se puede observar varias opciones a la hora de iniciar una simulación. La primera de ellas sería el resumen, la cual da información general de las características de la simulación que se quiere lanzar, tales como la aeronave, la ubicación y los ajustes generales seleccionados.

Posteriormente se encuentra la opción de aeronave, desde la cual se seleccionará la aeronave deseada para la simulación. Esta opción no solo variara el modelo 3D que se lanzara, sino que cambiara así mismo el modelo a utilizar.

Después toma lugar la opción de la ubicación, la cual permite seleccionar el lugar desde el cual se quiere lanzar la simulación, los cuales suelen tomar como referencia un aeropuerto en concreto. Si se desea volar en una localización específica, se deberán recurrir a la latitud y longitud para seleccionar el punto.

Se puede observar que la siguiente opción disponible es el ambiente, en la cual se configura tanto la estación del año como la hora a la que se quiere lanzar el vuelo, hasta el tiempo atmosférico con el que contará la simulación. Esto último es bastante interesante ya que nos ofrece un modelo básico del tiempo atmosférico común para todos los puntos, o uno detallado basado en el terreno local. Además de esto se puede seleccionar tanto el clima en tiempo real, para el cual se necesita acceso a la red, o un escenario predefinido (anticiclón, tormenta, CAT IIIb, niebla de madrugada, etc), muy útil para la familiarización con un aeropuerto en concreto o con una situación en particular.

A continuación, se tiene la opción de configuración, desde la cual se puede seleccionar ajustes de visualización para optimizar el rendimiento, la descarga automática de escenario cuando haya nuevas versiones o al introducirse en terreno nuevo, o activar el modo multijugador, la cual es la más importante. En ella se seleccionará la matrícula de la aeronave que se utilizara, y por la cual el resto de jugadores (servicios de ATC y otros pilotos) se referirán a la aeronave, además del servidor al cual se desea conectar.

Una vez configurado todas estas opciones, se podrá lanzar la simulación seleccionando la opción volar, habiendo seleccionado en el caso del ejemplo, una Cessna 172P localizada sobre el aeropuerto de Honolulu

(PHNL).



Ilustración 84 Detalle programa lanzado *Flightgear*

Como se observa en la Ilustración 84, *Flightgear* ofrece un modelo 3D de la aeronave, con el cual se puede interactuar en tiempo real, no solo mediante el teclado o *joystick*, sino con el propio ratón. Esto hace posible la manipulación de la panalería, introducir nuevos valores, o seleccionar entre varios niveles de palanca. No solo eso, sino que este ofrece una instrumentación totalmente funcional, la cual da referencia de actitud, posición, nivel de combustible, etc.

La otra forma de lanzar la simulación es mediante comandos en la consola de Windows (o en la que proceda, ya que está disponible tanto en distribuciones Linux como Mac OS C). En el caso de Windows, que es el sistema operativo de referencia para este proyecto, se deberá abrir una consola en la carpeta contenedora del programa, y una vez que está abierta ejecutar el comando “./fgfs”, lo cual ya fue mostrado anteriormente en la Ilustración 76, y de nuevo en la Ilustración 85.

Como se introdujo anteriormente, lo interesante de esta opción es la capacidad para configurar opciones *Flightgear* de manera más profunda y más orientadas funciones auxiliares como a la conectividad del programa con programas o aplicaciones de terceros que complementen la experiencia.

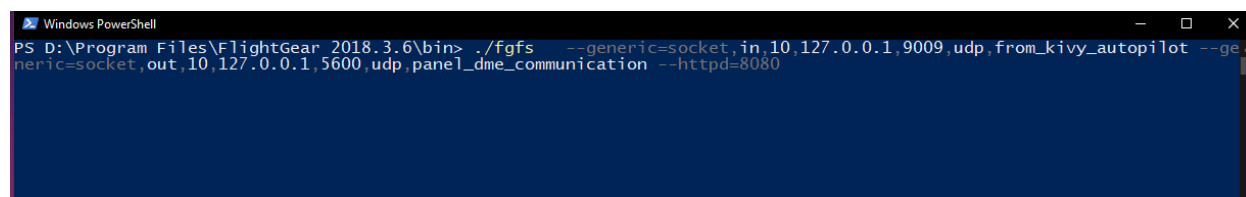


Ilustración 85 Detalle ejemplo ejecución *Flightgear* desde consola Windows

3.3.3. *Property tree* y herramientas auxiliares

Como se avanzó en el apartado anterior, al igual que sucedía con *JSBSim*, *Flightgear* tiene la capacidad de conectarse a internet o redes locales. Para ello, son utilizados diversos protocolos de comunicaciones, permitiendo crear un entorno más rico en dispositivos, como lo pueden ser un panel de instrumentos o una estación de instructor, como se verá posteriormente.

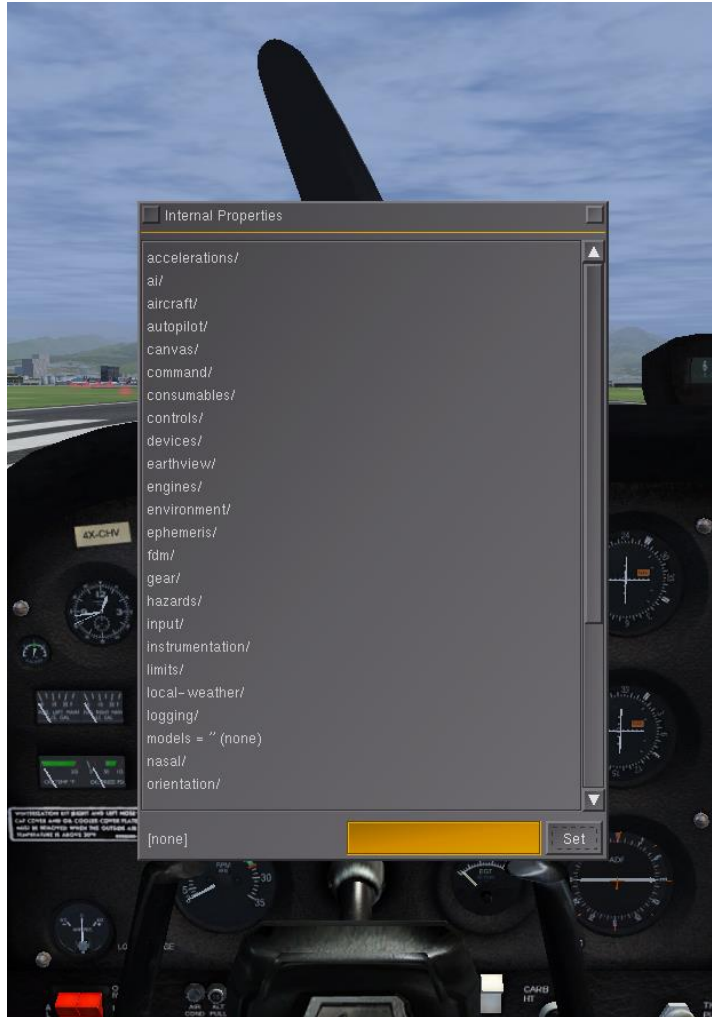


Ilustración 86 Detalle árbol de propiedades

Esto es posible gracias a la estructura interna de *Flightgear* y su espina dorsal, el *Property Tree*, o árbol de propiedades en castellano. Es una estructura en forma de árbol donde están albergados todos los parámetros necesarios para la simulación, y está más orientado al desarrollo que a la experiencia de vuelo. Al contener todos los parámetros importantes para la simulación, los diferentes sistemas pueden leer y escribir desde él, siendo un punto de referencia.

Para poner en contexto, en la Ilustración 86 se puede observar el visor interno de dicha estructura gracias a una interfaz integrada dentro de *Flightgear*. Se puede observar que lo primero que se tiene son las categorías principales de las cuales se abrirán otras subcategorías más específicas, recogiendo la información de sistemas concretos. Como ejemplo, se puede observar que existe la categoría *fdm* de la cual nacen otras tres categorías *fdm/ai-wake*, *fdm/jsbsim* y *fdm/trim*, como se puede observar en la Ilustración 87.

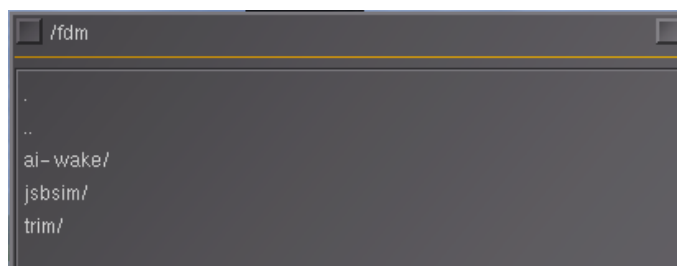


Ilustración 87 Detalle árbol de propiedades fdm/

Si se adentra en la opción *JSBSim*, podremos ver todos los datos salientes de este modelo dinámico de vuelo. Esto se muestra en la Ilustración 88.

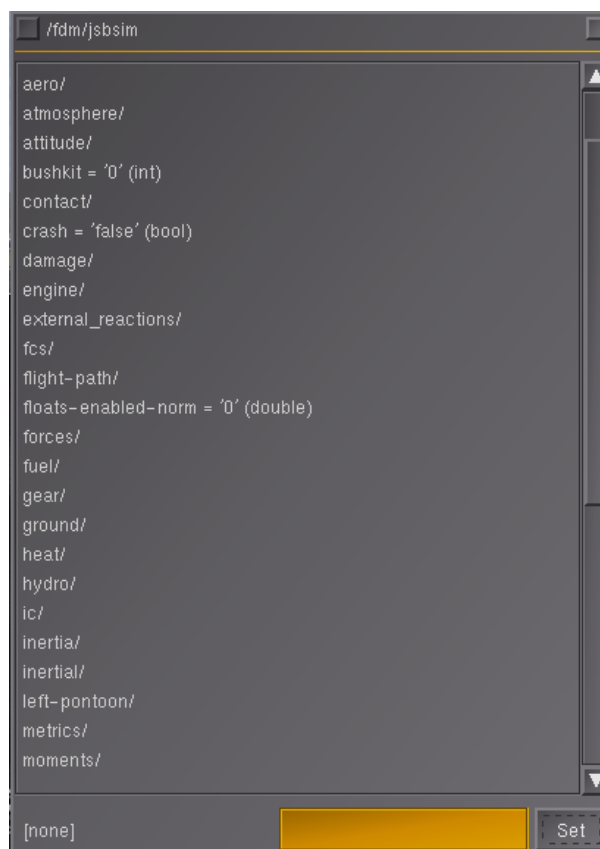


Ilustración 88 Detalle árbol de propiedades fdm/jsbsim

Con una estructura que va de lo más general a lo más concreto, este árbol de propiedades permite guardar toda la información en un mismo sitio, lo cual es muy útil a la hora de estructurar herramientas externas.

Flightgear permite la conexión a este árbol de propiedades mediante diferentes protocolos de comunicaciones, como puede ser mediante el *WebServer* (protocolo HTTPD) o el llamado *TelnetServer* (a través de *sockets* como ya se ha introducido en [3.2.2.9](#)). En la Ilustración 85 se puede observar un ejemplo de cómo ejecutar *Flightgear* haciendo configurado estas vías de entrada y salida.

Esta es la manera que tienen programas como *Matlab/Simulink* o el prototipo *Python* que se implementó en este proyecto de utilizar *Flightgear* como interfaz gráfica. Simplemente *Flightgear* permite la escritura y lectura desde programas externos, pudiéndose así ser implementadas diferentes herramientas externas.

Basado en esto, se han encontrado tres de las herramientas auxiliares del entorno *Flightgear* que pueden conformar una estación, o interfaz de usuario, más centrado en el propio uso del simulador. Destacar que dos de estas herramientas están enfocadas a un avión en particular, la *Cesna 172P*, y que pueden ser adaptadas a otros aviones configurando las direcciones en el árbol de propiedades que se deben leer y escribir.

3.3.3.1. c172p-WebPanel

Webpanel es una herramienta que utiliza un navegador web para cargar una réplica virtual del panel de instrumentos, que a través del protocolo HTTP es capaz de leer los parámetros desde el árbol de propiedades y así representarlos. En la Ilustración 89 se puede observar como los instrumentos propios del modelo 3D de *Flightgear* y los representados en el *WebPanel* están en fase.



Ilustración 89 Detalle c172p-WebPanel

Esta solución está basada en *HTML/JavaScript/CSS*, siguiendo una estructura similar a la de una página web corriente. Esto permite atacar al árbol de propiedades desde una red local, por lo que se puede ejecutar este panel de instrumentos desde otro dispositivo, como podría ser una raspberry o un teléfono móvil, simplemente mediante la conexión a la dirección IP del dispositivo anfitrión. Esto elimina problemas de compatibilidad y de configuración entre diferentes dispositivos, además de liberar al ordenador principal de la potencia de cálculo que supone este proceso.

En la Ilustración 90 se puede observar el código principal en HTML de este *c172p-WebPanel*. De la línea 42 a 66 se configuran los diferentes instrumentos que se utilizarán en el panel, organizados por filas. Cada uno de estos instrumentos tienen asociados una imagen vectorizada y un trozo de código en el cual se define las animaciones, siendo estos los archivos .json.

```

1 <!DOCTYPE html>
2 <html>
3 <head>
4 <meta charset=utf-8 />
5 <!-- Tweaks for panel on an ipad - from https://gist.github.com/tfausak/2222823 -->
6 <meta name="apple-mobile-web-app-capable" content="yes">
7 <!-- meta name="viewport" content="initial-scale=1.0, maximum-scale=1.0, minimum-scale=1.0" -->
8 <meta name="apple-touch-fullscreen" content="yes">
9 <meta name="apple-mobile-web-app-status-bar-style" content="black">
10
11 <title>FlightGear - Cessna 172 Instrument Panel</title>
12 <style type="text/css" media="screen">
13 body.html {
14     width: 100%;
15     height: 100%;
16     overflow: hidden;
17     background-color: #000000;
18     border: 0;
19     margin: 0;
20     padding: 0;
21 }
22
23 .instrument {
24     border: 1px solid #131313;
25     padding: 8px;
26     background-color: #111111;
27 }
28 </style>
29
30 <script type="text/javascript" charset="utf-8"
31     src="/3rdparty/jquery/jquery-1.11.1.min.js"></script>
32
33 <!-- Latest compiled and minified CSS -->
34 <link rel="stylesheet" href="bootstrap/bootstrap.min.css">
35 <!-- Latest compiled and minified JavaScript -->
36 <script src="bootstrap/bootstrap.min.js"></script>
37
38 <script type="text/javascript" charset="utf-8" src="/lib/jquery.fganimate.js"></script>
39 <script type="text/javascript" charset="utf-8" src="/lib/fgfs.js"></script>
40
41 </head>
42 <body data-fgpanel="true" data-fgpanel-props="c172p-webpanel-properties.json">
43 <div class="row">
44 <div id="ASI" class="instrument col-xs-2" data-fgpanel-instrument="ASI.json"></div>
45 <div id="AI" class="instrument col-xs-2" data-fgpanel-instrument="AI.json"></div>
46 <div id="ALT" class="instrument col-xs-2" data-fgpanel-instrument="ALT.json"></div>
47 <div id="VOR1" class="instrument col-xs-2" data-fgpanel-instrument="VOR1.json"></div>
48 <div class="instrument col-xs-2" data-fgpanel-instrument="Empty.json"></div>
49 <div class="instrument col-xs-2" data-fgpanel-instrument="Empty.json"></div>
50 </div>
51 <div class="row">
52 <div id="TC" class="instrument col-xs-2" data-fgpanel-instrument="TC.json"></div>
53 <div id="DG" class="instrument col-xs-2" data-fgpanel-instrument="DG.json"></div>
54 <div id="VSI" class="instrument col-xs-2" data-fgpanel-instrument="VSI.json"></div>
55 <div id="VOR2" class="instrument col-xs-2" data-fgpanel-instrument="VOR2.json"></div>
56 <div class="instrument col-xs-2" data-fgpanel-instrument="Empty.json"></div>
57 <div class="instrument col-xs-2" data-fgpanel-instrument="Empty.json"></div>
58 </div>
59 <div class="row">
60 <div id="RPM" class="instrument col-xs-2" data-fgpanel-instrument="RPM.json"></div>
61 <div class="instrument col-xs-2" data-fgpanel-instrument="Empty.json"></div>
62 <div id="EGT" class="instrument col-xs-2" data-fgpanel-instrument="EGT.json"></div>
63 <div id="ADF" class="instrument col-xs-2" data-fgpanel-instrument="ADF.json"></div>
64 <div class="instrument col-xs-2" data-fgpanel-instrument="Empty.json"></div>
65 <div class="instrument col-xs-2" data-fgpanel-instrument="Empty.json"></div>
66 </div>
67 <div class="row">
68 <div class="instrument col-xs-12" data-fgpanel-instrument="Empty.json"></div>
69 </div>
70 </body>
71 </html>

```

Ilustración 90 Detalle código HTML *c172p-WebPanel*

3.3.3.2. Phi – The FlightGear User Interface o panel de instructor

Posiblemente una de las herramientas más interesantes que alberga *Flightgear*. *Phi* es una aplicación desarrollada en HTML/JavaScript/CSS que hace de interfaz de usuario y de estación de instructor. Cabe destacar que, al contrario que la aplicación anterior la cual solo interactuaba pasivamente con *Flightgear* leyendo datos de la estructura de árbol, *Phi* es capaz de leer y escribir estos parámetros a través de varias interfaces. A través de esta aplicación la cual se ejecuta sobre un navegador web como ya pasaba con *WebPanel*, se puede configurar varias características del vuelo como se puede observar en la Ilustración 91.



Ilustración 91 Detalle *Flightgear* ejecutándose con *Phi*

Las funcionalidades disponibles para la versión en cuestión son varias, pero de las cuales destacan el poder congelar o continuar la simulación remotamente, la configuración del avión, la masa y centrado del avión, o el mapa.

La opción *Map* es la opción principal desde la cual se puede consultar los datos de navegación, observar el tráfico alrededor, se puede relocalizar la aeronave (arrastrando el avión a la posición deseada del mapa). Además, se puede consultar la traza de la aeronave y consultar parámetros tales como la velocidad de vuelo y la altitud.

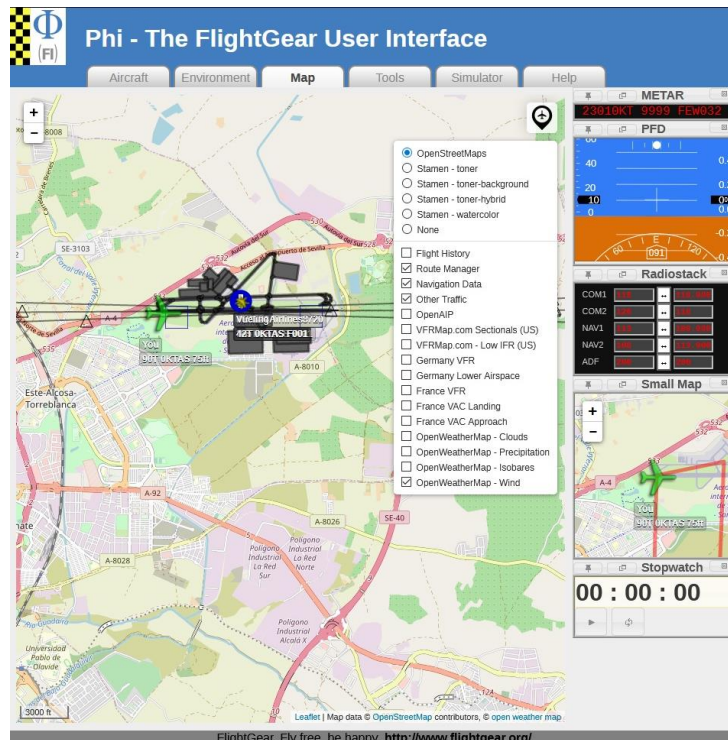


Ilustración 92 Detalle *Phi* opción *Map*

Dentro de la opción *environment*, se dispone de *Data & time*, *Weather* y *Position*, desde las cuales se puede configurar respectivamente la hora del día y la fecha, consultar el tiempo atmosférico en el área basado en el METAR, y por último una ventana para seleccionar un aeropuerto para relocalizar la aeronave (pudiéndose elegir la pista). Esto se puede observar en la Ilustración 93.

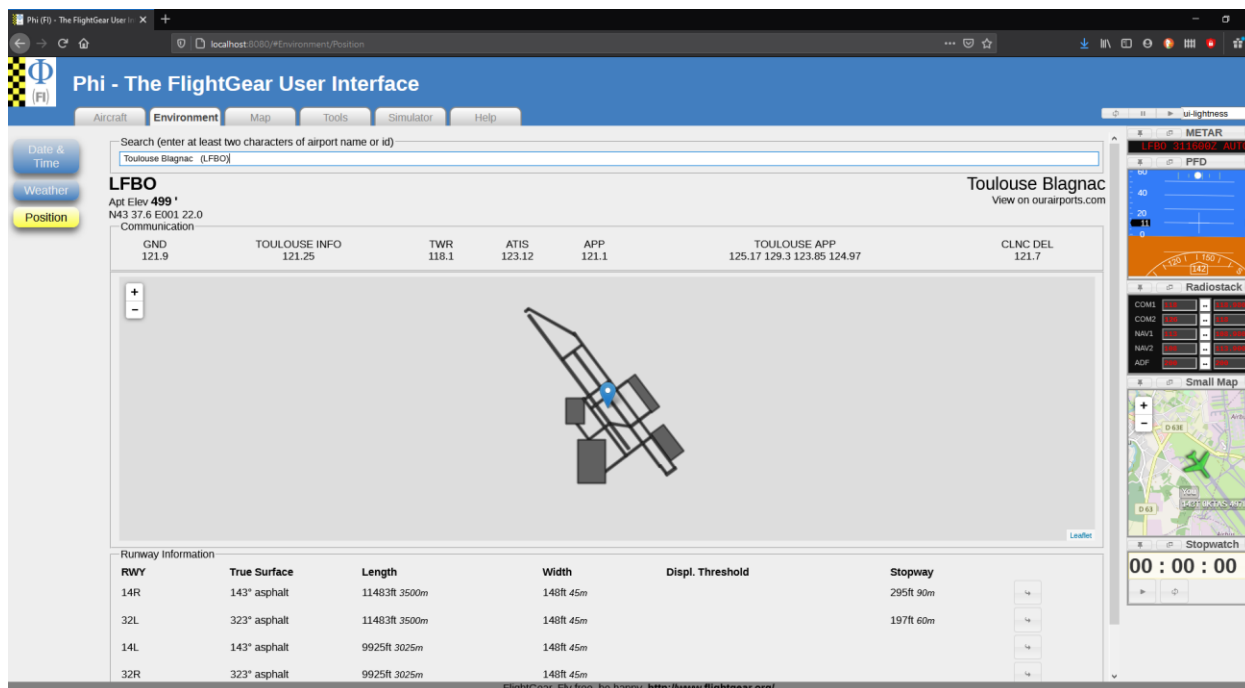


Ilustración 93 Detalle *Environment/Position*

Dentro de la opción *Aircraft*, se observan varias opciones, de las cuales no todas están disponibles (*Failures* no está todavía desarrollada), pero cabe de destacar la opción de masa y centrado (*Mass & Balance*). Desde esta opción, mediante una sencilla interfaz gráfica se puede modificar tanto la carga de combustible como la carga de pago, y ver la envolvente del centro de gravedad, viendo si en la configuración actual el avión puede ejecutar un vuelo estable. Dos distribuciones de peso pueden observar en las Ilustración 94 e Ilustración 95.

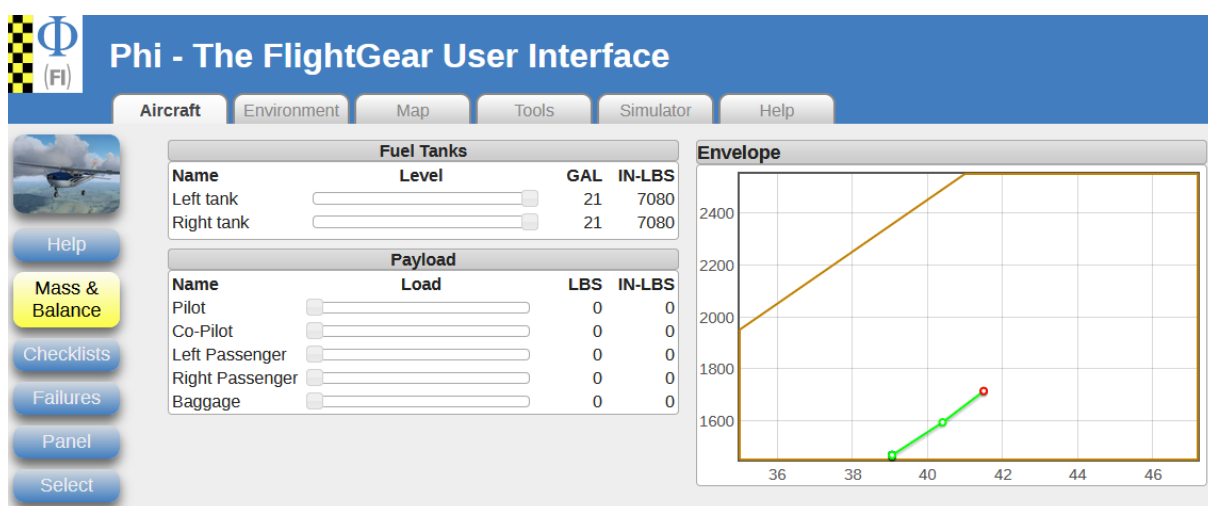


Ilustración 94 Detalle caso uno masa y centrado

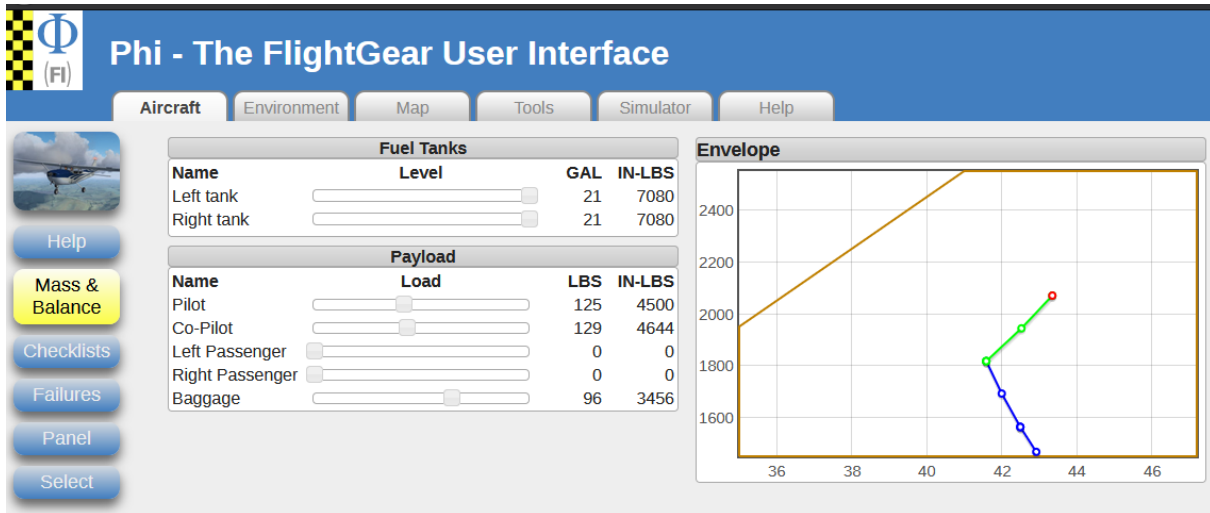


Ilustración 95 Detalle caso dos masas y centrado

Otra opción interesante, desde el punto de vista del usuario, es el de las *Checklists*, en el cual se da una guía de chequeo en varias situaciones, desde normales hasta de emergencia, ampliamente utilizadas en la aviación. Estas se pueden observar en la Ilustración 96.

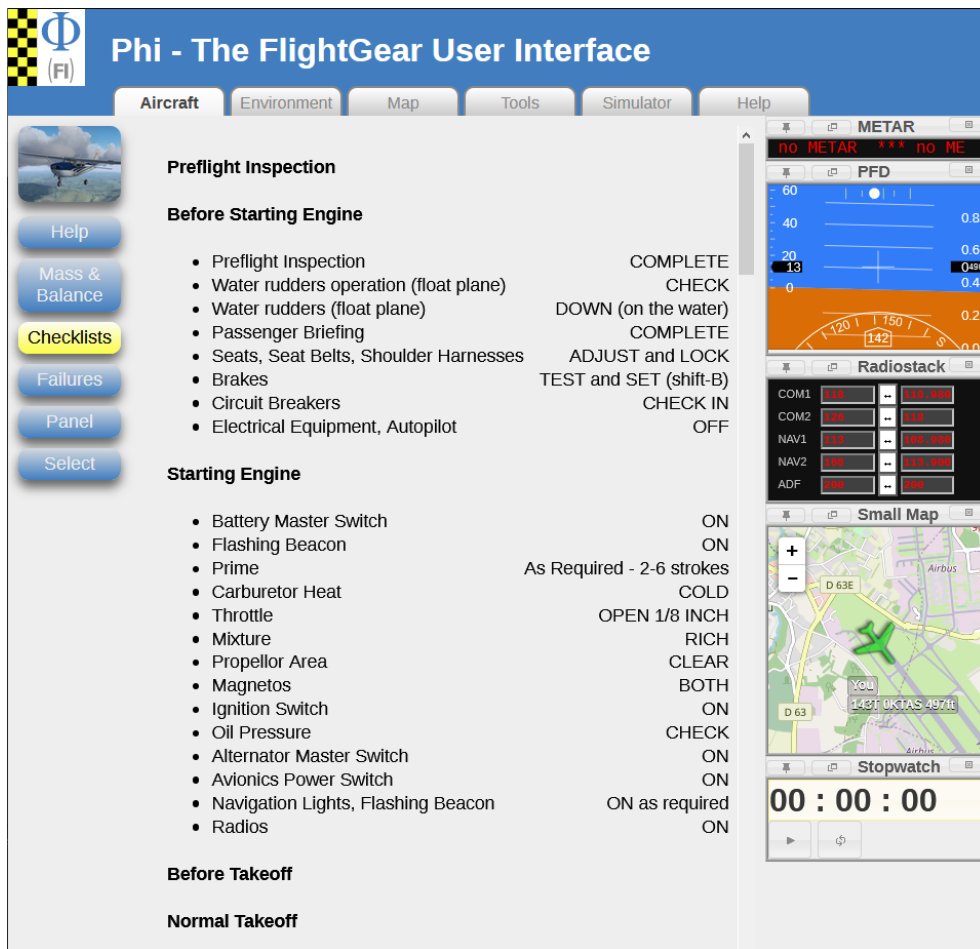


Ilustración 96 Ejemplo checklist caso C172P

Por último, se encuentra la opción *Simulator*, desde la cual se puede acceder al árbol de propiedades, tanto para la lectura como para la escritura. Esto permite desde una estructura externa indagar en la simulación e incluso modificar datos en tiempo real, pudiéndose incluso simular fallos de motor o diferentes casos de fallo.

3.3.3.3. Flightgear TQPanel

Las dos soluciones anteriores están basadas en HTML/JavaScript/CSS, una solución muy útil pero que requiere un conocimiento previo o una formación que no se tenía durante el desarrollo de este proyecto. Pero investigando poco a poco en las herramientas auxiliares generadas por la comunidad se llegó a *FlightGear TQPanel*.

Esta herramienta está desarrollada en *Python/Kivy*, librería la cual será introducida posteriormente. A modo de introducción, *Kivy* una librería enfocada al desarrollo de aplicaciones, pudiendo ser ejecutada en diferentes sistemas operativos y que permite la entrada simultánea de entradas en pantallas táctiles (*multitouch*).

Esto hace posible crear aplicaciones táctiles de todo tipo, desde minijuegos hasta tablas de datos, pasando por el susodicho *TQPanel*, el cual es un panel digital que permite interactuar con *Flightgear* haciéndole llegar entradas a través de puertos *sockets*. A través de una serie de botonería y palancas con varias posiciones, hace posible modificar parámetros tan importantes como la posición de la palanca de gases, la posición del *trim*, la posición de flaps o el freno de parking.



Ilustración 97 Detalle interfaz *TQPanel*

En la Ilustración 97 se observa un detalle de la interfaz *TQPanel*, en la cual se puede distinguir el diseño de la aplicación, distinguiéndose dos clases diferentes de interacción. La primera parte anteriormente comentada de las palancas, cuyo valor variara en función de la posición de la misma permitiendo introducir un rango de valores contenidos entre dos máximos. La segunda estarían la botonería, la cual permite introducir booleanos al sistema, encendido o apagado, arriba o abajo, para poder controlar variables tales como las luces o el tren de aterrizaje.

Se observa que este panel fue diseñado para una aeronave en particular, una aeronave bimotor, pero podría ser adaptada a otro tipo de aeronaves, como cuadrimotores o monomotores, modificando el código *Python*

original. De esta forma se está ante una herramienta muy potente a la hora de simular panalería necesaria para recrear la experiencia de vuelo, sentando un precedente para crear diferentes herramientas sin necesidad de un hardware específico para ella.

Debido al interés que suscitó esta herramienta, debido a su gran potencial para el desarrollo interfaces gráficas para la implementación de una estación de simulación, se hizo un estudio de la herramienta en cuestión, que será introducido a continuación.

3.3.3.3.1 Introducción a Kivy

Kivy es una herramienta permite utilizar una interfaz gráfica para ejecutar un código *Python*. En dicha interfaz gráfica se muestran por pantalla varios tipos de elementos con los que se pueden interactuar, desde palancas, botonería o ventanas de texto, los cuales son los disparadores de unas funciones ejecutadas en *Python*, pudiendo estas utilizar datos extraídos de ellos. Por ejemplo, se puede programar asociar a un botón de la pantalla un cierto evento, como podría ser que cierto booleano cambie su valor.

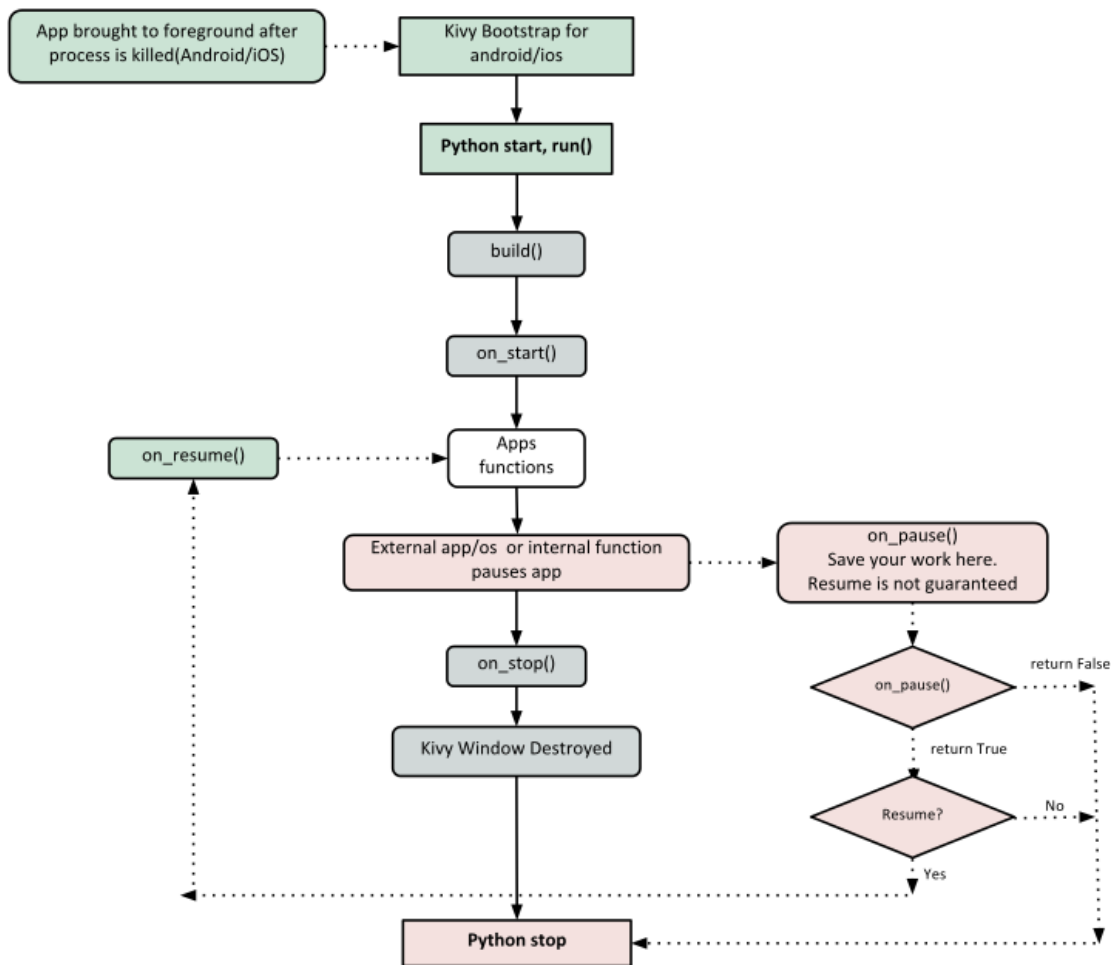


Ilustración 98 Detalle esquema funcionamiento de Kivy

En la Ilustración 98 se observa un esquema del funcionamiento de *Kivy*, en el que a grandes rasgos se puede observar un bucle infinito, el cual estará constantemente chequeando si hay algún evento en el recuadro *Apps functions*, hasta que eventualmente se mande una señal para cerrar la aplicación. La única forma de interactuar con dicho bucle infinito será mediante las entradas que se hayan configurado previamente.

Hay dos cosas a configurar, las funcionalidades y el apartado gráfico. Por una parte, se configurará las múltiples funciones que se ejecutaran a raíz que se interactúe con la pantalla, y después la propia apariencia de la aplicación, así como diferentes elementos de interacción. Aunque el aspecto gráfico se puede programar internamente desde el código *Python*, a medida que este se haga más complejo la claridad y la capacidad de

depurar código se ven afectadas. Para ello, *Kivy* ofrece su propio lenguaje de programación *Kv*, el cual está enfocado a separar las funciones del diseño de la interfaz.

Para visualizar más claramente estas dos partes, se va a introducir un ejemplo para poder explicar cada una de las partes más fácilmente, entrando en alguna de ellas en detalle, antes de entrar con la función principal.

3.3.3.2 Ejemplo Kivy

Primeramente, se va a abordar una función tipo en *Python*, posteriormente el código *Kv* y por último la aplicación final.

```
1 # -*- coding: utf-8 -*-
2 """
3 Created on Sat Feb 6 17:01:50 2021
4
5 @author: miguel
6 """
7 |
8
9 import numpy
10 import kivy
11 import math
12 from kivy.app import App
13 from kivy.uix.boxlayout import BoxLayout
14 from kivy.uix.floatlayout import FloatLayout
15 from kivy.uix.gridlayout import GridLayout
16 from kivy.clock import Clock
17
18 from kivy.config import Config
19 Config.set("graphics", "width", 1000)
20 Config.set("graphics", "height", 400)
21
22 import time
23 import sys
24 import errno
25 from socket import *
26
27
28
29 class example(FloatLayout):
30     def basic_function(self,*args):
31         print("I've been here")
32
33     def right_side(self,*args):
34         print("Right side")
35
36
37
38 class MainApp(App):
39     def build(self):
40         return example()
41
42
43 if __name__ == '__main__':
44     MainApp().run()
```

Ilustración 99 Detalle código *Python* ejemplo

En la Ilustración 99 se puede observar un detalle del código *Python* usado para este ejemplo. En él se pueden observar 4 partes bien diferenciadas mediante unos marcos rojos.

Dentro del primer marco se observa cómo se han importado varias librerías y funciones. Algunas de ellas son funciones matemáticas típicas, importadas desde las librerías *numpy* y *math*, y posteriormente se observa una serie de funciones extraídas de la librería *Kivy*. Se puede observar la función *App* (función necesaria para ejecutar *build()* ejemplificado en la ilustración 99), o las funciones que se podrían denominar *Layout*, las cuales configuran la manera en la que están organizados los *widgets* (que son los elementos con los cuales se pueden interactuar como los botones). Posteriormente se entrará más en detalle sobre estos dos últimos

conceptos desde la descripción del lenguaje *Kv*.

A parte de ello se observa que en el primer marco se han ejecutado dos funciones para predefinir las dimensiones exteriores de la ventana de la aplicación, todo ello mediante la acción *Config.set()*.

Dentro del segundo marco se define la clase *example*. Una clase es una plantilla genérica de un objeto en particular, que se puede utilizar tantas veces como se quiera para generar distintas instancias. En la Ilustración 100 se observa un ejemplo de una clase *identification()*, la cual es una clase para crear un objeto que recoja los datos personales de un individuo. Esta se puede utilizar para tantos usuarios como se quieran, simplemente introduciendo los datos particulares. Dos instancias genéricas serían *usuario1=identification("Manuel Pellegrino", "12345678A")* y *usuario2=identification("Sergio Canaletas", "87654321B")*.

Se observa que aparte de dichos atributos, se pueden definir una serie de funcionalidades que inician una acción o proceso, cuyas variables de entrada en este caso ejemplificador son *self* y **args*, aunque podrían ser entradas concretas.

Cuando se utiliza *Python* como lenguaje de programación orientado a objetos, se utiliza *self* como la etiqueta reservada para el primer parámetro o como raíz desde el cual se añadirán el resto de atributos de la clase, de una forma parecida estructura de árbol como la que se ha observado en *Flightgear*. Se observa en la Ilustración 100 un ejemplo concreto, donde la clase *identification* tiene como atributos el nombre completo (*self.whole_name*) y el código de identificación (*self.identification_code*).

En este ejemplo también se observa **args*, que es una herramienta para codificar que la función puede ser alimentada por un número de entradas indeterminado. Por ejemplo, si se configurara el objeto *usuario2=identification("SergioCanaletas", "87654321B")* y se ejecutara *usuario2.print_data("01/01/1991")*, la salida por la consola sería primero Sergio Canaletas, después el 87654321B y por último la 01/01/1991.

```
class identification():
    def __init__(self, whole_name, identification_code):
        self.whole_name= whole_name
        self.identification_code= identification_code

    def info(self):
        print('Mi nombre es' + str(self.whole_name) + 'y mi codigo de identificacion es' + str(self.identification_code))

    def print_data(self,*args):
        print(str(self.whole_name))
        print(str(self.identification_code))
        for arg in args:
            print(str(arg))
```

Ilustración 100 Ejemplo self y *args

En el tercer marco de la Ilustración 99 se muestra la clase *MainApp*, cuya entrada es *APP*. Esta clase es la base para crear aplicaciones *Kivy*, la cual será el medio principal para interactuar con el bucle cerrado. Es muy común que usando estas librerías se haga una subdivisión de clases, generando las funcionalidades en una clase auxiliar (en este caso sería la clase recogida en el marco dos) y una clase lanzadora *MainApp*.

En el cuarto marco se muestra cómo se puede lanzar la aplicación, mediante una iniciación del código y donde se ejecuta la función *run()* sobre la clase *MainApp* genérica para arrancar la simulación.

Una vez explicado el apartado de las funcionalidades en *Python*, se pasará a diseñar el apartado gráfico (el orden puede permutarse sin problema).

```

1 <example>:
2   canvas:
3       Color:
4           rgb: (0, 0, 0)
5       Rectangle:
6           pos: self.pos
7           size: self.size
8
9
10      FloatLayout:
11          Button:
12              size_hint: 0.1, 0.1
13              color_background: 0,0,0,1
14              pos_hint: {"center_x":0.5,"center_y":0.5}
15              text: "where?"
16              on_press: root.basic_function()
17
18          Button:
19              size_hint: 0.1, 0.1
20              color_background: 0,0,0,1
21              pos_hint: {"center_x":0.9,"center_y":0.5}
22              text: "right"
23              on_press: root.right_side()
24
25          Label:
26              text:"EXAMPLE"
27              color: 1,0,0,1
28              font_size: 100
29              size: 0.5, 0.1
30              pos_hint: {"center_x":0.5,"center_y":0.8}

```

Ilustración 101 Detalle lenguaje Kv

En la Ilustración 101 se observa un código en K_v, diseñado para la aplicación mostrada en la Ilustración 99. Es importante remarcar que para que funcione ambos archivos (archivo.py y archivo.kv) deben estar albergados a la misma altura, es decir, en la misma carpeta. Además de ello, se ve que para que K_{ivy} sepa que interfaz gráfica corresponde que clase, el nombre de las mismas coinciden. En el caso que se ha usado de ejemplo, la clase `example` le corresponde el panel `<example>`.

En este ejemplo se puede observar una primera función definida, `canvas`, la cual marca el fondo gráfico de la aplicación. Esta alberga una serie de atributos que se pueden modificar a través de una serie de comandos, como se observa con la aplicación del color. No se entrará en detalle de todas las funcionalidades albergadas, para ello se puede consultar la documentación oficial de la librería.

Posteriormente se encuentra la definición de como se organizará la información por pantalla, en este caso se ha definido como *FloatLayout*. Las conocidas *Layouts* marcan como se organizará la información por pantalla, siendo algunas de las opciones las siguientes:

- *AnchorLayout*: elementos de interacción posicionados arriba, abajo, izquierda y derecha
- *BoxLayout*: elementos de interacción posicionados secuencialmente
- *FloatLayout*: elementos de interacción posicionados sin restricciones
- *RelativeLayout*: elementos de interacción posicionados en relación con el *Layout*
- *GridLayout*: elementos de interacción posicionados en columnas y filas

Se pueden generar estructuras dentro de otras estructuras, pudiéndose así combinarse al gusto del diseñador. Dentro de ellas se definirán los distintos elementos de interacción, los cuales son llamados *widgets*. Esta librería nos ofrece una serie fija de formas de interaccionar con la pantalla, como son *button*, *slider*, *label*, *switch* etc. Cada una de ellas nos ofrece una forma de transmitir o interactuar información con el usuario, como por ejemplo *button* o botón en castellano puede ser asociados con la ejecución de un tipo de función *Python*, como se observa en el ejemplo de la Ilustración 101.

En ella se observa como los botones *where* y *right* tienen asociadas unas funciones definidas en la clase *example*, las cuales escribirán un texto particular por la consola. Además, se ha definido una label *EXAMPLE*

que es una forma de mostrar texto por pantalla, donde se pueden configurar una serie de atributos como el tamaño de la fuente o el color.



Ilustración 102 Detalle aplicación *example*

En la Ilustración 102 se muestra la pantalla generada una vez que se ha ejecutado el código *Python*. En ella se puede observar los *widgets* configurados en el ejemplo de la Ilustración 99, tanto los botones como la *label* o etiqueta. Cuando se hace clic sobre los diferentes botones, se ejecutan las correspondientes funciones definidas en la clase *example*. El texto mostrado por la consola de *Python* se observa en la Ilustración 103, donde se muestra la frase “*Right side*” y “*I’ve been here*” tras haber presionado dos veces cada uno de los botones “*where?*” Y “*right*”.

```
[INFO ] [Base      ] Start application main loop
[INFO ] [GL          ] NPOT texture support is available
I've been here
I've been here
Right side
Right side
```

Ilustración 103 Detalle salida de texto por consola

3.3.4. Ventajas e inconvenientes

Una vez estudiado a fondo *Flightgear* junto a todas las posibilidades que se ofrecen, se va a proseguir a un análisis ventajas e inconvenientes para tomar una decisión definitiva.

3.3.4.1. Ventajas

Una herramienta robusta y completa

No cabe duda que *Flightgear* es un simulador muy completo, incluso siendo un simulador código libre, el cual ya integra herramientas anteriormente testadas en este proyecto, como *JSBSim*, además de modelos gráficos, base de datos visuales, e incluso un modo multijugador que permite la coexistencia de varias entidades a la vez.

Esto sumado a su diseño inteligente, hace que su manipulación y su uso para productos de terceros sea no solo viable, sino que se promueva para ofrecer más funcionalidades a su comunidad y que esta crezca. También comentar que, aunque sea una herramienta en constante desarrollo, hay versiones estables que no necesitan de actualizaciones y que se pueden tomar como referencia.

No es sorprendente que diferentes empresas o universidades ya hayan confiado en esta herramienta para desarrollar múltiples proyectos.

Gran comunidad y soporte asegurado

Flightgear tiene una comunidad muy amplia bastante activa, tanto en el uso de la herramienta como en los foros de la misma, por lo que las dudas sobre el programa suelen haber sido ya preguntadas o se pueden postear en el foro oficial (<https://forum.flightgear.org/>). Esto hace que la sensación de estancamiento o frustración sea menor, porque posiblemente puedas ver las dudas o problemas reflejados ya y encuentres soluciones. No solo eso, también hace que se pueda estar al día de las nuevas funcionalidades

Básicamente se tiene un gran reservorio de información del cual nutrirse a la hora de abordar diferentes situaciones, en vez de empezar desde el desconocimiento, o desde manuales quizás no tan completos.

Cumple todos los requisitos

Si se vuelve a los requerimientos que se propusieron en este proyecto (1) salta a la vista que *Flightgear* cumple con todos ellos. Por un lado, es un simulador en código libre, que permitiría su difusión entre el alumnado con facilidad. Aunque este no sea un simulador modular, se puede construir a partir del gracias a su diseño inteligente a través de su árbol de propiedades. Y, para terminar, es un simulador versátil ya que los archivos de configuración siguen una estructura fija, conocida y replicable, haciendo que se puedan crear grandes librerías de aeronaves ya conocidas, o se puedan simular prototipos basados en estimaciones antes de su fabricación.

3.3.4.1. Desventaja

Consumo computacional mayor debido al procesamiento gráfico

La gran desventaja es que obviamente la capacidad de cómputo necesaria para ejecutar correctamente este programa no es equivalente a la de ejecutar *JSBSim* por sí solo. Esto hace que se requiera un hardware mínimo para su correcto uso, lo cual es un factor limitante, pero relativamente asumible.

Pero sí que limita su uso solo para ser una interfaz visual o para simular la experiencia de vuelo, dejando la simulación de la dinámica para librerías más eficientes.

Complejidad a la hora de aportar

En simulador ya muy robusto y testado, que ha pasado por infinidad de proyectos de voluntarios bien formados en sus ámbitos profesionales, por lo que innovar en la herramienta es relativamente difícil.

Los modelos dinámicos están muy bien empacados y son muy eficientes, los modelos gráficos requieren una formación extra, así como el desarrollo de páginas web que funcionan como interfaz de la propia herramienta.

En resumen, para poder aportar al desarrollo de esta herramienta es necesario un conocimiento previo de varios lenguajes de programación, así como el buen entendimiento de la misma.

3.3.5. Conclusiones

Flightgear es una herramienta muy completa, la cual tiene todas las cualidades requeridas para este proyecto.

Incluye la librería *JSBSim* para la simulación dinámica, una librería muy compacta y eficiente, ofrece una base de datos visual (modelos de montañas, ríos y edificios) y de navegación a nivel mundial, modelos tridimensionales de las aeronaves, sus cabinas, paneles e instrumentos, etc. La implementación de todos estos elementos uno a uno podría conllevar un trabajo quizás similar al de un proyecto individual.

Esto hace que las dos partes teóricas de un simulador estuvieran ampliamente cubiertas, tanto la estación, que ha sido la parte limitante para las otras plataformas, y la simulación, que tiene por defecto unas librerías muy potentes y eficientes.

3.4. **Decision entre las soluciones disponibles**

Una vez estudiadas las diferentes soluciones se entró en un dilema. Existían dos caminos para proseguir el proyecto.

El primero de ellos es utilizar para la simulación *Matlab/Simulink* o *JSBSim* y construir una estación externa (1.3) para su control, empezando así una serie de varios proyectos que implementarían base de datos visuales, de navegación, aplicaciones de control externo, etc.

La otra opción disponible es la de construir una cabina de entrenamiento basado en *Flightgear* para poder demostrar la potencia de la herramienta y abrir la posibilidad para su uso lectivo dentro de la ETSI.

El principal problema que tiene la primera opción es el tiempo de implementación, ya que es un trabajo que se antoja faraónico para una única persona. Como bien se vio que dicha empresa es posible, ya que se pudo implementar una simulación dinámica controlada externamente mediante un *gamepad*, pero generar un producto mínimo viable requeriría grandes esfuerzos para crear la llamada estación, teniéndose que implementar herramientas de control (desde interfaces para la precarga de datos, o los propios mandos) e implicaría el desarrollo de una interfaz gráfica, (lo cual necesitaría primeramente un motor gráfico y después mucho trabajo para generar las bases de datos visuales y de navegación).

Por ello, se decide optar por la segunda opción, y utilizar *Flightgear* para poder conseguir el objetivo del proyecto, el cual es el de poder realizar un vuelo tanto instrumental como manual. Para ello se van a utilizar todas las herramientas ya expuestas para *Flightgear*, y además se va a desarrollar un panel para manejar tanto el autopiloto como otras funcionalidades necesarias para un vuelo instrumental.

4. DESARROLLO DEL SIMULADOR DE VUELO

4.1. Arquitectura del simulador

En esta sección se va a exponer la estructura final del simulador, basándose en las conexiones entre las distintas partes. Como se introdujo en la subsección anterior (3.4), se han utilizado todas las herramientas anteriormente introducidas, a excepción de *Matlab/Simulink*, además del desarrollo final de un nuevo panel para poder manejar el autopiloto desde una estación externa.

Esta no es una decisión arbitraria, ya que previamente se analizó todo lo necesario para hacer una cabina funcional con todas las herramientas disponibles con *Flightgear*, y el último escollo para la navegación visual era poder manejar el autopiloto y otros elementos de una forma más cómoda desde una estación.

Se tomará como avión de referencia la C172P, no solo por ser la aeronave más completa en este simulador, sino para aportar otra herramienta a la comunidad *Flightgear*.

En la Ilustración 104 se puede observar la arquitectura final del simulador. Se ha utilizado *Flightgear* como base, debido a la versatilidad de su árbol de propiedades a la hora del intercambio de información.

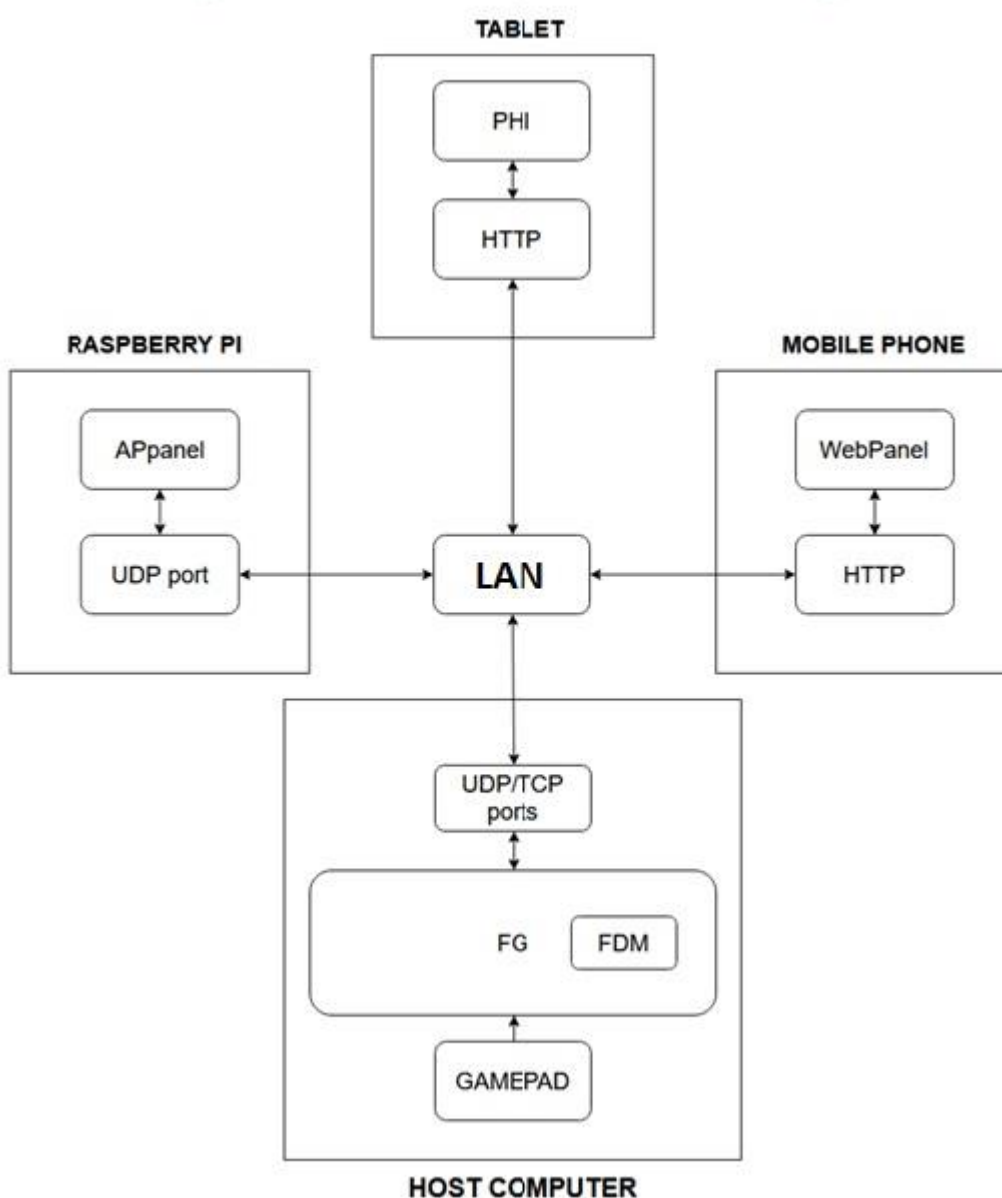


Ilustración 104 Detalle esquema de la arquitectura final

Se observa que *JSBSim* es utilizado como modelo dinámico de vuelo, estando este integrado en *Flightgear*. Posteriormente, mediante una red local (más conocida por sus siglas LAN de *Local Area Network*) y diferentes tipos de protocolos de comunicación, se conectan el resto de los elementos. Como soporte de dicha red local se utilizará la tecnología WiFi, la cual podría ser sustituida por un *Ethernet hub* si se deseará priorizar la fiabilidad y velocidad al precio.

La ventaja de esta estructura es que es escalable, siendo posible conectar a posteriori nuevas interfaces para implementar nuevas funcionalidades, como por ejemplo nuevos paneles o simuladores gráficos de palancas. No solo eso, sino que en el caso que se quisiera utilizar *Flightgear* solo como interfaz gráfica, se podría utilizar la misma estructura de red y redireccionar las interfaces activas (aquellas con capacidad de modificar datos) hacia el nuevo simulador.

Los elementos externos a *Flightgear* mostrados en la Ilustración 104 conforman la estación, cuyas funcionalidades van desde permitir usuario pilotar la aeronave (APpanel) y leer la información en tiempo real (WebPanel), y el segundo el puesto de instructor, mediante el cual se sería capaz de manipular la sesión de entrenamiento (*Phi*).



Ilustración 105 Detalle WebPanel lanzado a través de un teléfono inteligente

En la Ilustración 105 se puede observar el detalle de la aplicación de navegador WebPanel, la cual ha sido lanzada desde un *smartphone*. Lo más interesante de este método es la flexibilidad a la hora de reutilizar hardware, ya que cualquier artilugio que sea capaz de conectarse a una red WiFi y que sea capaz de lanzar un navegador muy posiblemente sea capaz de ejecutar estas herramientas basadas en HTML/JavaScript/CSS.

Cabe destacar que lo único que debe saber el usuario es la dirección IP del ordenador anfitrión, y con ella ejecutar desde el navegador del aparato correspondiente la aplicación de navegador objetivo, como se observa en las Ilustración 105 e Ilustración 106.

En la Ilustración 106 se puede observar el detalle del puesto de instructor *Phi*, el cual ya ha sido introducido previamente en el apartado [3.3.3.2.](#) Al igual que la aplicación WebPanel se puede lanzar desde cualquier aparato mediante la conexión de una red WiFi a través del navegador.

El gamepad se conecta al ordenador anfitrión, y es detectado automáticamente por *Flightgear*, pudiéndose modificar a través de su interfaz la asignación de funciones. Si se deseara cambiar el modelo dinámico de vuelo, en el caso de ejecutar *JSBSim* mediante *Python*, se deberá utilizar librerías tipo *pygame* para la utilización del mismo, como ya se adelantó en el apartado [3.2.4.2.1.](#)

Para cumplir el objetivo de hacer un vuelo instrumental satisfactorio, habiéndose estudiado el sistema desde un punto de vista de la funcionalidad, se llegó a la conclusión que era necesario no solo replicar el autopiloto, sino que con un trabajo extra se podrían añadir otras características útiles tales como la selección de frecuencias o cambiar el rumbo objetivo o el OBS, pudiendo así crear un panel totalmente funcional.

Entre las múltiples plataformas que se pueden utilizar para desarrollar estas aplicaciones, se va a optar por la

misma que se utilizó para el *Flightgear TQPanel*, siendo implementada en *Python/Kivy*. El motivo de esta decisión es que en el momento del desarrollo de este proyecto el conocimiento de este lenguaje es alto, y aunque soluciones en HTML/JavaScript/CSS se antojan más potentes desde el punto de vista de la multifuncionalidad, *Kivy* ofrece además la posibilidad de ser manejadas mediante pantallas táctiles *multitouch*.

Este se conectará al *Flightgear* para poder leer y modificar su árbol de propiedades mediante dos *sockets* UDP, uno dedicado a la entrada de datos y otro a la salida de los mismos, como se observa en la Ilustración 107.

Aunque primeramente se desarrolló el prototipo solamente utilizando la entrada de datos, posteriormente se vio que era necesaria la lectura de datos para poder actualizar algunas de las funcionalidades de navegación, como serían la lectura a tiempo real del DME y la simulación de la entrada del sensor de altitud-presión.

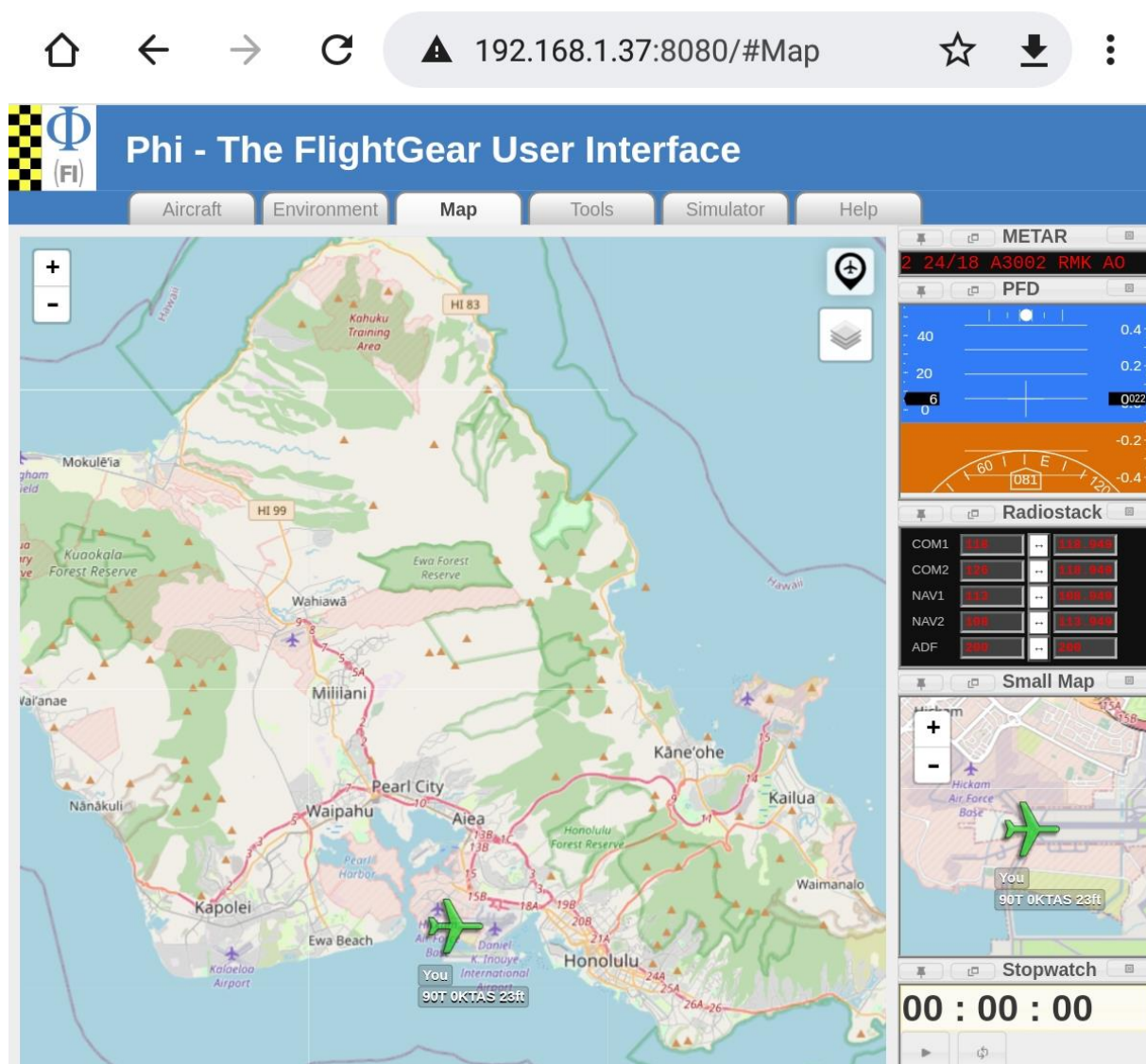


Ilustración 106 Detalle aplicación Phi lanzada desde Tablet

Para ello se necesitan las direcciones IP de cada uno de los aparatos, al ser una comunicación bidireccional. En la Ilustración 107 se observan ambos puertos UDP lanzados desde Flightgear, siendo cada uno de ellos direccionado a una IP diferente, siendo la IP del input la dirección del ordenador anfitrión, y la IP del output la dirección del aparato receptor.

Asociadas a estas direcciones IP se deben configurar unos puertos, como se ilustra en la Ilustración 108. Cabe destacar que se debe chequear a priori los puertos libres, ya que de otra forma la comunicación no podrá efectuarse si se usan puertos ocupados por programas terceros.

```
Windows PowerShell
PS D:\Program Files\FlightGear 2018.3.6\bin> ./fgfs --generic=socket,in,10,192.168.1.37,6065,udp,from_kivy_autopilot --generic=socket,out,100,192.168.1.43,5050,udp,panel_dme_communication --httpd=8080
```

Ilustración 107 Detalle lanzamiento socket entrada y salida Flightgear

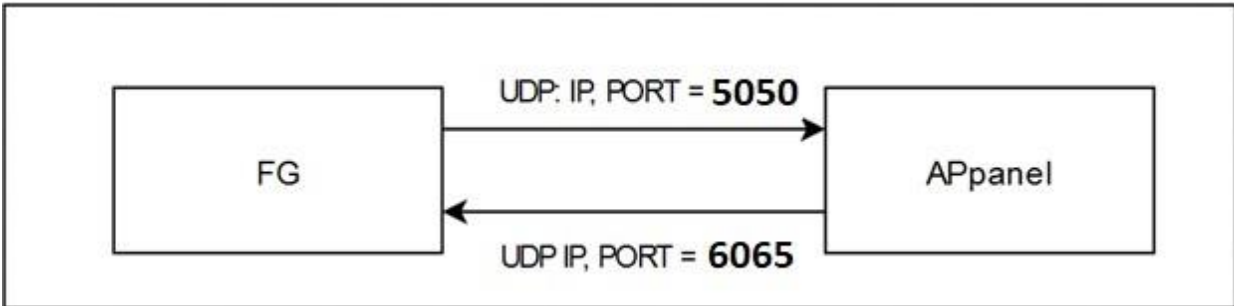


Ilustración 108 Detalle comunicación Flightgear y APpanel

En la Ilustración 109 e Ilustración 110 se observa una captura de pantalla de APpanel ejecutándose durante una simulación de vuelo. En ella se puede observar tanto la interfaz gráfica, como los datos enviados desde *Flightgear* alojados en la consola *Python*.

```
107, 1, 1.9, 24.703915
107, 1, 1.9, 24.700163
108, 1, 1.9, 24.696409
108, 1, 2.0, 24.692652
108, 1, 2.0, 24.688896
108, 1, 2.0, 24.685137
109, 1, 2.0, 24.681459
109, 1, 2.0, 24.677612
109, 1, 2.0, 24.673847
109, 1, 2.1, 24.670080
110, 1, 2.1, 24.666311
110, 1, 2.1, 24.662457
110, 1, 2.1, 24.658768
110, 1, 2.2, 24.654993
110, 1, 2.2, 24.651218
111, 1, 2.2, 24.647358
111, 1, 2.2, 24.643579
```

Ilustración 109 Detalle datos enviados

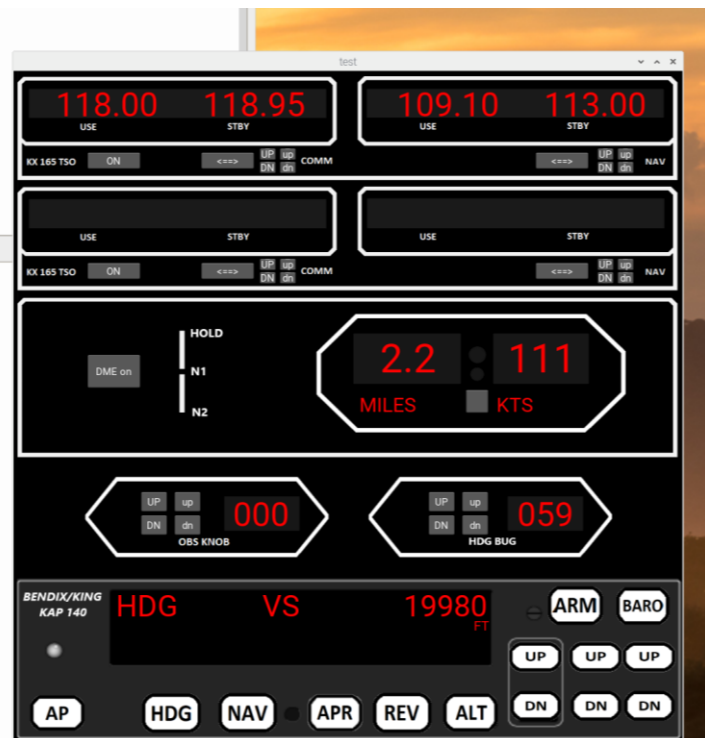


Ilustración 110 Detalle APpanel

4.2. Desarrollo APpanel

A la hora de plantear el APpanel, se organizó un diagrama de requerimientos, en el cual se desglosaban las múltiples funciones que se debían cumplir, además de la tecnología utilizada para su desarrollo. Como se observa en la Ilustración 111, el APpanel este compuesto por un apartado gráfico, el cual estaría a su vez desglosado en la organización de la estructura y el desarrollo de las imágenes, y del apartado de funcionalidades, el cual estaría a su vez subdividido en la funcionalidad de cada uno de los botones y las comunicaciones con el programa principal.

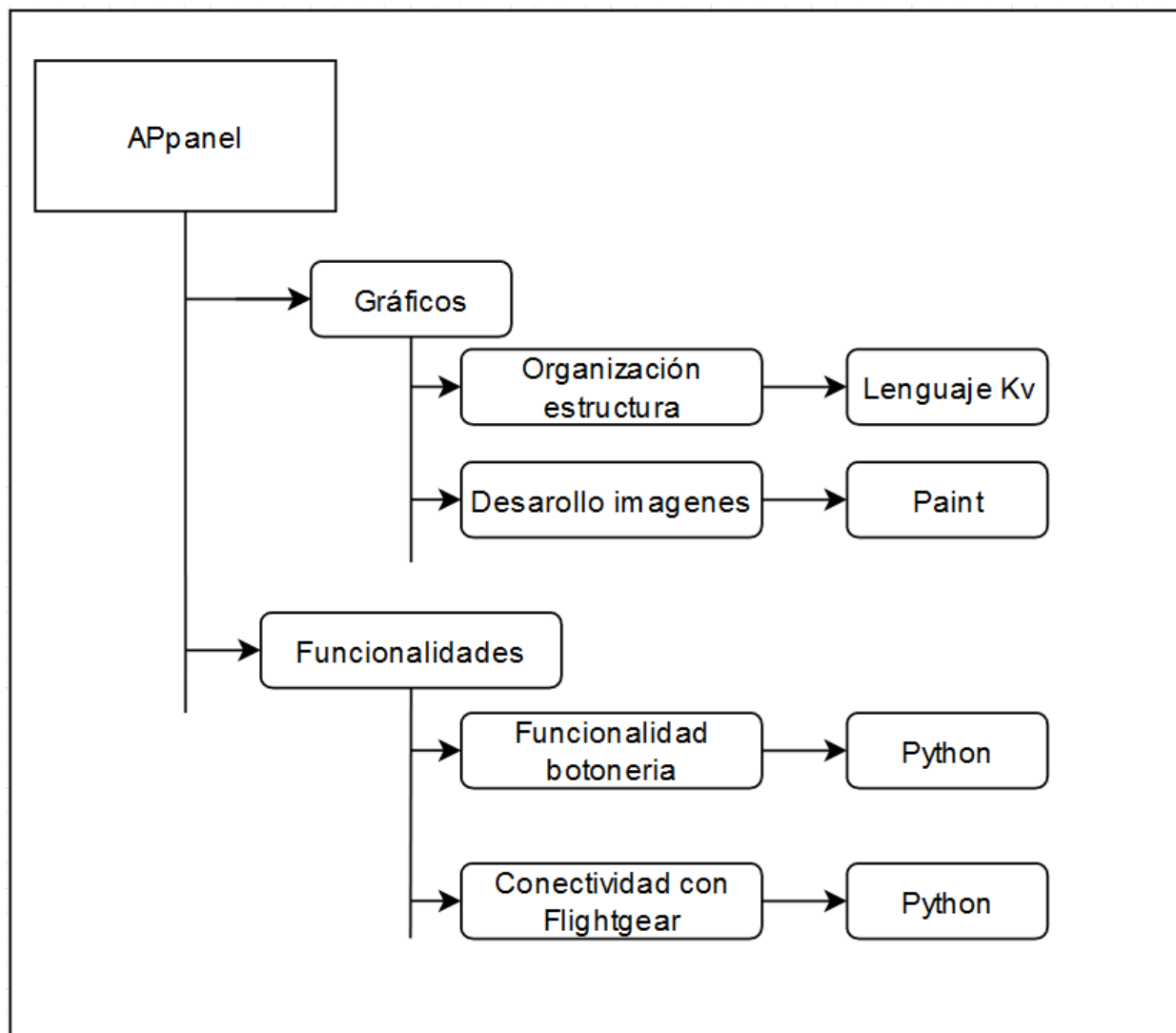


Ilustración 111 Detalle esquema de desarrollo APpanel

Todo esto se planificó una vez entendido el funcionamiento básico de *Kivy* y de todo su potencial, explicado en detalle en la sección [3.3.3.3.1](#).

Para llevar a cabo esta empresa, se debía primero comprender como estos sistemas estaban integrados en *Flightgear*, su apartado gráfico y como funcionaban, utilizando como aeronave de referencia la C172P simulado en *Flightgear*.

A la hora de la réplica de las funcionalidades, se utilizó una simulación donde se empezó a hacer una batería de pruebas al autopiloto, fijándose dentro del árbol de propiedades que elementos cambiaban sus valores y que comportamientos había detrás de ellos.



Ilustración 112 Detalle cabina y árbol de propiedades

En la Ilustración 112 se observa el árbol de propiedades apuntando a la parte del mismo donde se almacena la información relativa al autopiloto, junto a la cabina. En dicho cuadro se puede observar diferentes parámetros, los cuales son booleanos o números enteros, cuyos valores configuran el modo de funcionamiento a llevar a cabo.

De una manera simple, cuando dentro de la cabina virtual de *Flightgear* se aprieta el botón de AP (acción de encendido del autopiloto), esto se simula mediante una serie de parámetros que cambian su valor con el fin de que el sistema inicie el modo de funcionamiento por defecto, que en este caso sería activar el modo *rol* y *pitch* (están separados el plano horizontal del vertical).

Antes de progresar se explicarán los modos de funcionamiento que nos ofrece dicho autopiloto:

- Modo ROL: modo que mantiene un vuelo simétrico.
- Modo HDG: el avión captura el rumbo deseado introducido con el Reading bug. Si este se modifica mientras se captura dicho rumbo se seguirá dinámicamente hasta capturar el siguiente.
- Modo NAV (Navigation Mode Selector): Cuando se selecciona dicho modo, se procederá a capturar y seguir el VOR, LOC o GPS seleccionado en el HSI o CDI. Dicho modo se recomienda para la navegación en ruta.
- Modo APR (Approach Mode Selector): Cuando se selecciona dicho modo, se procederá a capturar y seguir el VOR, LOC o GPS seleccionado en el HSI o CDI. Dicho modo se recomienda para las aproximaciones instrumentales.
- Modo REV (Back Course Approach Mode Selector): Cuando se selecciona dicho modo, el funcionamiento será similar al del modo APR, pero de manera inversa. En otras palabras, este modo es para aproximaciones desde el lado contrario a la senda de planeo.

4.2.1 Apartado gráfico

A la hora del diseño del apartado gráfico, se identificaron los instrumentos necesarios para el vuelo instrumental dentro de la interfaz *Kivy*. Debido a limitaciones de la herramienta, donde las ruletas de selección no están recogidas como *widgets*, pasaron a definirse como *buttons* ofreciendo las mismas funcionalidades. Todo esto se puede observar en la Ilustración 113, donde queda recogida la interfaz final del panel simulado. Para más información, hay un desarrollo en detalle de la herramienta *Kivy* en el apartado [3.3.3.3.1](#).

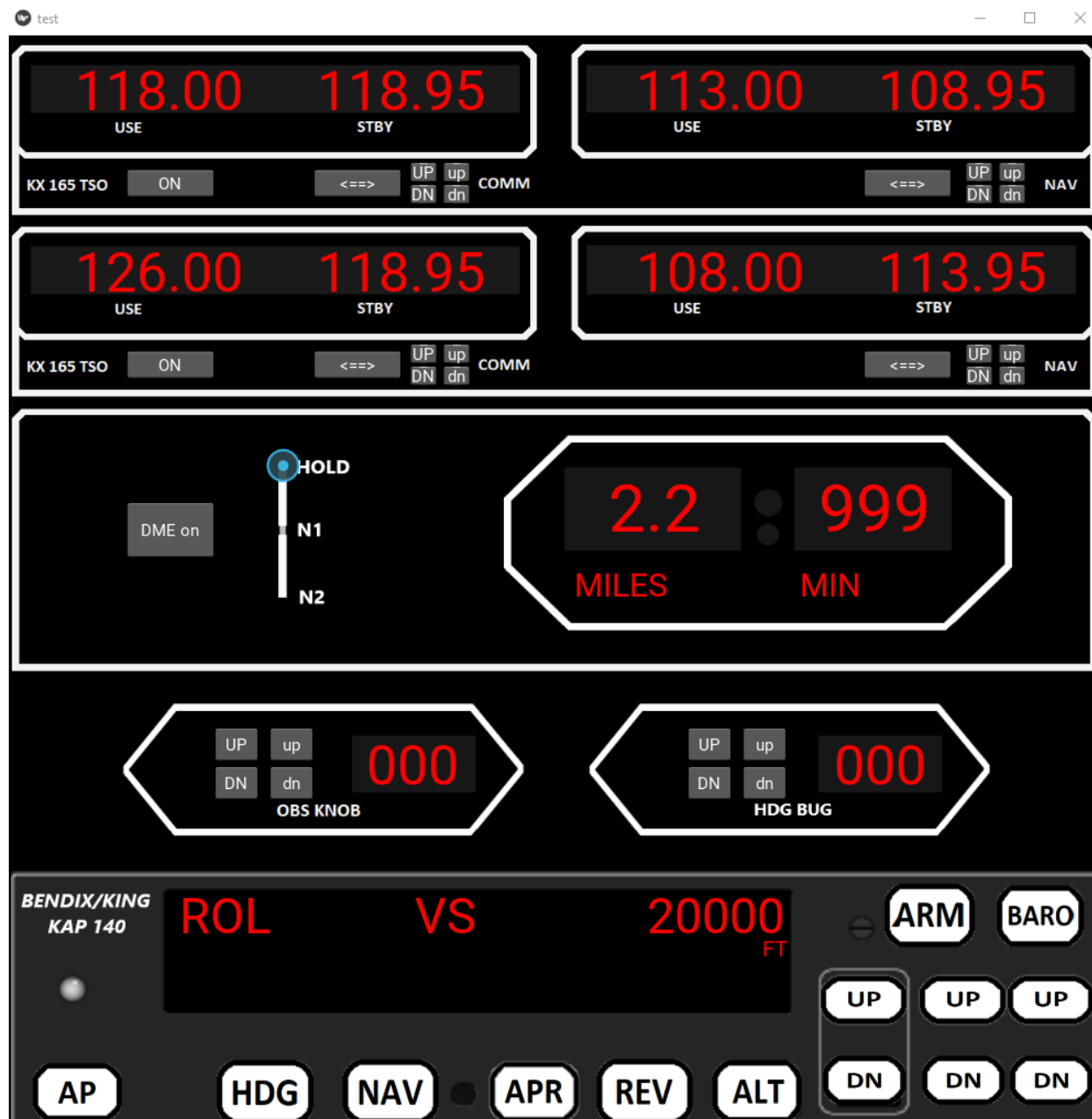


Ilustración 113 Detalle del panel simulado

En la ilustración anterior se pueden observar todos los sistemas replicados, los cuales son:

- Autopiloto: Principal objetivo de la herramienta, este depende de otros sistemas para poder ser totalmente funcional. A parte de poder seleccionar los distintos modos de navegación, es posible modificar el reglaje barométrico.
- OBS knob: Para poder seleccionar el radial deseado a capturar, replicar el OBS knob es totalmente necesario. Gracias a este selector simulado, se podrá seleccionar el radial a capturar deseado. Recordar que gracias a *c172p-WebPanel* ([3.3.3.1](#)) se dispone de una réplica del indicador. Cabe destacar que dicho OBS knob corresponde al VOR primario en este prototipo.
- HDG bug: Al igual que el caso anterior, para cambiar el rumbo de la aeronave este es totalmente necesario, en él se puede seleccionar mediante botonería el rumbo deseado a capturar con el modo NAV del autopiloto.
- DME: Permite seleccionar las frecuencias de navegación deseadas a la hora de conectarse a diferentes DME. Por pantalla se mostrará tanto las millas a las cuales se encuentra la aeronave respecto a una

baliza DME y en función de la elección del piloto se podrá mostrar los minutos necesarios para llegar a la antena si se vuela directamente hacia a ella, o la velocidad de la aeronave.

- Selector de frecuencias de comunicación y navegación: En dicho instrumento se pueden seleccionar las frecuencias en uso y *stand-by*, tanto para las comunicaciones como para los datos de navegación.

La estructura de la aplicación se definió mediante el lenguaje *Kv*, introducido anteriormente la sección [3.3.3.3.1](#), y en él se definen el aspecto general, todos los *widgets* que se van a emplear y las *labels*, encargadas de replicar las pantallas LCD en la interfaz gráfica.

En la Ilustración 114 se puede observar cómo están definidos los botones y las *labels*. Todos ellos contarán con un identificador (*id*), un texto asociado el cual será modificable mediante las funciones *Python*, y otros parámetros como la posición que ocuparan dentro *FloatLayout* o su tamaño.

```

Button:
    id: dn1_nav2
    size_hint: 0.025, 0.0175
    pos_hint: {"center_x":0.885,"center_y":0.69}
    text: "DN"
    on_press: root.dn1_nav2()
Button:
    id: dn2_nav2
    size_hint: 0.025, 0.0175
    pos_hint: {"center_x":0.915,"center_y":0.69}
    text: "dn"
    on_press: root.dn2_nav2()

#####

Label:
    id: hdg_mode_text
    font_size: 45
    size: 0.09, 0.065
    background_color: 1, 1, 1, 1
    color: 1,0,0,1
    pos_hint: {"center_x":0.2,"center_y":0.2}
    text: ""
Label:
    id: ap_activated_text
    font_size: 45
    size: 0.09, 0.065
    background_color: 1, 1, 1, 1
    color: 1,0,0,1
    pos_hint: {"center_x":0.3,"center_y":0.2}
    text: ""

```

Ilustración 114 Detalle código *Kv*

Además, cada uno de estos elementos definidos en el lenguaje *Kv* pueden tener asociados unas imágenes. En el caso de este proyecto, se utilizó la aplicación de Windows *Paint* para el desarrollo de alguna de ellas, ya que, aunque esta aplicación no es la más potente, posiblemente sea la más resolutiva.

Desde el código implementado en el lenguaje *Kv* se puede asociar las funciones dentro del código *Python* que se ejecutará cuando se pulsen los diferentes *widgets*. De esta manera, queda recogido el diseño de la interfaz gráfica, y será en *Python* donde se diseñarán todas las funcionalidades.

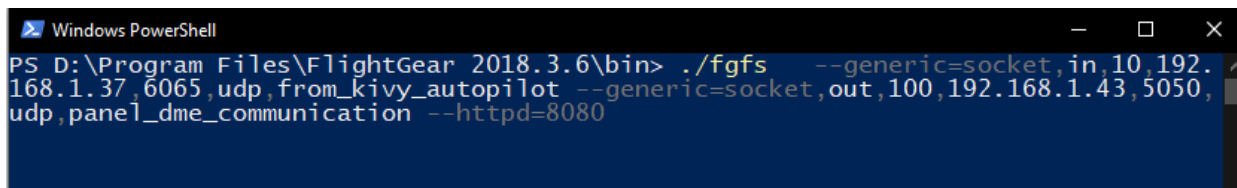
4.2.2 Apartado de funcionalidades

La estructura de dicho código *Python* seguirá una forma similar a la del ejemplo descrito anteriormente, por lo que no se indagará en sus parámetros, pero si se describirán las funciones principales o más relevantes. Para más información, ir a la sección [3.3.3.3.1](#).

Para las comunicaciones desde el lado de APpanel, ya que estas quedan embebidas dentro de las funcionalidades de la botonería, se explicarán a la par que dichas funciones.

A la hora de establecer las comunicaciones con *Flightgear*, se debe ejecutar el simulador mediante la consola *Windows*, para poder abrir así una vía de comunicación mediante un *socket* de entrada y otro de salida, para

conectarse con el panel. Cada una de estas direcciones IP estará asociadas al aparato receptor y emisor.



```
Windows PowerShell
PS D:\Program Files\FlightGear 2018.3.6\bin> ./fgfs --generic=socket,in,10,192.168.1.37,6065,udp,from_kivy_autopilot --generic=socket,out,100,192.168.1.43,5050,udp,panel_dme_communication --httpd=8080
```

Ilustración 115 Detalle definición puertos de entrada a *Flightgear*

Para ello, se debe configurar a priori que datos se le van a enviar y a que espacio de memoria van dirigidos.

Como se puede observar en la Ilustración 115, es posible lanzar el simulador con unos *sockets* genéricos. Estos se pueden configurar mediante un archivo XML, como se observa en la Ilustración 116.



```
<?xml version="1.0"?>
<PropertyList>
  <generic>
    <input>
      <line_separator>newline</line_separator>
      <var_separator>,</var_separator>
    <chunk>
      <name>alt_hold</name>
      <type>bool</type>
      <!-- <format>%f</format> -->
      <format>%f</format>
      <node>/autopilot/KAP140/locks/alt-hold</node>
    </chunk>
    <chunk>
      <name>apr_hold</name>
      <!-- <type>float</type> -->
      <type>bool</type>
      <format>%f</format>
      <!-- <node>/controls/engines/engine/throttle</node> -->
      <node>/autopilot/KAP140/locks/apr-hold</node>
    </chunk>
  </generic>
</PropertyList>
```

Ilustración 116 Detalle archivo de configuración de *sockets* en *Flightgear*

Dentro de la carpeta contenedora de *Flightgear* existe una carpeta *data*, la cual recoge datos no básicos del simulador. Esta alberga una carpeta llamada *protocol*, dentro de la cual se pueden encontrar estos archivos de configuración XML.

En la Ilustración 116 se puede observar la estructura de dicho archivo, donde primeramente se define si son datos de entrada o de salida, y el símbolo de separación entre datos (en la ilustración *input* y ,). Una vez definido esto, se van configurando uno por uno todos los parámetros que se quieren enviar a *Flightgear*, definiendo para ellos un nombre de identificación, que tipo de dato se trata (un booleano, un número entero o un float), el formato en el cual este será guardado, y la posición del árbol de propiedades que debe albergarlos.

Gracias a esta comunicación, se puede definir en mediante código los parámetros que se quieren manipular y así puentear los sistemas de *Flightgear* para que sean activados mediante herramientas externas.

4.2.2.1 Características iniciales y definición de *socket*

En la Ilustración 117 se puede observar las librerías necesarias para la ejecución del panel, tanto propias para *Kivy* como otras que añaden diferentes funcionalidades. Caben destacar que las líneas 16 y 17 que aparecen en

dicho código, en las cuales se definen el ancho y el alto de la interfaz. A parte de esto, se puede observar la definición tanto las IPs como los puertos a utilizar, además de la apertura del socket de para la salida de la información.

```

7   import numpy
8   import kivy
9   import math
10  from kivy.app import App
11  from kivy.uix.boxlayout import BoxLayout
12  from kivy.uix.floatlayout import FloatLayout
13  from kivy.clock import Clock
14
15  from kivy.config import Config
16  Config.set("graphics", "width", 1000)
17  Config.set("graphics", "height", 1000)
18
19  import time
20  import sys
21  import errno
22  from socket import *
23  import threading
24
25
26
27  # host = "169.254.237.53"
28  host='192.168.1.37'
29  host_in="0.0.0.0"
30  outline = "foo"
31  FORMAT="utf-8"
32  port = 6065
33  in_port = 5050
34  # telnet_port = 9009
35  #buf = 1024
36  addr = (host,port)
37  in_addr = (host_in,in_port)
38  UDPSock = socket(AF_INET,SOCK_DGRAM)
39  UDPSock.setsockopt(SOL_SOCKET, SO_SNDBUF, 4096)
40  # UDPSock.bind(addr)

```

Ilustración 117 Detalle código *Python*

```

38  class my_complete_panel(FloatLayout
39      # None
40
41      # =====
42      # General definitions
43      # =====
44      alt_hold = 0
45      apr_hold = 0
46      gs_hold = 0
47      hdg_hold = 0
48      nav_hold = 0
49      pitch_arm = 0
50      pitch_axis = 0
51      pitch_mode = 0
52      rev_hold = 0
53      roll_arm = 0
54      roll_axis = 0
55      roll_mode = 0
56
57      vector = [alt_hold,
58                apr_hold,
59                gs_hold,
60                hdg_hold,
61                nav_hold,
62                pitch_arm,
63                pitch_axis,
64                pitch_mode,
65                rev_hold,
66                roll_arm,
67                roll_axis,
68                roll_mode]
69

```

Ilustración 118 Detalle inicialización

Una vez importadas todas las librerías, se pasa a definir la clase principal, donde quedan recogidos tanto los parámetros iniciales necesarios para el buen funcionamiento de la aplicación, como otros parámetros auxiliares necesarios para la ejecución de las distintas funciones.

En la Ilustración 118 se observan todos los parámetros principales con los que se configuran los modos del autopiloto. Estos están basados del árbol de propiedades de *Flightgear* y cabe destacar que casi todos ellos son booleanos (simulando el estado encendido o apagado de un botón para la ejecución de distintos modos). Para facilidad a la hora de inicializar o apagar el sistema, se define una lista en *Python* llamada *vector*, la cual recogerá todos estos valores, para un procesamiento más rápido como ya se verá posteriormente.

4.2.2.2 AP

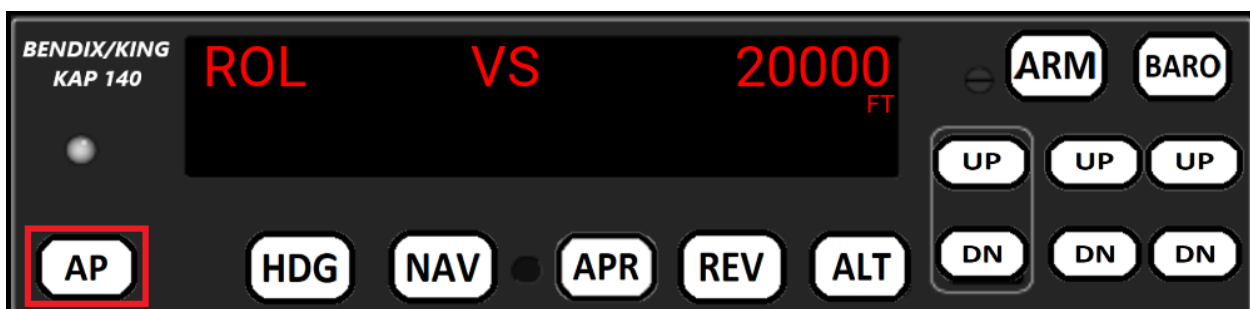


Ilustración 119 Detalle botón AP

En la Ilustración 119, se encuentra el botón AP, el cual es el responsable del encendido del autopiloto. Esta función ha sido reproducida mediante código, configurando los diferentes valores de la lista *vector* introducida anteriormente en este documento.

Será una constante dentro de este tipo de funciones ver varias partes con distintos objetivos, normalmente un

primero para cambiar la configuración, y un segundo para actualizar la interfaz gráfica.

```

369     def app_enabled(self,*args):
370
371         if sum(self.vector) == 0:
372             self.pitch_axis = 1
373             self.pitch_mode = 1
374             self.roll_axis = 1
375             self.roll_mode = 1
376
377             self.vector = [self.alt_hold,
378                           self.apr_hold,
379                           self.gs_hold,
380                           self.hdg_hold,
381                           self.nav_hold,
382                           self.pitch_arm,
383                           self.pitch_axis,
384                           self.pitch_mode,
385                           self.rev_hold,
386                           self.roll_arm,
387                           self.roll_axis,
388                           self.roll_mode]
389
390             print(self.vector)
391             self.udp_tx()
392
393             hdg_mode_text = self.ids.hdg_mode_text
394             hdg_mode_text.text = "ROL"
395             alt_mode_text = self.ids.alt_mode_text
396             alt_mode_text.text = "VS"
397             self.update_label_alt(self, *args)
398             alt_units_text = self.ids.alt_units_text
399             alt_units_text.text = "FT"
400
401             self.function_interval = Clock.schedule_interval(self.update_label_ap,0.75)
402             Clock.schedule_once(self.stop_interval_ap,5)
403
404         else:
405             self.vector = numpy.zeros(12,dtype=int)
406             self.alt_hold = 0
407             self.apr_hold = 0
408             self.gs_hold = 0
409             self.hdg_hold = 0
410             self.nav_hold = 0
411             self.pitch_arm = 0
412             self.pitch_axis = 0
413             self.pitch_mode = 0
414             self.rev_hold = 0
415             self.roll_arm = 0
416             self.roll_axis = 0
417             self.roll_mode = 0
418             print(self.vector)
419             self.udp_tx()
420             hdg_mode_text = self.ids.hdg_mode_text
421             hdg_mode_text.text = ""

```

Ilustración 120 Detalle función app_enabled(): encendido

La primera define la funcionalidad del botón, actuando de forma que si cuando se presiona dicho botón la suma de las componentes de la lista *vector* es igual a 0 (línea 371), significa que el autopiloto esta apagado y se procederá a su inicialización, que configurará los modos por defecto del autopiloto, definirá el vector, y lo enviará.

La segunda parte define lo que debe hacer la interfaz gráfica (línea 392), configurando los diferentes *labels* que deben cambiar dentro de la pantalla generada por *Kivy*.

Si, por el contrario, algún modo del autopiloto está encendido, lo cual se traduce a que la suma de las componentes de la lista es distinta de 0 (línea 403), y se procederá al apagado del autopiloto. Como se observa en la Ilustración 121, también consta a su vez de dos partes, una primera que se configura a 0 todos los modos y se envía la información a *Flightgear*, y una segunda parte donde se configura el comportamiento gráfico de la aplicación.

Cabe destacar que, tanto al encendido como al apagado, se ha replicado el comportamiento gráfico del autopiloto simulado en *Flightgear*, que hace que la *label* AP parpadee durante 5 segundos para mostrar al usuario el cambio en el funcionamiento en el sistema. Esto se puede observar tanto en las líneas 400 y 401 para la Ilustración 120, como en las líneas 441 a 446 en la Ilustración 121, en el cual, si el autopiloto se apaga posteriormente a haberse encendido dentro del lapso de tiempo en el que se ejecuta el parpadeo, para la

función anterior y la ejecuta nuevamente para mantener dichos 5.

```

403     else:
404         self.vector = numpy.zeros(12,dtype=int)
405         self.alt_hold = 0
406         self.apr_hold = 0
407         self.gs_hold = 0
408         self.hdg_hold = 0
409         self.nav_hold = 0
410         self.pitch_arm = 0
411         self.pitch_axis = 0
412         self.pitch_mode = 0
413         self.rev_hold = 0
414         self.roll_arm = 0
415         self.roll_axis = 0
416         self.roll_mode = 0
417         print(self.vector)
418         self.udp_tx()
419         hdg_mode_text = self.ids.hdg_mode_text
420         hdg_mode_text.text = ""
421         alt_mode_text = self.ids.alt_mode_text
422         alt_mode_text.text = ""
423         ap_activated_text = self.ids.ap_activated_text
424         ap_activated_text.text = ""
425         alt_number_text = self.ids.alt_number_text
426         alt_number_text.text = ""
427         alt_units_text = self.ids.alt_units_text
428         alt_units_text.text = ""
429         rev_mode_text = self.ids.rev_mode_text
430         rev_mode_text.text = ""
431
432         arm_mode_text = self.ids.arm_mode_text
433         arm_mode_text.text = ""
434         arm_mode_text_letter_a =self.ids.arm_mode_text_letter_a
435         arm_mode_text_letter_a.text = ""
436         arm_mode_text_letter_r =self.ids.arm_mode_text_letter_r
437         arm_mode_text_letter_r.text = ""
438         arm_mode_text_letter_m =self.ids.arm_mode_text_letter_m
439         arm_mode_text_letter_m.text = ""
440
441         self.stop_interval_ap()
442         if self.ids.ap_activated_text.text == "AP":
443             self.ids.ap_activated_text.text = ""
444
445         self.function_interval = Clock.schedule_interval(self.update_label_ap,0.75)
446         Clock.schedule_once(self.stop_interval_ap,5)

```

Ilustración 121 Detalle función `app_enabled()`: apagado

4.2.2.3 HDG

En esta función se han definido las funcionalidades del modo HDG o *Heading* y ROL, ambos ejecutados mediante el botón HDG. En la Ilustración 123 e Ilustración 124 se puede observar tres casuísticas, replicando así el comportamiento al lanzarse dicho modo a través de *Flightgear*.

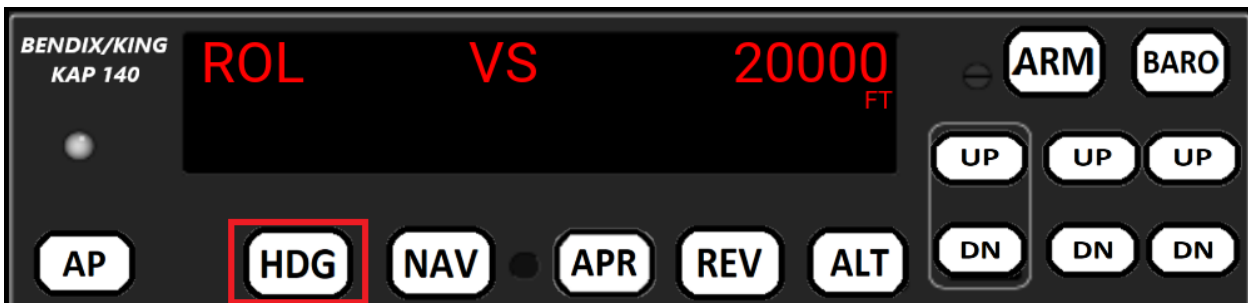


Ilustración 122 Detalle botón HDG

La primera de ellas sería que, si el autopiloto está apagado, no se realice ninguna acción. Si el modo ROL está activado, se procederá a ejecutar el modo HDG, el cual llevará a la aeronave a seguir un rumbo predefinido mediante el *HDG BUG* (el cual será introducido posteriormente). Si, en cambio, está el modo HDG activado, se procederá a ejecutarse el modo ROL, que dispondrá a la aeronave mantener un vuelo nivelado.

```
576     def hdg_enabled(self,*args):
577         # print(self.alt_hold)
578
579         if sum(self.vector) == 0:
580             None
581         elif int(self.roll_mode) == 1 or \
582             int(self.roll_mode) == 3 or \
583             int(self.roll_mode) == 5:
584             self.roll_mode = 2
585             self.hdg_hold = 1
586             self.roll_arm = 0
587             self.nav_hold = 0
588             self.apr_hold = 0
589             self.rev_hold = 0
590
591             self.vector = [self.alt_hold,
592                           self.apr_hold,
593                           self.gs_hold,
594                           self.hdg_hold,
595                           self.nav_hold,
596                           self.pitch_arm,
597                           self.pitch_axis,
598                           self.pitch_mode,
599                           self.rev_hold,
600                           self.roll_arm,
601                           self.roll_axis,
602                           self.roll_mode]
603
604             print(self.vector)
605             self.udp_tx()
606             # print(self.ids)
607             # print(self.ids.)
608             hdg_mode_text = self.ids.hdg_mode_text
609             hdg_mode_text.text = "HDG"
610             rev_mode_text = self.ids.rev_mode_text
611             rev_mode_text.text = ""
612
613
614         elif int(self.roll_mode) == 2 and int(self.hdg_hold) == 1:
```

Ilustración 123 Detalle de la función `hdg_enabled()` parte 1

Al igual que paso en códigos anteriores, consta de tres partes, una parte donde se configuran los diferentes modos de funcionamiento, un vertido de la información al simulador mediante *sockets* y la actualización de la interfaz gráfica. Esto se observa tanto en la Ilustración 123 e Ilustración 124.

El resto de la botonería de configuración del autopiloto funciona de una forma similar, donde primeramente se estudió la casuística a la ejecución de los diferentes botones, y se han replicado mediante la configuración deseada de la lista *vector*. Se tienen en cuenta algunas configuraciones que inhiben otras configuraciones, y todas estas funciones tienen una estructura similar de actualización de la lista *vector*, un lanzamiento de la información y una actualización de la interfaz gráfica.

```

614         elif int(self.roll_mode) == 2 and int(self.hdg_hold) == 1:
615
616             self.roll_mode = 1
617             self.hdg_hold = 0
618             self.target_turn_rate=0
619             self.roll_mode = 1
620             self.rev_hold = 0
621             self.vector = [self.alt_hold,
622                           self.apr_hold,
623                           self.gs_hold,
624                           self.hdg_hold,
625                           self.nav_hold,
626                           self.pitch_arm,
627                           self.pitch_axis,
628                           self.pitch_mode,
629                           self.rev_hold,
630                           self.roll_arm,
631                           self.roll_axis,
632                           self.roll_mode]
633
634             print(self.vector)
635             self.udp_tx()
636             hdg_mode_text = self.ids.hdg_mode_text
637             hdg_mode_text.text = "ROL"
638             rev_mode_text = self.ids.rev_mode_text
639             rev_mode_text.text = ""

```

Ilustración 124 Detalle de la función `hdg_enabled()` parte 2

4.2.2.4 ALT



Ilustración 125 Detalle botón ALT

Para poder controlar el perfil vertical, se recurre al modo ALT. Este modo permite seleccionar entre un modo VS (*vertical speed* o velocidad vertical en castellano), el cual fija una tasa de ascensión o de descenso que será la opción por defecto, y el modo ALT, que permite mantener la altitud actual o incrementarla en múltiplos de veinte pies. Esta función está dividida en tres casuísticas, las cuales son introducidas a continuación. e

La primera de ellas sería que, si el autopiloto está apagado, no se realice ninguna acción. Si el modo por defecto está activado, se pasará al modo ALT. La particularidad que tiene este modo de funcionamiento es que necesita leer datos desde *Flightgear*, por lo que es necesario recurrir a otro *socket* de entrada de datos. Esto es observable en la Ilustración 126, desde la línea 750 hasta la 762, en la cual se abre un *socket* específico y se recogen los datos de *Flightgear*. Gracias a este proceso, se pueden enviar al simulador principal la altitud-presión objetivo que debe seguir el autopiloto.

```

743     def alt_enabled(self,*args):
744         # print(self.alt_hold)
745         if sum(self.vector)==0:
746             None
747         elif self.pitch_mode == 1:
748             self.pitch_mode = 2
749             self.alt_hold = 1
750             try:
751                 iface=socket(AF_INET,SOCK_DGRAM)
752                 iface.setsockopt(SOL_SOCKET, SO_SNDBUF, 4096)
753                 iface.bind(in_addr)
754                 datafg, addrinput = iface.recvfrom(1024)
755                 print(datafg.decode(FORMAT))
756                 print(datafg)
757                 # iface.shutdown(socket.SHUT_RDWR)
758                 iface.close()
759                 # datafg=iface.recv(1024)
760             except:
761                 print("can't connect to udp out port" + str(in_addr))
762                 # datafg = "0,0,0,0"
763
764             # datafg=iface.recv(1024)
765             print(float(datafg.decode(FORMAT).split(',')[3]))
766             self.target_alt_pressure = float(datafg.decode(FORMAT).split(',')[3])
767             self.vector = [self.alt_hold,
768                           self.apr_hold,
769                           self.gs_hold,
770                           self.hdg_hold,
771                           self.nav_hold,
772                           self.pitch_arm,
773                           self.pitch_axis,
774                           self.pitch_mode,
775                           self.rev_hold,
776                           self.roll_arm,
777                           self.roll_axis,
778                           self.roll_mode]
779             print(self.vector)
780             self.udp_tx()
781             alt_mode_text = self.ids.alt_mode_text
782             alt_mode_text.text = "ALT"
783
784         elif self.pitch_mode == 2:

```

Ilustración 126 Detalle de la función alt_enabled() parte 1

Si el modo ALT es el que está activo, se pasará a utilizar el modo VS, configurando los parámetros en que definen los modos de funcionamiento, vertiendo la información a *Flightgear* mediante *sockets*, y actualizando la interfaz gráfica de una forma similar a la vista anteriormente.

Para poder modificar tanto la tasa de ascenso o descenso como si se quiere incrementar o disminuir la altitud objetivo, es necesario utilizar la botonería UP and DN mostradas en la Ilustración 125. Dichos botones configuran unos parámetros diferentes en función del modo vertical que este activo.

Si se está en modo VS, se modificará el ratio de presión objetivo y si se está en el modo ALT se modificará la presión objetivo. Además de ello, se hacen las fases de actualización gráfica de la interfaz y el envío de la información, como se puede observar en la Ilustración 128.


```

786     elif self.pitch_mode == 2:
787         self.pitch_mode = 1
788         self.alt_hold = 0
789
790     self.vector = [self.alt_hold,
791                   self.apr_hold,
792                   self.gs_hold,
793                   self.hdg_hold,
794                   self.nav_hold,
795                   self.pitch_arm,
796                   self.pitch_axis,
797                   self.pitch_mode,
798                   self.rev_hold,
799                   self.roll_arm,
800                   self.roll_axis,
801                   self.roll_mode]
802
803     print(self.vector)
804     self.udp_tx()
805     alt_mode_text = self.ids.alt_mode_text
806     alt_mode_text.text = "VS"

```

Ilustración 127 Detalle de la función alt_enabled() parte 2

```

807     def up_enabled(self,*args):
808         if self.pitch_mode == 1:
809             # if self.pitch_mode == 1:
810             # delta_pressure_rate= 0.001724137931
811             self.target_pressure_rate -= 0.001724137931
812             self.delta_pressure_rate_fpm += 100
813             self.udp_tx()
814
815             self.update_label_vs()
816             Clock.schedule_once(self.stop_interval_vs,5)
817         elif self.pitch_mode == 2:
818             self.target_alt_pressure -= self.delta_target_alt_pressure
819             self.udp_tx()
820
821
822     def dn_enabled(self,*args):
823         if self.pitch_mode == 1:
824             # delta_pressure_rate= 0.001724137931
825             self.target_pressure_rate += 0.001724137931
826             # delta_pressure_rate_fpm=100
827             self.delta_pressure_rate_fpm -= 100
828             self.udp_tx()
829             self.update_label_vs()
830             Clock.schedule_once(self.stop_interval_vs,5)
831         elif self.pitch_mode == 2:
832             self.target_alt_pressure += self.delta_target_alt_pressure
833             self.udp_tx()

```

Ilustración 128 Detalle de las funciones up_enabled() y dn_enabled()

4.2.2.5 BARO

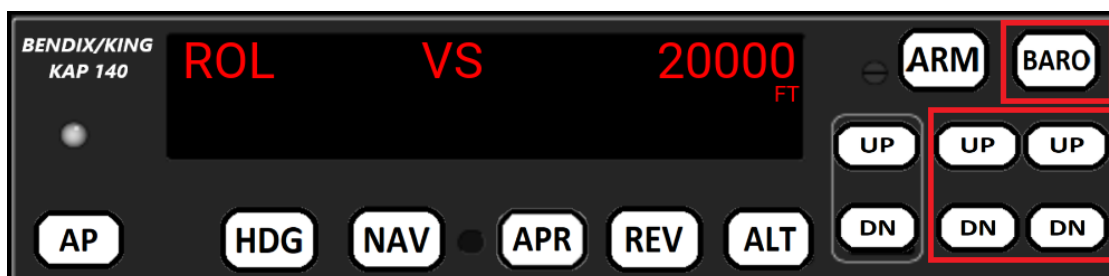


Ilustración 129 Detalle botón BARO y selectores

Dentro de la botonería del autopiloto, cabe destacar que el botón de BARO, ya que modifica la funcionalidad de la botonería que replica a las ruletas para que pasen de configurar una altitud objetivo a que pasen a configurar la el reglaje barométrico, de forma simular a lo que ocurre en el autopiloto real.

Para ello, pulsar el botón hace que se configure una bandera barométrica a uno (línea 781, *baro_flag*), la cual modificará la funcionalidad de los botones UP y DN que se alojan justo debajo del mismo. Como se puede ver en la Ilustración 130, posteriormente se ejecutará una función en segundo plano, la cual hará que a los 10 segundos de haber presionado dicho botón pase a la configuración por defecto (línea 784), cambiando el valor de dicha bandera a 0 nuevamente.

Posteriormente se observa una función típica de configuración para modificar los parámetros, típicamente con dos botones para replicar la parte principal de la ruleta, los cuales servirán para poder modificar a placer los distintos parámetros. En este caso consta de las partes típicas como ya suele ser recurrente, una que suma al parámetro original una cierta cantidad, una que comprueba si el parámetro en cuestión ha superado un cierto valor para poder volver a reiniciarlo, el envío de la información mediante el *socket* y la actualización de los valores en la interfaz gráfica.

```

779     def baro_enabled(self,*args):
780         self.update_label_baro(self, *args)
781         self.baro_flag = 1
782         print(self.baro_flag)
783         # Clock.schedule_once(self.stop_interval_vs,5)
784         self.enable_timer_label(self, *args)
785
786     def up_enabled_2(self,*args):
787         if self.baro_flag == 0:
788             self.target_alt_ft += 20
789             if self.target_alt_ft > 99999:
790                 self.target_alt_ft = 0
791             self.udp_tx()
792             # print(self.target_alt_ft)
793             self.update_label_alt(self, *args)
794         elif self.baro_flag == 1:
795             self.baro_setting_inhg += 0.01
796             if self.baro_setting_inhg >= 100:
797                 self.baro_setting_inhg = 0
798             self.udp_tx()
799             self.update_label_baro(self, *args)
800             # self.enable_timer_label(self, *args).cancel()
801             self.enable_timer_label(self, *args)
802
803     def dn_enabled_2(self,*args):

```

Ilustración 130 Detalle función `baro_enabled()` y ejemplo botón de configuración

4.2.2.6 OBS KNOB y HDG BUG

Posteriormente al autopiloto, se debían simular todo el entorno necesario para poder utilizar este de forma completa y satisfactoria. Para ello se empezó replicando el OBS knob y el HDG bug, pertenecientes a los instrumentos HSI y giro direccional. Estos instrumentos totalmente simulados se pueden observar en la Ilustración 131, y hacen de las ruletas selectoras. Gracias a estos, se pueden seleccionar los radiales que se quieren capturar, o el rumbo objetivo del autopiloto sin recurrir a la interfaz gráfica de *Flightgear*, siendo totalmente necesarios para los modos HDG, y el resto de los modos en el plano horizontal.

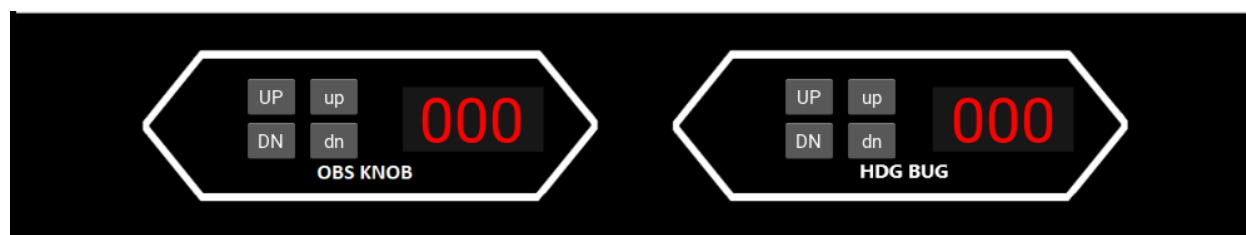


Ilustración 131 Detalle selectores simulados, OBS KNOB y HDG BUG

En la Ilustración 132, se pueden observar unas capturas en las cuales en la esquina inferior izquierda y derecha respectivamente para el OBS knob y el HDG bug, las ruletas reales que han tenido que ser rediseñadas para adaptarse a la herramienta *Kivy*, pero con idéntica funcionalidad.



Ilustración 132 Detalle HSI y giro direccional respectivamente

El diseño del código para ambos ha sido similar a lo que se ha visto en sistemas anteriores. Consta de una parte de ejecución, en la cual actualiza el valor anterior del ángulo objetivo y se compara para ver si ha superado los límites, para restablecer su valor. Posteriormente, se envía la información a *Flightgear*, y se actualizan las la interfaz gráfica.

Estos instrumentos son totalmente necesarios para los modos de funcionamiento del autopiloto HDG, NAV, APR y REV, y es la razón por lo que se decidió incluirlos en el panel.

```

870     def dn1_obs_knob(self,*args):
871         self.obs_knob -= 10
872         if self.obs_knob < 0:
873             self.obs_knob += 360
874         self.udp_tx()
875         self.update_label_obs_knob(self, *args)
876
877     def dn2_obs_knob(self,*args):
878         self.obs_knob -= 1
879         if self.obs_knob < 0:
880             self.obs_knob += 360
881         self.udp_tx()
882         self.update_label_obs_knob(self, *args)
883
884     def up1_hdg_bug(self,*args):
885         self.hdg_bug += 10
886         if self.hdg_bug >= 360:
887             self.hdg_bug -= 360
888         self.udp_tx()
889         self.update_label_hdg_bug(self, *args)
890
891     def up2_hdg_bug(self,*args):
892         self.hdg_bug += 1
893         if self.hdg_bug >= 360:
894             self.hdg_bug -= 360
895         self.udp_tx()
896         self.update_label_hdg_bug(self, *args)

```

Ilustración 133 Detalle funciones de configuración OBS KNOB y HDG BUG

4.2.2.7 Selector de frecuencias, indicador DME y adquisición de la información

Al igual que paso con los selectores de OBS y HDG, para poder cambiar entre distintas frecuencias de navegación, es estrictamente necesario poder hacerlo con cierta facilidad a la hora de efectuar un vuelo instrumental. Para ello se decidió de incluir dicho instrumento dentro del panel, y finalmente se incluyeron ambos selectores, el principal y el secundario, por una cuestión meramente estética, ya que el selector de OBS, estará solamente conectado a la frecuencia de navegación del selector principal, en este prototipo.

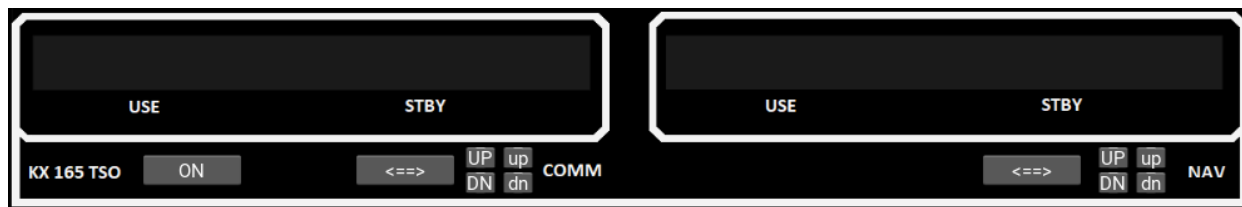


Ilustración 134 Detalle selectores de frecuencias de comunicación y de navegación

Se observa en la Ilustración 134 que el diseño de dicho selector es una réplica de sistema real alojado en el modelo de la Cessna 172P, aunque se han introducido una modificación, convirtiendo la ruleta en botonería, por la misma razón comentada anteriormente.

Cada uno de los botones se ha configurado de forma que se replique el funcionamiento del instrumento alojado en *Flightgear*. Primeramente, se tiene en cuenta si el instrumento está encendido, y en el caso contrario, no se realiza ninguna acción. Si este lo está, cada uno de los botones sirve para seleccionar una frecuencia de navegación o de comunicación, teniendo cada uno de los botones distintas precisiones. Además, existe un botón para poder intercambiar las frecuencias de la principal a la secundaria, de forma similar al sistema real.

Cabe destacar que dichos selectores solo dejan seleccionar frecuencias dentro del ancho de banda permitido para cada una de las funcionalidades, estableciendo límites superiores e inferiores. Esto se puede observar en la Ilustración 135, y como posteriormente se hace una actualización de las *labels* y se envía la información a *Flightgear*.

```

914     def up1_comml(self,*args):
915         if self.radl_on ==0:
916             None
917         elif self.radl_on == 1:
918             self.freq1_ent_comml += 1
919             if self.freq1_ent_comml > 136:
920                 self.freq1_ent_comml -= 19
921             self.freq1_comml = self.freq1_ent_comml + (self.freq1_dec_comml/100)
922             self.update_label_freq1_comml(self, *args)
923             # print(self.freq1_comml)
924             # self.update_label_freq1_comml(self, *args)
925             self.udp_tx()
926
927     def up2_comml(self,*args):
928         if self.radl_on == 0:
929             None
930         elif self.radl_on == 1:
931             self.freq1_dec_comml += 5
932             if self.freq1_dec_comml == 100:
933                 self.freq1_dec_comml = 0
934                 self.freq1_comml = float(self.freq1_ent_comml)
935                 self.update_label_freq1_comml(self, *args)
936                 # self.update_label_freq1_comml(self, *args)
937                 # print(self.freq1_comml)
938                 self.udp_tx()
939             else:
940                 self.freq1_comml = self.freq1_ent_comml + (self.freq1_dec_comml/100)
941                 self.update_label_freq1_comml(self, *args)
942                 # print(self.freq1_comml)
943                 # self.update_label_freq1_comml(self, *args)
944                 self.udp_tx()

```

Ilustración 135 Detalle funciones de configuración de las frecuencias

Si se está conectado a una baliza DME, el indicador DME mostrará por pantalla las millas en línea recta desde la aeronave hasta la baliza, y el tiempo de vuelo a la estación en cuestión de minutos o la velocidad respecto a esta. En las Ilustración 136 e Ilustración 137 se observa un ejemplo de ambas casuísticas.

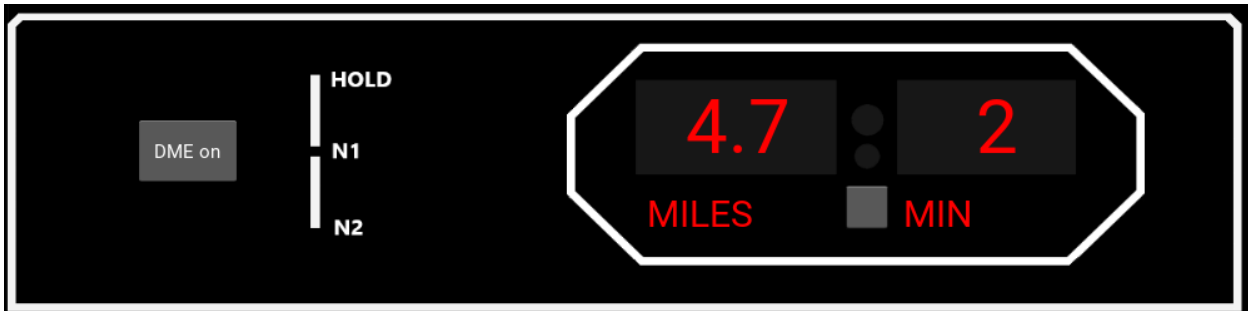


Ilustración 136 Detalle indicador DME, estimación de tiempo



Ilustración 137 Detalle indicador DME, estimación de velocidad

Para replicar dichas funcionalidades, se recurre a la función `dme_enable` y `dme_min2kt`. La primera de estas activa el indicador DME, y lo ejecuta por defecto. Para ello, al activarlo ejecuta una función en segundo plano, la cual se ejecuta a un ritmo de 0.75 segundos, como se puede observar en la línea 247 de la Ilustración 138. Al utilizar esta función una segunda vez, se cancela la ejecución de dicha función y se ponen los visores a cero.

```

245     def dme_enable(self, *args):
246         if self.dme_on == 0:
247             self.dme_update = Clock.schedule_interval(self.update_label_dme,0.75)
248             self.dme_on = 1
249         elif self.dme_on == 1:
250             self.dme_update.cancel()
251             self.dme_on = 0
252             self.ids.dme_miles_text.text = ""
253             self.ids.dme_minkt_text.text = ""
254
255     def dme_min2kt(self, *args):
256         if self.dme_min2ktflag == 0:
257             self.dme_min2ktflag = 1
258             # print(2)
259             # print(str(self.ids.dme_unit2_text.text))
260             self.ids.dme_unit2_text.text = "KTS"
261         elif self.dme_min2ktflag == 1:
262             self.dme_min2ktflag = 0
263             print(str(self.ids.dme_unit2_text.text))
264             self.ids.dme_unit2_text.text = "MIN"

```

Ilustración 138 Detalle función `dme_enable()` y `dme_min2kt()`

Para completar la funcionalidad, se añade la función `dme_min2kt` para poder cambiar la selección de minutos a nudos mediante una serie de banderas y cambiarán la leyenda del indicador (*KTS* o *MIN*). Estas se tendrán en cuenta en la función `update_label_dme()`, la cual se encarga de leer los datos provenientes de *Flightgear* mediante un *socket* de entrada, mostrará por pantalla las millas y en función de la opción seleccionada mostrara la estimación de tiempo o la velocidad respecto la baliza.

```

223     def update_label_dme(self, *args):
224         try:
225             iface=socket(AF_INET,SOCK_DGRAM)
226             iface.setsockopt(SOL_SOCKET, SO_SNDBUF, 4096)
227             iface.bind(in_addr)
228             datafg, addrinput = iface.recvfrom(1024)
229             # print(datafg.decode(FORMAT))
230             print(datafg)
231             # iface.shutdown(socket.SHUT_RDWR)
232             iface.close()
233         except:
234             print("can't connect to udp out port" + str(in_addr))
235
236
237         self.ids.dme_miles_text.text = datafg.decode(FORMAT).split(',')[2]
238
239         if self.dme_min2ktflag == 0:
240             self.ids.dme_minkt_text.text = datafg.decode(FORMAT).split(',')[1]
241         elif self.dme_min2ktflag == 1:
242             self.ids.dme_minkt_text.text = datafg.decode(FORMAT).split(',')[0]

```

Ilustración 139 Detalle función `update_label_dme()`

4.2.2.8 Funciones auxiliares y envío de la información

A parte de todas las funcionalidades que se han explicado anteriormente, se han codificado otras funciones auxiliares más enfocadas a la actualización de las *labels* que tienen detrás un cierto tipo de condicionantes, como se puede observar en la Ilustración 140.

```

130 # =====
131 #           Updating autopilot labels
132 # =====
133     def update_label_ap(self, *args):
134         if self.ids.ap_activated_text.text == "":
135             self.ids.ap_activated_text.text = "AP"
136         elif self.ids.ap_activated_text.text == "AP":
137             self.ids.ap_activated_text.text = ""
138
139     def stop_interval_ap(self, *args):
140         self.function_interval.cancel()
141
142     def update_label_vs(self, *args):
143
144         number = abs(self.delta_pressure_rate_fpm)
145         sign = numpy.sign(self.delta_pressure_rate_fpm)
146
147         if len(str(number)) == 1:
148             self.ids.alt_number_text.text = "0000"+str(number)
149             self.ids.alt_units2_text.text = "FPM"
150         elif len(str(number)) == 3:
151             if sign == 1:
152                 self.ids.alt_number_text.text = "00" + str(number)
153                 self.ids.alt_units2_text.text = "FPM"
154             elif sign == -1:
155                 self.ids.alt_number_text.text = "-00" + str(number)
156                 self.ids.alt_units2_text.text = "FPM"
157         elif len(str(number)) == 4:
158             if sign == 1:
159                 self.ids.alt_number_text.text = "0" + str(number)
160                 self.ids.alt_units2_text.text = "FPM"
161             elif sign == -1:
162                 self.ids.alt_number_text.text = "-0" + str(number)
163                 self.ids.alt_units2_text.text = "FPM"
164         elif len(str(number)) == 5:
165             if sign == 1:
166                 self.ids.alt_number_text.text = "" + str(number)
167                 self.ids.alt_units2_text.text = "FPM"
168             elif sign == -1:
169                 self.ids.alt_number_text.text = "-" + str(number)
170                 self.ids.alt_units2_text.text = "FPM"
171         self.ids.alt_units_text.text = ""
172

```

Ilustración 140 Detalle función auxiliar

Esto resulta especialmente útil a la hora de reutilizar código para actualizar las diferentes *labels*, ya que cambiando simplemente los parámetros y siguiendo una buena estructura se pudo hacer un trabajo tedioso en relativamente poco tiempo.

4.2.2.9 Función para el envío de información

La última función y la más crítica es la encargada de enviar la información. Como ya se adelantó en el punto [4.2.2.1](#), cada vez que se envía información a *Flightgear* mediante sockets, se debe configurar este a priori y hacer coincidir con las estructuras definidas en los archivos XML alojados en la carpeta *protocols*.

Esto se puede observar desde las líneas 1154 a 1183 de la Ilustración 141, en la cual se organizan los parámetros que se van a enviar en forma de texto, los cuales posteriormente serán transformados dentro de *Flightgear* al tipo de variable deseada.

Una vez que se tiene el mensaje estructurado y actualizado, mediante el comando *try* codifica el mensaje en el formato correcto, y se intenta enviar al *socket* la información necesaria. Si es imposible establecer la conexión con ese *socket*, se mostrará un error por pantalla para que se tenga en cuenta, y se tomen medidas.

```

1151     def udp_tx(self,*args):
1152         global outputline
1153
1154         outputline = str(self.alt_hold) + "," + "\
1155                     str(self.apr_hold) + "," + "\
1156                     str(self.gs_hold) + "," + "\
1157                     str(self.hdg_hold) + "," + "\
1158                     str(self.nav_hold) + "," + "\
1159                     str(self.pitch_arm) + "," + "\
1160                     str(self.pitch_axis) + "," + "\
1161                     str(self.pitch_mode) + "," + "\
1162                     str(self.rev_hold) + "," + "\
1163                     str(self.roll_arm) + "," + "\
1164                     str(self.roll_axis) + "," + "\
1165                     str(self.roll_mode) + "," + "\
1166                     str(self.target_turn_rate) + "," + "\
1167                     str(self.target_pressure_rate) + "," + "\
1168                     str(self.delta_pressure_rate_fpm) + "," + "\
1169                     str(self.baro_setting_hpa) + "," + "\
1170                     str(self.baro_setting_inhg) + "," + "\
1171                     str(self.target_alt_ft) + "," + "\
1172                     str(self.target_alt_pressure) + "," + "\
1173                     str(self.freq2_nav1) + "," + "\
1174                     str(self.freq1_nav1) + "," + "\
1175                     str(self.freq2_nav2) + "," + "\
1176                     str(self.freq1_nav2) + "," + "\
1177                     str(self.freq2_comml) + "," + "\
1178                     str(self.freq1_comml) + "," + "\
1179                     str(self.freq2_comm2) + "," + "\
1180                     str(self.freq1_comm2) + "," + "\
1181                     str(self.hdg_offset) + "," + "\
1182                     str(self.hdg_bug) + "," + "\
1183                     str(self.dme_on) + "\n"
1184
1185         #
1186         #     print outputline
1187         #     print(outputline)
1188
1189         try:
1190             # UDPSock.connect(addr)
1191             outputline = outputline.encode(FORMAT)
1192             UDPSock.sendto(outputline,addr)
1193         except:
1194             print("can't connect to udp out port" + str(addr))

```

Ilustración 141 Detalle función `udp_tx()`

4.3. Resultados y conclusiones

El código se fue testeando poco a poco a medida que se fue desarrollando cada una de las funcionalidades siguiendo una estructura modular, haciendo que el resultado final fuera bastante robusto. Tras muchos intentos, mucha ingeniería inversa, planificación y trabajo, se consiguió un panel funcional en el cual estuvieran representadas muchas de las funcionalidades objetivo.

Como se observa en la Ilustración 142, el envío de la información se puede efectuar correctamente, gracias a la definición del *socket*. Aparecen en la imagen tanto la lista *vector* como el mensaje final que se envía al simulador. Esto hace que, mediante estas entradas, se puede seleccionar muchas de las funcionalidades internas de *Flightgear*, y siembra un precedente para poder reutilizar y reciclar código de esta herramienta para implementar otras herramientas propias para desarrollar una cabina completa.

```
[INFO ] [Text      ] Provider: sdl2
[INFO ] [Base       ] Start application main loop
[0, 0, 0, 0, 0, 0, 1, 1, 0, 0, 1, 1]
0,0,0,0,0,0,1,1,0,0,1,1,0,0,0,1.013207589,29.92,20000,27.23,113.0,108.95,108.0,113.95,118.0,1
18.95,126.0,118.95,0,0,0
[0 0 0 0 0 0 0 0 0 0 0 0]
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1.013207589,29.92,20000,27.23,113.0,108.95,108.0,113.95,118.0,1
18.95,126.0,118.95,0,0,0
```

Ilustración 142 Detalle envío de la información

Para exponer los resultados, se irán desglosando una por una todas las funcionalidades que se han conseguido implementar en dicho panel, y las dificultades que se han encontrado durante el desarrollo.

Tanto el encendido como el apagado de los sistemas se ha implementado de una forma sencilla, ya que simplemente mediante los *sockets* se pueden modificar. Por lo que la implementación de los selectores de frecuencia los selectores de OBS y HDG o la altitud objetivo y el reglaje aerodinámico ha sido implementado sin mayores complicaciones.

A la hora de replicar los modos del autopiloto, ha habido una mayor dificultad detrás, ya que primeramente se requiere conocer el sistema para poder replicar su funcionalidad de la manera más fehaciente posible.

Dentro del autopiloto, la funcionalidad más sencilla de replicar fue la del botón HDG, donde tanto los modos de mantener un vuelo nivelado como la captura de un rumbo especificado se pueden especificar simplemente cambiando unos parámetros dentro del árbol de propiedades de *Flightgear*. En la Ilustración 143 se puede observar el detalle de la traza de la trayectoria seguida por el avión, habiendo utilizado la aplicación para comandar las acciones, habiéndose utilizado la aplicación *Phi* para la representación.



Ilustración 143 Detalle cambios de rumbo

Cabe destacar que en función de qué tipo de rumbo se quiera seguir, magnético o verdadero, habrá que ajustar primeramente la declinación magnética en el offset del giro direccional. En la ilustración anterior se puede observar la diferencia entre ambos casos.

A la hora de efectuar diferentes capturas de radiales, el procedimiento varía. Primeramente, lo más importante sería seleccionar la frecuencia de la estación VOR a la cual nos queramos referir, como por ejemplo la frecuencia seleccionada en este caso de 113.9 MHz, como se observa en la Ilustración 144.

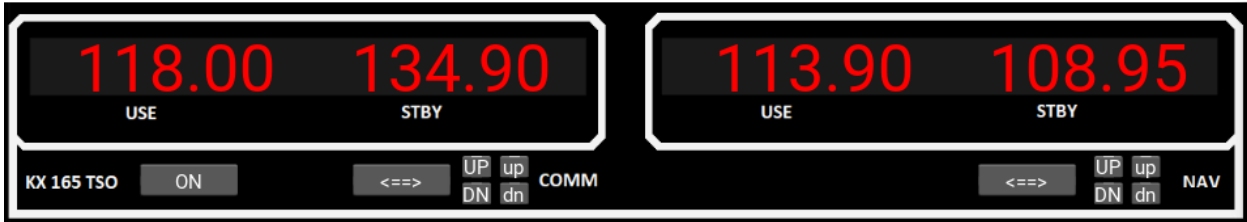


Ilustración 144 Detalle selector de frecuencias

Una vez seleccionada la frecuencia deseada, previamente de utilizar el modo NAV, se deben manipular los selectores OBS y HDG para poder capturar el radial deseado, estando ambos al mismo valor. Cuando se haya finalizado la selección, se pasará a activar dicha funcionalidad mediante la pulsación del botón NAV, el cual solo se ejecutará si previamente el modo HDG estaba activo.

La aeronave pasará entonces a capturar el radial deseado gracias al HSI (*Horizontal Situation Indicator*), el cual proporcionará la información de la deriva respecto al radial deseado, la cual el autopiloto tratará de mitigar. En la Ilustración 145 se puede observar la traza de la trayectoria de una aeronave capturando varios radiales mediante la aplicación.

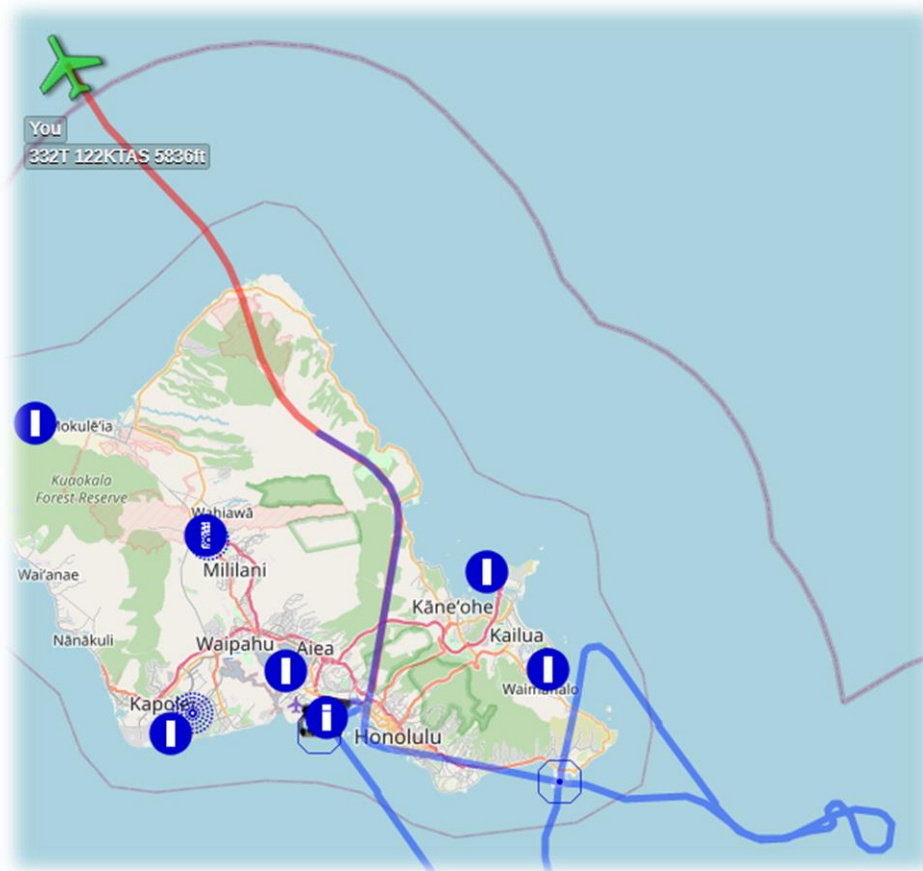


Ilustración 145 Detalle capturas diferentes radiales

Al pasar por la antena VOR, si se desea capturar otro radial, simplemente se deberá esperar a que la bandera TO/FROM cambie de TO a FROM para seleccionar en ambos selectores OBS y HDG el nuevo radial a capturar. Se puede observar en la Ilustración 146 dicho comportamiento, y la precisión a la hora de la captura. Cabe destacar que los ángulos de intercepción serán por definición 45° .

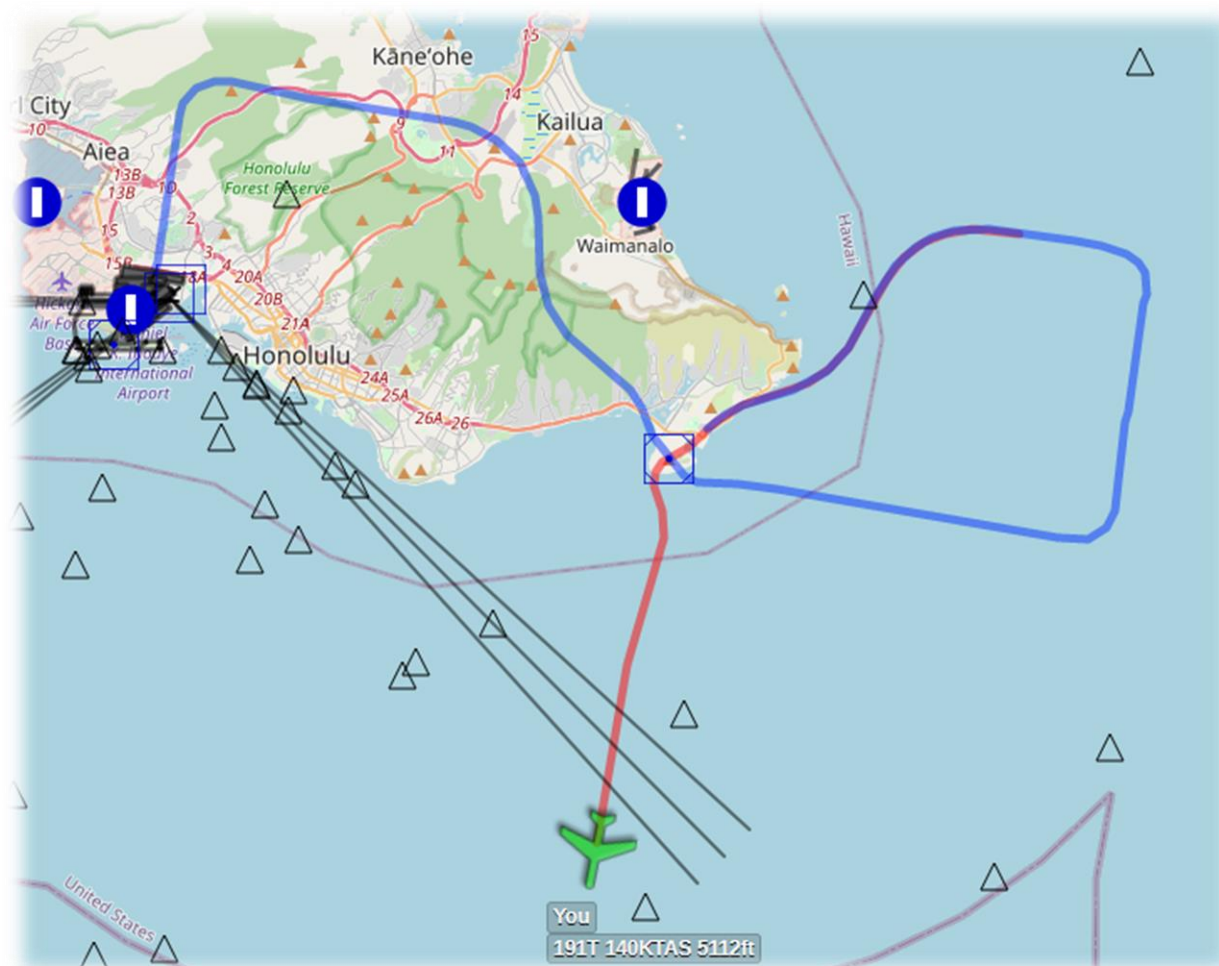


Ilustración 146 Detalle capturas diferentes radiales

Una vez implementados dichos modos, se pasó a implementar los modos APR y REV, especialmente útiles a la hora de aproximaciones instrumentales. El procedimiento es relativamente sencillo a la hora de capturar la senda de planeo. Primeramente, se deberá seleccionar la frecuencia ILS para poder utilizar el HSI, como por ejemplo la frecuencia 113.9 MHz mostrada en el selector de frecuencias de navegación de la Ilustración 144.

En la Ilustración 147, donde se puede observar la traza de la trayectoria de una maniobra de aproximación. La primera parte de la maniobra fue ejecutada con el modo back course (botón REV) el cual captura la senda de planeo de forma contraria. Para ejecutarlo, se deberá seleccionar el radial deseado, configurando los selectores OBS y HDG respectivamente en el panel, en este caso 233° . Una vez capturada la senda de planeo, se hará una maniobra de ruptura a 45° , usando el modo HDG del mismo autopiloto.

Una vez alcanzado el punto deseado, se hace una maniobra de aproximación usando el modo HDG de nuevo, girando 180° el selector HDG. Una vez que se encara la senda de planeo, se configura el selector OBS y el HDG para capturarla, en este caso a 53° , que será efectuado mediante el modo APR.

Una vez configurado, se deberán seguir las PAPIs para seguir la senda de planeo en el eje vertical. En la Ilustración 147 se puede observar dicha aproximación. Cabe señalar que todas las maniobras fueron ejecutadas desde el panel implementado en este TFM.

Se pueden observar también oscilaciones, fruto de la distancia a la antena y de la velocidad a la hora de la

captura.

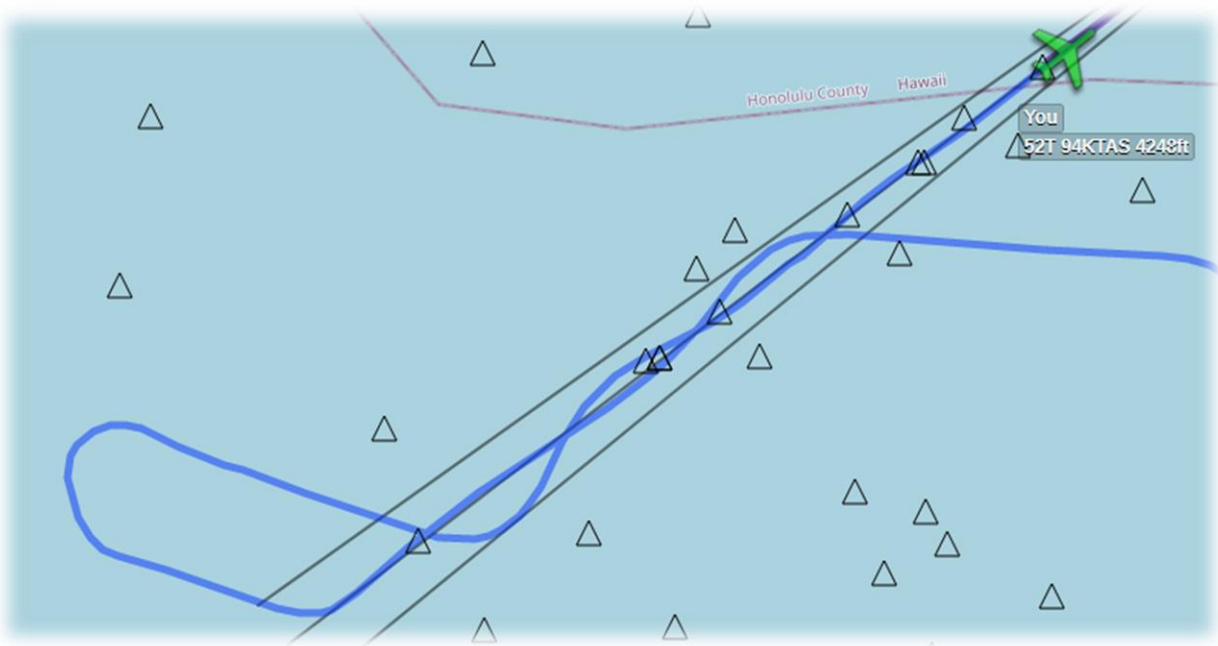


Ilustración 147 Detalle traza maniobra de aproximación



Ilustración 148 Detalle aproximación instrumental en *Flightgear*

Estos modos se pueden utilizar de una forma similar para maniobras típicas del vuelo instrumental, como puede ser un hipódromo, ejemplificado en la Ilustración 149.

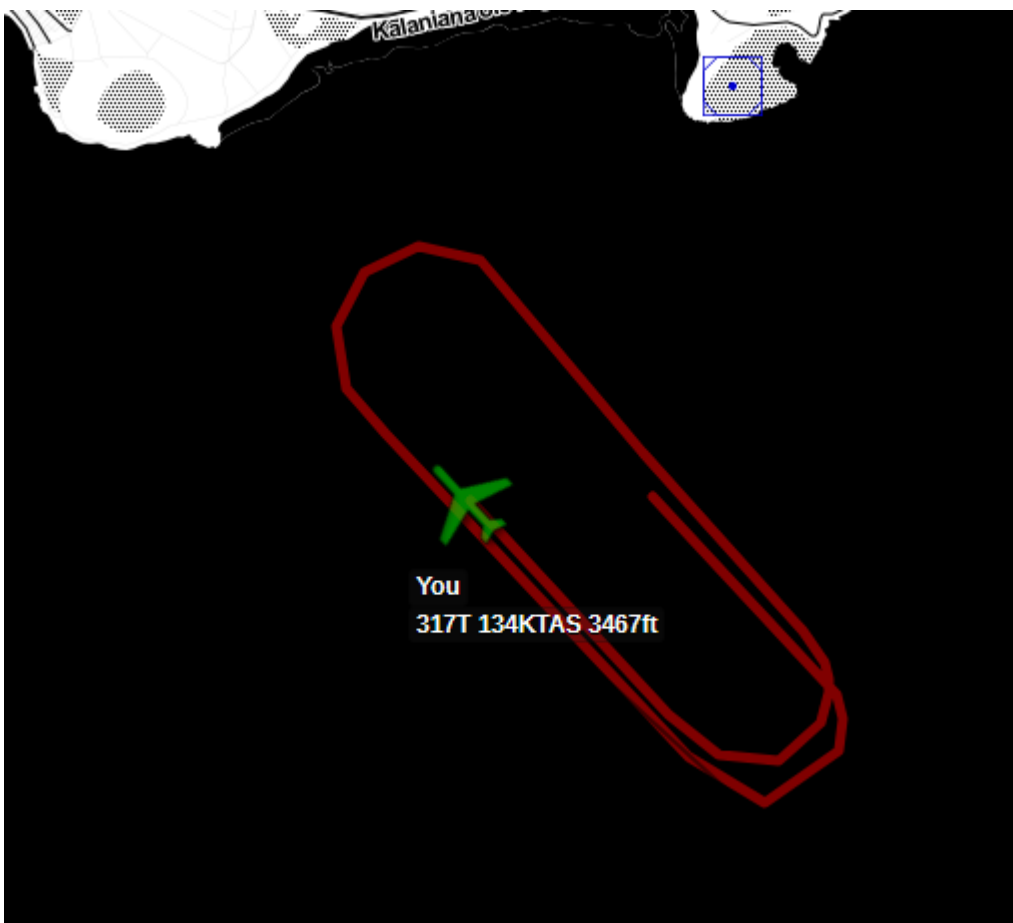


Ilustración 149 Detalle traza de un hipódromo de espera

A la hora de manejar los modos verticales se tienen dos opciones. El modo por defecto, el cual fija una velocidad vertical (VS) en múltiplos de 100 FPM (pies por minuto) o el modo ALT, el cual mantendrá la altitud actual y se podrá modificar en múltiplos de 20 pies.

Aquí se encontró la última limitación, posteriormente encontrada también en la implementación del instrumento DME (Distance Measuring Equipment) ya que se era incapaz de abrir un socket de entrada y de salida coetáneamente, pero esto finalmente fue resuelto pudiéndose leer la presión actual y los datos DME. Aun así, los modos verticales están completamente operativos.



Ilustración 150 Detalle set-up final

Finalmente se puede observar en la Ilustración 150 el *set-up* final con las herramientas disponibles para poder montar el simulador totalmente funcional. En ella se observan tanto *Flightgear* como interfaz gráfica, PHI, *c172p-WebPanel* y el panel desarrollado en este TFM.

Con dichas herramientas, se podría construir una cabina totalmente funcional, ya que se pueden realizar tanto hacer vuelos instrumentales y manuales simplemente añadiendo un *joystick* y una palanca de gases (o utilizando *Flightgear TQPanel* para simularlo vía software). Teniendo en cuenta todo el *know-how* generado en este Trabajo de Fin de Master, este panel puede modificarse y mejorarse, abriendo múltiples opciones para poder simular distintos tipos de avión y construir en *Python* diferentes sistemas, como pueden ser sistemas de seguridad de *autothrust* o limitaciones de ángulos de ataque para evitar entradas en pérdida.

Una vez concluido el trabajo, se pasará a un balance de ventajas e inconvenientes, seguido de unas conclusiones.

4.2.3 Ventajas e inconvenientes

4.2.3.1 Ventajas

Modularidad y escalabilidad

Tomando esta herramienta como plantilla, se puede seguir creando nuevas herramientas útiles para la simulación de vuelo, cumpliendo uno de los objetivos de este Trabajo fin de Master (un simulador modular).

Teniendo como referencia *Flightgear* y este TFM, se pueden hacer múltiples herramientas de este estilo, teniendo una estructura de clases en la que solo cambiaría la interfaz gráfica pero donde las funcionalidades podrían variar de una herramienta a otra con relativo poco trabajo. Así, por ejemplo, se podrían reproducir los paneles superiores de la cabina, sin demasiada complicación.

Código libre

El hecho de que toda esta herramienta esté basada en *Python* permite a la universidad seguir produciendo y desarrollando por esta vía sin necesidad de una inversión económica extra, y permitiendo que estas herramientas puedan ser distribuidas a los alumnos sin costes adicionales, enriqueciendo así la docencia.

Además, permite reutilizar miles de librerías disponibles gratuitamente, que pueden servir para montar otro tipo de herramientas, como puede ser una herramienta que dibuje las gráficas de los parámetros para poder así analizar las maniobras.

Valor añadido y viabilidad

Al poder centrarse en añadir funcionalidades, solo hace falta conocer el sistema en particular que se está replicando para poder centrarse en él. Gracias a esta especialización, los trabajos son mucho más abarcables y se puede medir mejor los objetivos, por lo que mejora la viabilidad.

Además de esto, se pueden implementar diferentes sistemas para mejorar las funcionalidades de los aviones, como generar sistemas de auto empuje y limitaciones en las deflexiones de las superficies aerodinámicas para evitar situaciones de entrada en pérdida.

4.2.3.2 Desventajas

Perdida de contexto

Esta parte del proyecto está muy centrada en la rama de la estación de un simulador, por lo que se pierde un poco de contexto de la dinámica de vuelo, como podría ser cuando se desarrolló la herramienta *Matlab/Simulink*. Si el objetivo es conocer de una forma profunda las ecuaciones o el modelaje de una aeronave, serían más útiles otras herramientas no de tan alto nivel como *Flightgear*.

Limitaciones Python

El hecho de utilizar este lenguaje hace que haya una serie de limitaciones intrínsecas a él, como por ejemplo dificultades a la hora de enviar y recibir información coetáneamente. Las aplicaciones HTML/JavaScript/CSS, mucho más enfocadas en la gestión de *socket*, son mucho más efectivas y potentes, como se puede observar en por ejemplo la herramienta *Phi*.

Esto ha causado dificultades en la implementación de funcionalidades de la herramienta, como por ejemplo siendo imposible leer los datos del DME.

5 CONCLUSIONES

Durante todo este proyecto se ha seguido el mismo objetivo para construir un simulador con las herramientas disponibles en la red con el fin de alcanzar el objetivo de poder reproducir un vuelo instrumental. Para ello, se han investigado muchas vías diversas, las cuales no eran las más óptimas para dicha empresa, pero que abren un mundo de posibilidades para la docencia.

Desde que se expone la estructura teórica del simulador, se han atacado sus dos vertientes, la estación y la simulación, como dos partes indispensables de lo mismo. Para desarrollar la parte de la simulación se ha visto que *Matlab/Simulink*, aunque es una herramienta ampliamente utilizada en la industria y muy potente, se ve limitada su escalabilidad. Todo esto sin tener en cuenta que su distribución es complicada en un mundo de escasos recursos.

En cambio, gracias a *Python*, se pueden desarrollar de forma totalmente gratuita modelos dinámicos de vuelo y otras interfaces, pero también se puede reutilizar trabajo ya hecho, para poder apostar por el valor añadido. Eso permite al usuario tener una herramienta robusta, como lo es *JSBSim*, y centrarse en diseñar sus propias aeronaves y sistemas de una forma sencilla e intuitiva, con las estructuras de los archivos de configuración. Desde la docencia, hace posible integrar el diseño de las aeronaves con unos medios que permiten testear los resultados.

De esta forma, basado en este modelo dinámico, se puede desarrollar toda la parte de la simulación, diseñando y testeando todo el entorno de sistemas. Esto puede ser un trabajo que puede ser planificado e implementado de una forma sencilla, perfecto para un simulador colaborativo como el que se propuso en inicio.

Pero si por el contrario lo que se quiere hacer es centrarse en el pilotaje, se ha visto que es posible utilizar estas herramientas código libre para poder montar cabinas sin necesidad de muchos recursos reutilizando interfaces gráficas y sistemas ya producidos. Gracias a *Python*, una vez más, y las librerías que este aporta, se pueden generar nuevas herramientas gráficas que permitan hacer de la experiencia de la simulación algo mucho más real y cercano, y acercarlas al gran público mediante su distribución gratuita.

Aunque también se ha concluido para dichos fines, existen herramientas mucho más potentes y adecuadas para poder conectar varios sistemas a un computador principal, como lo es HTML/JavaScript/CSS. Esta herramienta está enfocada a la transmisión y lectura de datos, por lo que lo hace óptima para este tipo de herramientas.

Este proyecto deja muchas puertas abiertas al futuro, ya que en base a este se puede implementar toda la cabina de un simulador modularmente. Cada una de las funciones puede ejecutarse desde un aparato independiente, estando conectadas mediante una red LAN. Esto permitiría externalizar de *Flightgear* todos los sistemas, y desarrollarse uno por uno a la par de su interfaz gráfica. De esta forma, solo se utilizaría *Flightgear* como interfaz gráfica del mundo, pudiéndose así investigar, experimentar y desarrollar diferentes sistemas, como por ejemplo leyes de control de vuelo o protecciones basados en *Python*.

Otra línea futura en busca de la portabilidad, sería implementar mediante HTML/JavaScript/CSS lo que se ha programado en *Kivy* en este TFM, dando la posibilidad de ejecutar esta herramienta desde un teléfono móvil sin necesidad de ninguna aplicación preinstalada.

REFERENCIAS

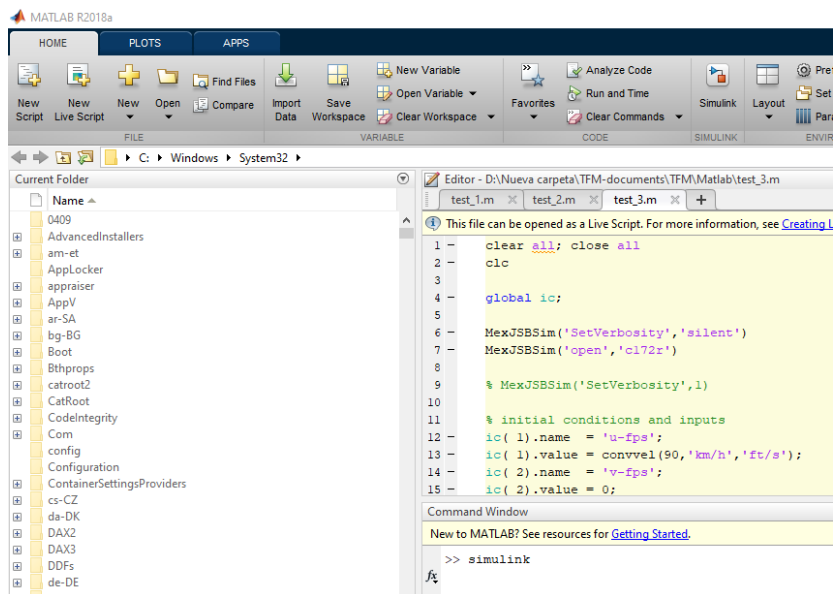
- [1] J. S. B. & t. J. Development, «JSBSim: AN open source, platform-independent, flight dynamics model in C++,» 2011. [En línea]. Available: <http://jsbsim.sourceforge.net/>.
- [2] R. & M. T. Tanner, «Stability and control derivative estimates obtained from flight data for the Beech 99 aircraft,» NTRS - NASA Technical Reports Server, 1979.
- [3] Autor, «Este es el ejemplo de una cita,» *Tesis Doctoral*, vol. 2, nº 13, 2012.
- [4] O. Autor, «Otra cita distinta,» *revista*, p. 12, 2001.
- [5] «Kivy,» The Kivy Authors, 2018. [En línea]. Available: <https://kivy.org/#aboutus>.
- [6] D.-l. & Michat, «Flightgear TQPanel,» 2014. [En línea]. Available: http://wiki.flightgear.org/FlightGear_TQPanel.
- [7] «Amateur flight simulation,» 08 11 2020. [En línea]. Available: https://en.wikipedia.org/wiki/Amateur_flight_simulation.
- [8] «Antoinette (manufacturer),» 07 11 2020. [En línea]. Available: [https://en.wikipedia.org/wiki/Antoinette_\(manufacturer\)](https://en.wikipedia.org/wiki/Antoinette_(manufacturer)).
- [9] «Distributed Interactive Simulation,» 08 11 2020. [En línea]. Available: https://en.wikipedia.org/wiki/Distributed_Interactive_Simulation.
- [10] «Flight simulator,» 07 11 2020. [En línea]. Available: https://en.wikipedia.org/wiki/Flight_simulator.
- [11] «High Level Architecture,» 08 11 2020. [En línea]. Available: https://en.wikipedia.org/wiki/High_Level_Architecture.
- [12] «Link Trainer,» 07 11 2020. [En línea]. Available: https://en.wikipedia.org/wiki/Link_Trainer.

ANEXO A: MANUAL DE DESPLIEGUE

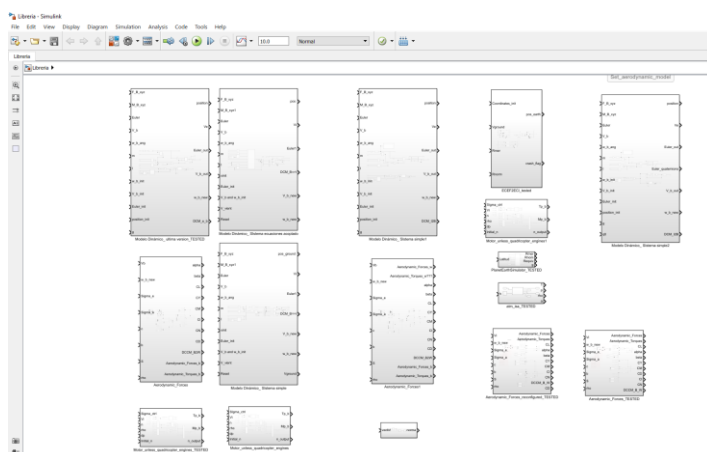
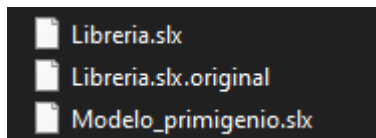
En este anexo se va a recopilar todos los pasos necesarios para poder ejecutar en un sistema operativo Windows las diferentes herramientas mostradas en este TFM.

A. Primeramente, se abordará *Matlab/Simulink*.

1. Instalar el programa
2. Ejecutar *Matlab*
3. Ejecutar *Simulink* dentro de *Matlab* mediante el comando “*simulink*”



4. Abrir el archivo .slx deseado



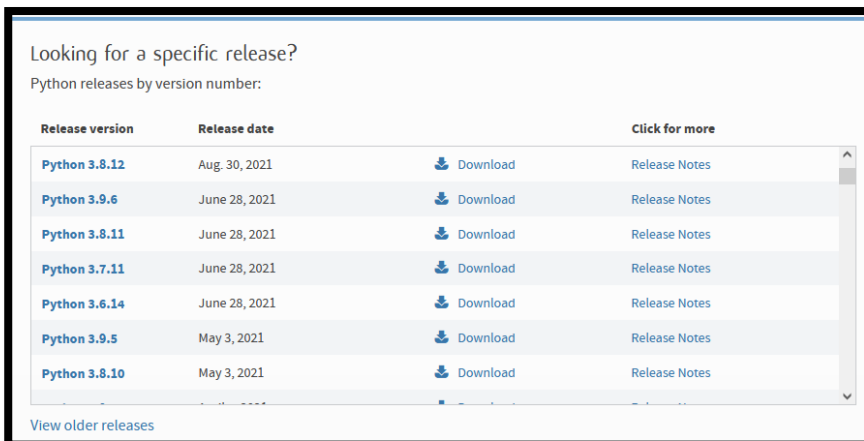
B. Ahora se abordará la instalación de Python y las herramientas necesarias para la instalación de sus librerías. En este caso se abordará como instalarlo en Windows, si se quiere utilizar en Linux los pasos son equivalentes pero ejecutados de una forma distinta.

1. Descargar la versión de Python deseada desde la pagina oficial

<https://www.python.org/downloads/windows/>

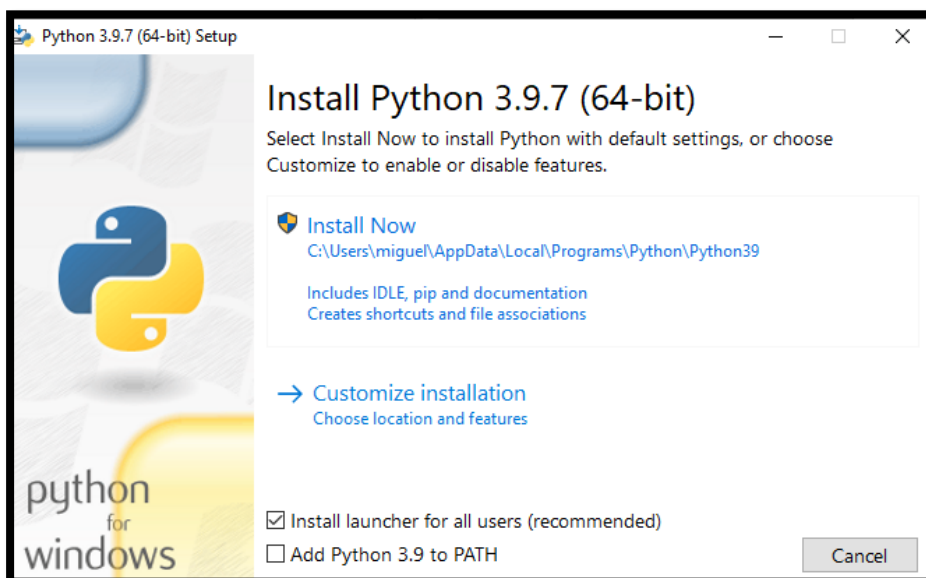


Para correr *JSBSim* será necesario Python 3, y no necesariamente se tiene que descargar la última versión ya que *JSBSim* tiene diferentes librerías en función de la versión de Python que se tenga. Para descargar una versión específica, accediendo a la opción *Downloads* se encontrarán versiones estables pasadas de *Python*.



Una vez descargado el archivo .exe, se pasará a la instalación.

Filename, size	File type	Python version	Upload date	Hashes
JSBSim-1.1.8-588-cp36-cp36m-macosx_10_14_x86_64.whl (1.3 MB)	Wheel	cp36	Jul 24, 2021	View
JSBSim-1.1.8-588-cp36-cp36m-win_amd64.whl (793.5 kB)	Wheel	cp36	Jul 24, 2021	View
JSBSim-1.1.8-588-cp37-cp37m-macosx_10_14_x86_64.whl (1.3 MB)	Wheel	cp37	Jul 24, 2021	View
JSBSim-1.1.8-588-cp37-cp37m-win_amd64.whl (798.6 kB)	Wheel	cp37	Jul 24, 2021	View
JSBSim-1.1.8-588-cp38-cp38-macosx_10_14_x86_64.whl (1.3 MB)	Wheel	cp38	Jul 24, 2021	View
JSBSim-1.1.8-588-cp38-cp38-win_amd64.whl (800.7 kB)	Wheel	cp38	Jul 24, 2021	View
JSBSim-1.1.8-588-cp39-cp39-macosx_10_14_x86_64.whl (1.3 MB)	Wheel	cp39	Jul 24, 2021	View
JSBSim-1.1.8-588-cp39-cp39-win_amd64.whl (793.8 kB)	Wheel	cp39	Jul 24, 2021	View



2. En el caso de utilizar una versión de Python inferior a 3.4 o 2.7.9, será necesario descargar PIP. PIP es un acrónimo de “PIP Installs Packages” o “Preferred Installer Program” y es un comando que sirve para instalar, reinstalar, actualizar o desinstalar. Se deberá chequear si se ha instalado correctamente mediante el comando siguiente.

```
py -m pip --version
```

```
PS C:\Windows\system32> py -m pip --version
pip 21.2.4 from C:\Users\miguel\AppData\Local\Packages\PythonSoftwareFoundation.Python.3.9_qbz5n2kfra8p0\LocalCache\loca
```

Si no se tiene instalado, se deberá descargar e instalar mediante el comando siguiente:

```
py -m ensurepip --default-pip
```

Gracias a este comando se podrán instalar todas las librerías que se han usado para este Trabajo fin de Master. Gracias a este comando, una vez se comande instalar una librería en específico, se descargará aquella que sea compatible con la versión de Python que se está utilizando por defecto.

3. Descargar todas las librerías necesarias para ejecutar los códigos implementados en este Trabajo fin de Master, ya sea desde una consola *Python* o desde Windows powershell añadiendo al principio del código *Python -m*:

- a. *JSBSim*, mediante el comando “*Python -m pip install JSBSim*”

```
PS C:\Windows\system32> python -m pip install JSBSim
Collecting JSBSim
  Downloading JSBSim-1.1.8-588-cp39-cp39-win_amd64.whl (793 kB)
    |-----| 793 kB 939 kB/s
Collecting numpy
  Downloading numpy-1.21.2-cp39-cp39-win_amd64.whl (14.0 MB)
    |-----| 14.0 MB 726 kB/s
Installing collected packages: numpy, JSBSim
  WARNING: The script f2py.exe is installed in 'C:\Users\miguel\AppData\Local\Packages\PythonSoftwareFoundation.Python.3
  .9_qbz5n2kfra8p0\LocalCache\local-packages\Python39\Scripts' which is not on PATH.
  Consider adding this directory to PATH or, if you prefer to suppress this warning, use --no-warn-script-location.
Successfully installed JSBSim-1.1.8 numpy-1.21.2
```

- b. *Pygame*, mediante el comando “*Python -m pip install pygame*”
- c. *Kivy*, mediante el comando “*Python -m pip install Kivy*”

A veces es necesario instalar otras librerías para hacer que *Kivy* funcione, sobretodo para raspberry Pi OS, donde se recomienda seguir la guía de esta página web <https://kivy.org/doc/stable/installation/installation-rpi.html> o ejecutar desde la consola Linux la siguiente comanda:

“*sudo apt install libsdl2-dev libsdl2-image-dev libsdl2-mixer-dev libsdl2-ttf-dev*”

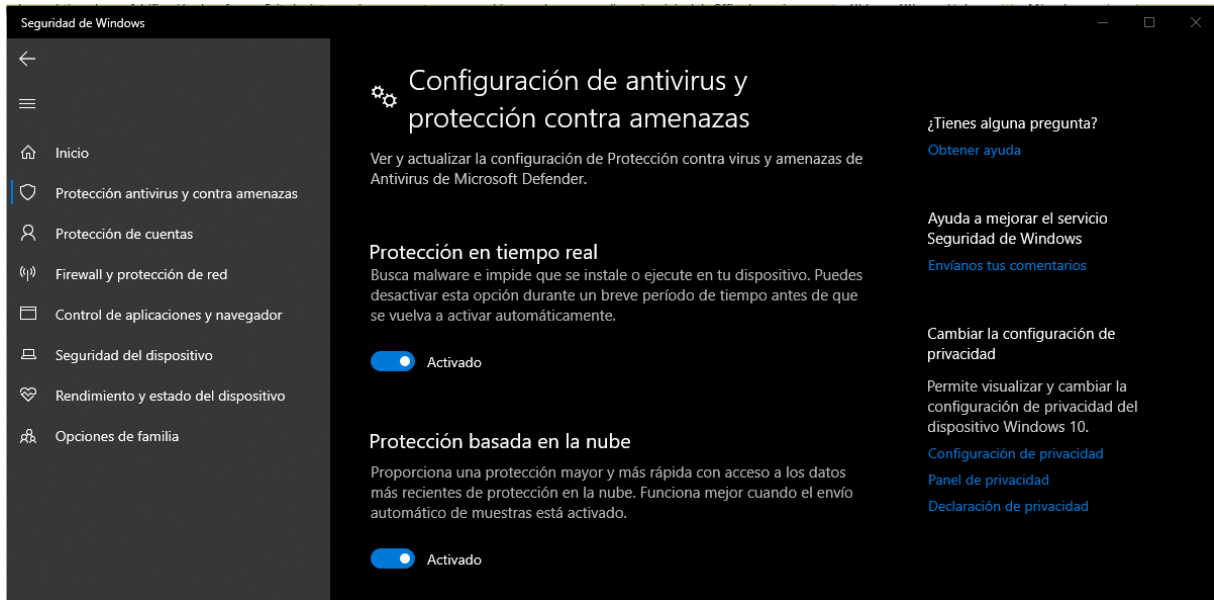
- d. *Socket*, mediante el comando “*Python -m pip install sockets*”

- c. Instalar Flightgear

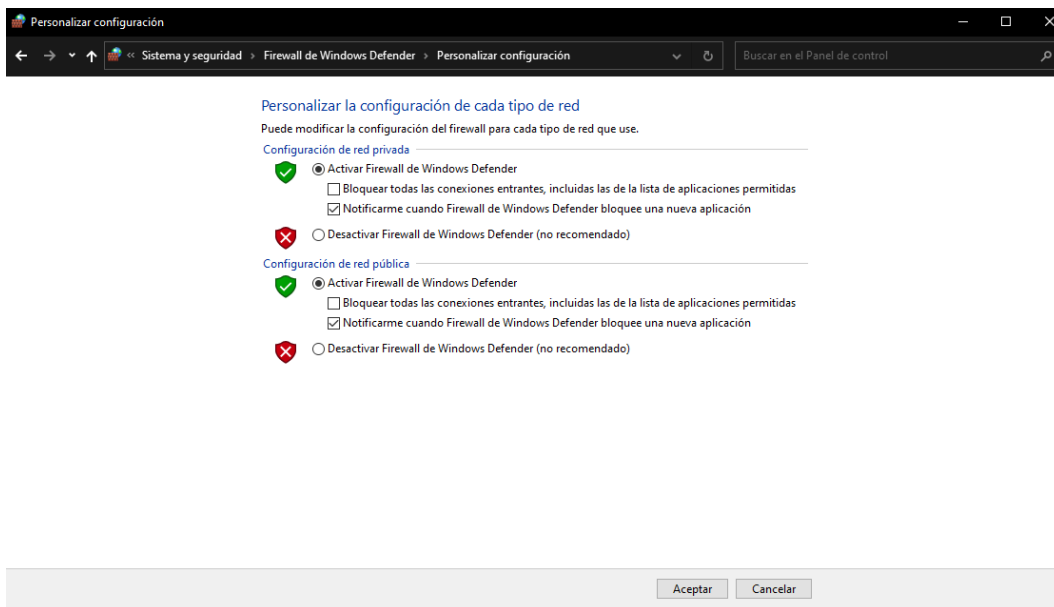
The screenshot shows the website <https://www.flightgear.org/download/>. The page features a blue header with the text "FLIGHTGEAR FLIGHT SIMULATOR" and the tagline "sophisticated, professional, open-source". Below the header is a navigation menu with links: Home, About, Download, Visit Store, Flying, Posts, Developers, and Legacy Site. The main content area is titled "DOWNLOAD CENTRAL" and contains the following text: "Download FlightGear 2020.3 – the latest stable, supported release – for free." Below this, there is a list of download links:

- Download [FlightGear 2020.3 for Windows](#) (versions 7, 8, 10)
- Download [FlightGear 2020.3 for macOS](#)
- [AppImage binary release](#) for Linux x86 systems
- Download the sources other platforms on the [download for other platforms page](#).

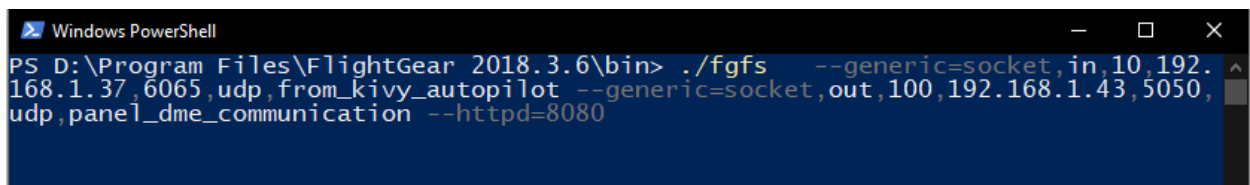
- a. Descargar Flightgear desde la pagina oficial, y ejecutar el archivo .exe.
<https://www.flightgear.org/download/>
- b. Para establecer conexión entre las múltiples herramientas desde la red WiFi, será necesario para Windows 10 desactivar la Protección en tiempo real y el Firewall de Windows, como se indica en las siguientes capturas. Pasos a seguir:
 - Seguridad de Windows =>Proteccion antivirus y contra amenazas => Administrar configuración => Protección en tiempo real desactivada



- Firewall de Windows Defender => Activar o desactivar el Firewall de Windows Defender => desactivar Firewall de Windows Defender



- c. A la hora de ejecutarlo, ir a la carpeta contenedora y abrir una consola (mayus + click derecho). Una vez abierta introducir las ordenes expuesta en la siguiente ilustración.



ANEXO A: CÓDIGO *PYTHON* JSBSIM

SIMULACIÓN TIEMPO REAL

```
# Importante para el controller
import pygame
# Importante para la simulacion dinamica
import jsbsim
# Tiempo de simulacion
import time
from controller import controller
# Inicializamos y creamos objeto joystick
pygame.init()
joysticks = []

#####
# clock = pygame.time.Clock()
keepPlaying = 1
for i in range(0, pygame.joystick.get_count()):
    # create an Joystick object in our list
    joysticks.append(pygame.joystick.Joystick(i))
    # initialize them all (-1 means loop forever)
    joysticks[-1].init()
    # print a statement telling what the name of the controller is
    print(f"Detected joystick {joysticks[-1].get_name()}")

#####

# Init jsbsim
fdm =jsbsim.FGFDMEExec('.', None)
fdm.load_model('c172p')
fdm.load_ic('reset00.xml', True)

#####

fdm['propulsion/set-running'] = -1
fdm['ic/alpha-deg'] = 3.2758
fdm['ic/theta-deg'] = 3.2787
fdm['ic/psi-true-deg'] = 307
fdm['gear/gear-cmd-norm'] = 0
fdm['ic/q-rad_sec'] = 0.0
fdm['ic/p-rad_sec'] = 0.0
fdm['ic/r-rad_sec'] = 0.0
fdm['ic/roc-fps'] = 0.00001
fdm['fcs/pitch-trim-cmd-norm'] = -0.2
fdm['fcs/roll-trim-cmd-norm'] = 0.0
fdm['fcs/yaw-trim-cmd-norm'] = 0.0
fdm['fcs/flap-cmd-norm'] = 0
fdm['fcs/elevator-cmd-norm'] = 0.0
fdm['fcs/roll-cmd-norm'] = 0.0
```



```

fdm['fcs/aileron-cmd-norm'] = 0.0
fdm['fcs/throttle-cmd-norm'[-1]] = 0.8
fdm['propulsion/refuel'] = 1
fdm['propulsion/total-fuel-lbs'] = 185

# fdm['fuel'] = 0.8
fdm['simulation/force-output'] = True
fdm.run_ic()
fdm.enable_output()

#####

fdm.print_simulation_configuration()
frame_duration = fdm.get_delta_t()
sleep_nseconds = (frame_duration if True else sleep_period) * 1E9
current_seconds = initial_seconds = time.time()
result = fdm.run()

#####

# print('***8 + 'Initial Conditions' + ***8 )
# print(fdm['ic/vc-kts'])
# print(fdm['ic/h-sl-ft'])
# print(fdm['ic/u-fps'])
# print(fdm['ic/p-rad_sec'])
# print(fdm['ic/q-rad_sec'])
# print(fdm['ic/r-rad_sec'])
# print(fdm['ic/roc-fps'])
# print(fdm['ic/alpha-deg'])
# print(fdm['ic/theta-deg'])
throttle_cmd = 1
elevator_cmd = 0
aileron_cmd = 0
rudder_cmd = 0

#####

while keepPlaying:
    # clock.tick(60)
    current_seconds = time.time()
    actual_elapsed_time = current_seconds - initial_seconds
    sim_lag_time = actual_elapsed_time - fdm.get_sim_time()
    iterations = int(sim_lag_time / frame_duration)

    for _ in range(int(sim_lag_time / frame_duration)):
        event = controller(joysticks)
        if event != None:
            if event.type == pygame.JOYAXISMOTION:
                if event.axis == 1:
                    # elevator_cmd = float(event.value)*0.5 +0.5
                    elevator_cmd = float(event.value)
                elif event.axis == 2:
                    throttle_cmd = float(event.value)*float(0.5) + 0.5
                    # throttle_cmd = float(event.value)*2 + 0.5

```

```

        # throttle_cmd = float(event.value)
    elif event.axis == 0:
        # aileron_cmd = float(event.value)*0.5 +0.5
        aileron_cmd = float(event.value)
    elif event.axis == 3:
        # roll_cmd = float(event.value)*0.5 +0.5
        rudder_cmd = float(event.value)

# throttle_cmd = 1
#
fdm['fcs/elevator-cmd-norm'] = elevator_cmd
fdm['fcs/rudder-cmd-norm'] = rudder_cmd
fdm['fcs/aileron-cmd-norm'] = aileron_cmd
fdm['fcs/throttle-cmd-norm'][-1] = throttle_cmd
# print(elevator_cmd)
print(throttle_cmd)
# fdm['fcs/mixture-cmd-norm'][-1] = throttle_cmd
# print(fdm.get_aircraft())
# fdm['simulation/dt'] = 2
result = fdm.run()
# print(int(result))
# fdm.output()
# print(result)
# print('*8 + 'Current Conditions' + '*8 )
# print('Current Alpha: %f'%(fdm['aero/alpha-deg']))
# print('Current Altitude: %f'%(fdm['position/h-sl-ft']))
# print('Current speed: %f'%(fdm['velocities/vc-kts']))
# print('Pitch degree: %f'%(fdm['attitude/pitch-rad']*57.2958))
# print('Roll degree: %f'%(fdm['attitude/roll-rad']*57.2958))
# print('Heading degree: %f'%(fdm['attitude/psi-rad']*57.2958))
# print('Engine 1 N2: %f'%(fdm['propulsion/engine[0]/n2']))
# print('Engine 2 N2: %f'%(fdm['propulsion/engine[1]/n2']))
# print('Engine 1 thrust-lbs: %f'%(fdm['propulsion/engine[0]/thrust-lbs']))
# print('Engine 2 thrust-lbs: %f'%(fdm['propulsion/engine[1]/thrust-lbs']))
current_seconds = time.time()
actual_elapsed_time = current_seconds - initial_seconds
# print(fdm.get_sim_time())

```

ANEXO B: CÓDIGO *PYTHON* PRUEBA PYGAME

```

import pygame
pygame.init()

def main():
    screen = pygame.display.set_mode((640, 480))
    pygame.display.set_caption("Joystick Testing / thrustmaster")

    background = pygame.Surface(screen.get_size())
    background = background.convert()
    background.fill((255, 255, 255))

    joysticks = []
    clock = pygame.time.Clock()
    keepPlaying = True

    # for al the connected joysticks
    for i in range(0, pygame.joystick.get_count()):
        # create an Joystick object in our list
        joysticks.append(pygame.joystick.Joystick(i))
        # initialize them all (-1 means loop forever)
        joysticks[-1].init()
        # print a statement telling what the name of the controller is
        print(f"Detected joystick {joysticks[-1].get_name()}")
        while keepPlaying:
            clock.tick(60)
            for event in pygame.event.get():
                if event.type == pygame.QUIT:
                    print("Received event 'Quit', exiting.")
                    keepPlaying = False

                elif event.type == pygame.KEYDOWN and event.key == pygame.K_ESCAPE:
                    print("Escape key pressed, exiting.")
                    keepPlaying = False
            # elif event.type == pygame.KEYDOWN:
            #     print(f"Keydown, {event.key}")
            # elif event.type == pygame.KEYUP:
            #     print(f"Keyup {event.key}")
            #elif event.type == pygame.MOUSEMOTION:
            #     print "Mouse movement detected."
            elif event.type == pygame.MOUSEBUTTONDOWN:
                print(f"Mouse button {event.button} down at {pygame.mouse.get_pos()}")
            elif event.type == pygame.MOUSEBUTTONUP:
                print(f"Mouse button {event.button} up at {pygame.mouse.get_pos()}")
            elif event.type == pygame.JOYAXISMOTION:
                print(f"Joystick {joysticks[event.joy].get_name()} axis {event.axis} motion.")
                print(float(event.value))
                print(event)

```

```
        # print(joysticks[event.joy])
elif event.type == pygame.JOYBUTTONDOWN:
    print(f"Joystick {joysticks[event.joy].get_name()} button {event.button} down.")
if event.button == 0:
    background.fill((255, 0, 0))
    print(event)
    # event.values
elif event.button == 1:
    background.fill((0, 0, 255))
    print(event)
    # print(float(event.value))
elif event.button == 3:
    background.fill((0, 124, 255))
    print(event)
    # print(float(event.value))
elif event.type == pygame.JOYBUTTONUP:
    print(f"Joystick {joysticks[event.joy].get_name()} button {event.button} up.")
if event.button == 0:
    background.fill((255, 255, 255))
    print(event)
    # print(int(event.button))
elif event.button == 1:
    background.fill((255, 255, 255))
    print(event)
    # print(int(event.button))
elif event.button == 3:
    background.fill((255, 255, 255))
    print(event)
    # print(int(event.button))
elif event.type == pygame.JOYHATMOTION:
    print(f"Joystick {joysticks[event.joy].get_name()} hat {event.hat} moved.")
    print(event)

screen.blit(background, (0, 0))
pygame.display.flip()

main()
pygame.quit()
```