UNIVERSITAT POLITÈCNICA DE VALÈNCIA

Escuela Técnica Superior de Ingeniería Aeroespacial y Diseño Industrial

Análisis y simulación de llegadas en trombón al Aeropuerto de Barcelona frente a llegadas tradicionales.

Trabajo Fin de Máster

Máster Universitario en Ingeniería Aeronáutica

AUTOR/A: Lidón González, Alejandro

Tutor/a: Vila Carbó, Juan Antonio

Cotutor/a: Yuste Pérez, Pedro

CURSO ACADÉMICO: 2022/2023

TRABAJO DE FIN DE MÁSTER

Análisis y simulación de llegadas en trombón al Aeropuerto de Barcelona frente a llegadas tradicionales

Autor: Lidón González, Alejandro
Tutor: Vila Carbó, Juan Antonio
Cotutor: Yuste Pérez, Pedro

Máster en ingeniería aeronáutica
Valencia, Julio de 2023

Resumen

En el presente proyecto se desarrolla una simulación de una aproximación al Aeropuerto de Barcelona-El Prat en el lenguaje de programación Java que aplicará de manera autónoma protocolos de control y secuenciación de tráfico y se desarrolla experimentos con ella. Se simularán una aproximación tradicional y una aproximación en trombón, para poder utilizar la herramienta desarrollada para extraer conclusiones sobre la eficacia y el uso de ambas.

En primer lugar, se declaran los objetivos del proyecto y los requerimientos que debe cumplir el código realizado.

Posteriormente, se explican los conceptos teóricos necesarios para comprender los elementos del espacio aéreo y control de tráfico aéreo necesarios para poder entender el contexto del proyecto. Además de los contenidos básicos de aeronavegación, se explican las operaciones de descenso continuo, la secuenciación, los tipos de aproximación, la capacidad de los aeropuertos y el Proyecto BRAIN que define la nueva aproximación en trombón al Aeropuerto de Barcelona.

Seguidamente, se explica la obtención de los datos de las cartas aeronáuticas y el diseño del código desarrollado en Java, profundizando en los componentes del código, la estructura del programa y la interfaz gráfica de usuario. Después se expone su implementación a nivel de código y el manual de usuario.

Por último, se realizan los experimentos con la simulación realizada, de los que se analizan los resultados para extraer conclusiones sobre las aproximaciones.

Resum

En aquest projecte es desenvolupa una simulació d'una aproximació a l'Aeroport de Barcelona-El Prat en el llenguatge de programació Java que aplicarà de manera autònoma protocols de control i seqüenciació de trànsit i s'hi desenvolupa experiments. Se simularan una aproximació tradicional i una aproximació a trombó, per poder utilitzar l'eina desenvolupada per extreure conclusions sobre l'eficàcia i l'ús de totes dues.

En primer lloc, es declaren els objectius del projecte i els requeriments que ha de complir el codi realitzat.

Posteriorment, s'expliquen els conceptes teòrics necessaris per comprendre els elements de l'espai aeri i el control de trànsit aeri necessaris per poder entendre el context del projecte. A més dels continguts bàsics d'aeronavegació, s'expliquen les operacions de descens continu, la seqüenciació, els tipus d'aproximació, la capacitat dels aeroports i el projecte BRAIN que defineix la nova aproximació al trombó a l'aeroport de Barcelona.

Seguidament, s'explica l'obtenció de les dades de les cartes aeronàutiques i el disseny del codi desenvolupat a Java, aprofundint els components del codi, l'estructura del programa i la interfície gràfica d'usuari. Després s'exposa la implementació a nivell de codi i el manual d'usuari.

Finalment, es fan els experiments amb la simulació realitzada, dels quals s'analitzen els resultats per extreure'n conclusions sobre les aproximacions.

Abstract

In the present project, a simulation of an approach to the Barcelona-El Prat Airport is developed in the Java programming language that will autonomously apply traffic control and sequencing protocols and is used to experiment. A traditional approach and a trombone approach will be simulated to be able to use the developed tool to draw conclusions about the effectiveness and use of both.

Firstly, the objectives of the project and the requirements that the created code must meet are declared.

Subsequently, the theoretical concepts necessary to understand the elements of airspace and air traffic control necessary to understand the context of the project are explained. In addition to the basic air navigation concepts, continuous descent operations, sequencing, approach types, airport capacity and the BRAIN Project that defines the new trombone approach to Barcelona Airport are explained.

Next, the obtaining of data from aeronautical charts and the design of the code developed in Java are explained, delving into the components of the code, the structure of the program and the graphical user interface. Afterwards, its implementation at the code level and the user manual are explained.

Finally, the experiments are carried out with the developped simulation, from which the results are analyzed to draw conclusions about the approximations.

Agradecimientos

Me gustaría agradecer a mi tutor, Juan Antonio Vila Carbó, por proponer la idea de este proyecto y ayudarme durante todo su desarrollo.

Agradezco también a mi cotutor Pedro Yuste Pérez por buscar y propocionarme los documentos que he necesitado para realizar la sección teórica del proyecto.

Agradezco por último a mi padre y a mi madre por su apoyo y por haber hecho posible que persiga este máster y anteriormente el grado.

ÍNDICE

Resumen I
Resum II
Abstract III
Agradecimientos IV
ÍNDICE DE FIGURAS VIII
ÍNDICE DE TABLAS X

1. Introducción **1**
1.1. Objetivos 1
1.2. Requerimientos 2
1.2.1. Requerimientos de software 2
1.2.2. Requerimientos de usuario 2

2. Estado del arte **3**
2.1. Cartas aeronáuticas 3
2.2. PBN 4
2.3. NAVAIDs 5
2.4. Navegación por GNSS 7
2.5. Aproximación 8
2.6. Fijos o waypoints 9
2.7. Circuitos de espera 10
2.8. Control de Tráfico Aéreo 11
2.9. Secuenciación 12
2.9.1. Operaciones de descenso continuo 12
2.9.2. Métodos de diseño de CDO 13
2.9.3. Métodos de secuenciación 14
2.9.4. Vectorización 15
2.10. Tipos de aproximaciones 16
2.10.1. Aproximación directa 16
2.10.2. Patrones de apilamiento 17
2.10.3. Point merge 17
2.10.4. Aproximación Fan 18
2.10.5. Aproximación en trombón 19
2.11. Capacidad de un aeropuerto 20
2.12. Aeropuerto de Barcelona-El Prat 21
2.12.1. Proyecto BRAIN 22
2.13. Sistemas de coordenadas 23
2.13.1. Coordenadas geográficas 23
2.13.2. Coordenadas ENU 25

3. Herramientas **26**
3.1. Programación orientada a objetos 26
3.1.1. Clases y objetos 26

3.1.2. Pilares de la programación orientada a objetos 27
3.2. Java 28
3.2.1. Introducción a Java 29
3.2.2. Concurrencia: threads 29
3.2.3. Colecciones: List y Map 31
3.2.4. Sockets 32
3.2.5. GUI 33
3.3. MATLAB 34
3.3.1. Función lla2enu 35

4. Diseño 36
4.1. Inserción de las coordenadas en el código 36
4.1.1. Procedimiento tradicional VOR 36
4.1.2. Procedimiento de transición a la aproximación final 39
4.2. Capacidad del aeropuerto 40
4.3. Arquitectura del software 41
4.3.1. Paquete navigation 42
4.3.2. Paquete control 47
4.3.3. Paquete GUI 55

5. Implementación 58
5.1. Paquete navigation 58
5.1.1. navigation.Traffic 58
5.1.2. navigation.Waypoint 61
5.1.3. navigation.Route 63
5.1.4. navigation.Position 64
5.1.5. navigation.Fly 65
5.1.6. navigation.Flyby 66
5.1.7. navigation.TraditionalArrival 67
5.1.8. navigation.NewArrival 68
5.2. Paquete control 69
5.2.1. control.TrafficControl 69
5.2.2. control.TrafficControlNew 71
5.2.3. control.TrafficControlNew2 73
5.2.4. control.TrafficGenerator 75
5.2.5. control.TrafficGeneratorNew 77
5.3. Paquete GUI 78
5.3.1. GUI.WaypointPlot 78
5.3.2. GUI.TrafficPlot 80
5.3.3. GUI.TrackPlot 82
5.3.4. GUI.RadarPanel 83
5.3.5. GUI.RadarPanelNew 85
5.3.6. GUI.RadarPanelNew2 87
5.3.7. GUI.Frame, GUI.FrameNew y GUI.FrameNew2 89

6. Manual de usuario 90

7. Experimentos **94**
7.1. Caracterización de la carga 94
7.2. Experimento 1: separación de 1.5 minutos 95
7.2.1. Resultados 95
7.2.2. Análisis de resultados 96
7.3. Experimento 2: separación de 2 minutos 96
7.3.1. Resultados 96
7.3.2. Análisis de resultados 97
7.4. Experimento 3: separación de 1.2 minutos 98
7.4.1. Resultados 98
7.4.2. Análisis de resultados 99
7.5. Experimento 4: separación de 2.5 minutos 100
7.5.1. Resultados 100
7.5.2. Análisis de resultados 101
7.6. Experimento 5: separación de 3.5 minutos 101
7.6.1. Resultados 101
7.6.2. Análisis de resultados 102

8. Presupuesto **103**
8.1. Coste del material 103
8.2. Coste de personal 103
8.3. Costes indirectos 104
8.4. Presupuesto total 104

9. Objetivos de Desarrollo Sostenible **105**

10. Conclusiones **106**

ÍNDICE DE FIGURAS

2.1. Estación DME 6
2.2. Estación VOR 6
2.3. Sistema ILS 6
2.4. Arquitectura del GBAS (FAA) 8
2.5. Fases de la aproximación (Obtenido de apuntes) 9
2.6. Giros con fly-by y fly-over (Obtenido de apuntes) 10
2.7. Simbología de fijos y waypoints en la cartografía aeronáutica [5] 10
2.8. Ejemplo de circuito de espera con viraje a la derecha [6] 11
2.9. Ejemplo de diseño de trayectoria cerrada [8] 13
2.10. Ejemplo de diseño de trayectoria abierta vectorizada [8] 14
2.11. Ejemplo de diseño de trayectoria abierta vectorizada parcialmente [8] . . . 14
2.12. Maniobra de *point merge* [8] 15
2.13. Maniobra de *path stretching* [8] 16
2.14. Esquema de una aproximación directa [9] 16
2.15. Esquema de un holding stack vertical [10] 17
2.16. Esquema de una aproximación con stacking [9] 17
2.17. Esquema de una aproximación con point merge [9] 18
2.18. Esquema de una aproximación fan [9] 18
2.19. Esquema de una aproximación en trombón [9] 19
2.20. Esquema de una aproximación en trombón con pistas paralelas [9] 20
2.21. Tramo en trombón del Aeropuerto de Barcelona-El Prat [11] 20
2.22. Vista aérea del Aeropuerto de Barcelona-El Prat 22
2.23. Esquema del proyecto BRAIN [16] 23
2.24. Representación de la latitud [18] 24
2.25. Representación de la longitud [18] 24
2.26. Representación de la altitud [18] 24
2.27. Representación de las coordenadas ENU [18] 25

3.1. Ejemplo de herencia en Java [20] 28
3.2. Representación de threads concurrentes [20] 30
3.3. Ejemplo de comunicación entre threads [20] 30
3.4. Tipos de conexión con sockets [20] 33
3.5. Modelo MVC [20] 33
3.6. Coordenadas en el JPanel [20] 34

4.1. Carta IAC VOR RWY 24R [21] 37
4.2. Código en MATLAB utilizado para convertir de LLA a ENU 38
4.3. Carta IAC TRAN RWY 24R [11] 39
4.4. Paquetes del proyecto 41
4.5. Estructura del paquete navigation 42
4.6. Aspecto de la función flyby en MATLAB 45
4.7. Aspecto de la función fly en MATLAB 46
4.8. Interfaz simplificada de los protocolos de control 50

4.9. Estructura del paquete GUI 55

6.1. Ejecutables del proyecto 90
6.2. Aspecto del panel de la simulación 91
6.3. Aspecto de la aproximación en trombón en funcionamiento en pantalla completa 92
6.4. Aspecto de la aproximación tradicional en funcionamiento en pantalla completa 92
6.5. Aspecto del cuadro de outputs 93

ÍNDICE DE TABLAS

4.1. Coordenadas de los waypoints de la carta IAC VOR RWY 24R [21] 38
4.2. Coordenadas de los waypoints de la carta IAC TRAN RWY 24R [11] . . . 40
4.3. Fragmento de la tabla de la capacidad aeroportuaria para el verano de 2023 [22] 41

7.1. Media de los resultados del experimento 1 para aproximación VOR tradicional 95
7.2. Media de los resultados del experimento 1 para aproximación en trombón con cola única 95
7.3. Media de los resultados del experimento 1 para aproximación en trombón con dos colas 95
7.4. Media de los resultados del experimento 2 para aproximación VOR tradicional 97
7.5. Media de los resultados del experimento 2 para aproximación en trombón con cola única 97
7.6. Media de los resultados del experimento 2 para aproximación en trombón con dos colas 97
7.7. Media de los resultados del experimento 3 para aproximación VOR tradicional 98
7.8. Media de los resultados del experimento 3 para aproximación en trombón con cola única 99
7.9. Media de los resultados del experimento 3 para aproximación en trombón con dos colas 99
7.10. Media de los resultados del experimento 4 para aproximación VOR tradicional 100
7.11. Media de los resultados del experimento 4 para aproximación en trombón con cola única 100
7.12. Media de los resultados del experimento 4 para aproximación en trombón con dos colas 100
7.13. Media de los resultados del experimento 5 para aproximación VOR tradicional 101
7.14. Media de los resultados del experimento 5 para aproximación en trombón con cola única 102
7.15. Media de los resultados del experimento 5 para aproximación en trombón con dos colas 102

8.1. Costes de material 103
8.2. Costes de personal 104
8.3. Costes de material 104

Capítulo 1

Introducción

1.1. Objetivos

El objetivo general de este proyecto es la comparación del funcionamiento de una aproximación clásica a un aeropuerto contra una aproximación en trombón al mismo, de manera que se puedan observar las diferencias en diferentes situaciones de tráfico, con diferente media de tiempo entre llegadas. Para realizar esta comparación se utiliza una de las llegadas al Aeropuerto de Barcelona - El Prat, del que hay disponibles cartas con los procedimientos de aproximación clásicos y los modernos, con aproximación en trombón. Con esta simulación se pretenden realizar experimentos para obtener conclusiones sobre los procedimientos. de esta manera se tienen dos objetivos principales:

- Construir un **simulador** en Java con ambos esquemas de aproximación (VOR y trombón) basada en los datos de una aproximación real a Barcelona. Esta simulación deberá visualizar los aviones mientras van llegando y se acercan al aeropuerto, y deberá permitir implementar esquemas de control que decidan los criterios en base a los que se da permiso a los aviones que llegan a realizar la aproximación, además del orden de preferencia para iniciarla.

- Analizar el rendimiento de los diferentes esquemas mediante **experimentos** que utilicen la herramienta desarrollada. De esta manera los resultados dependerán de la calidad de la aplicación desarrollada. Si se plantease utilizar una aplicación de terceros no existiría este problema, pero al tener que desarrollar la aplicación los resultados obtenidos tienen más significado, ya que se entienden los procesos que llevan a ellos.

Con este proyecto se intentan conseguir además los siguientes objetivos secundarios:

- Mejorar los conocimientos previos de programación basada en objetos con Java utilizando el entorno NetBeans IDE, para continuar con el desarrollo realizado en el segundo año de Máster.

- Crear una representación gráfica simple que sea sencilla de interpretar en la que se muestren los aviones en movimiento de forma lo más realista posible.

- Explicar la capacidad de un aeropuerto y calcular cuánto de esta se está utilizando según el ratio de llegadas de cada simulación realizada.

- Explicar los conceptos de secuenciación de tráfico que utiliza ATC en las aproximaciones, especialmente la vectorización.

- Encontrar el mejor sistema de control posible para decidir cuándo dar permiso a los aviones y en qué orden, explicando por qué se considera que el sistema escogido es el más adecuado, y cómo se podría cambiar por otro en caso de que se quiera mejorar.
- Simular de manera aleatoria en qué punto de entrada llega cada avión y que su separación temporal siga una distribución estadística, para que la simulación no sea rígida y ofrezca resultados diferentes y más realistas en cada ejecución.

1.2. Requerimientos

En este apartado se nombrarán los requerimientos que le serán exigidos al código desarrollado. Estos requerimientos se puede clasificar en requerimientos de software y requerimientos de interacción con el usuario.

1.2.1. Requerimientos de software

Los únicos requerimientos de software que se le exigirá al programa de simulación son los requeridos para cumplir con los objetivos del proyecto, que se pueden resumir en utilizar Netbeans como entorno de desarrollo integrado de Java, ya que se pretende expandir los conocimientos desarrollados en el Máster con Java y con este entorno en particular.

1.2.2. Requerimientos de usuario

La simulación que se pretende conseguir conceptualmente está muy automatizada, debido a que se quiere conseguir que no se pueda prever el orden de llegada de los aviones en un tiempo exacto y en qué waypoint entrará cada avión a la aproximación. Por tanto el usuario solo podrá tener control sobre los aspectos relacionados con el funcionamiento de la simulación, como la media estadística de la separación temporal de los aviones o la velocidad de simulación (cuánto es acelerada la simulación con respecto a la realidad).

Además, el software desarrollado estará basado en simplificaciones aceptables de la realidad y en coordenadas introducidas manualmente en el código, por lo que no tendrá la capacidad de elegir un diferente aeropuerto para la simulación, o un procedimiento de aproximación diferente para el mismo aeropuerto, ya que esto se sale de los objetivos establecidos para el proyecto.

Capítulo 2

Estado del arte

En este capítulo se expondrán los diferentes conceptos teóricos que conciernen a los temas tratados en el proyecto, explicando los procedimientos y tecnologías que forman el trasfondo del proyecto, y que son necesarios para poder entender lo que se trata de conseguir con la simulación.

2.1. Cartas aeronáuticas

Las cartas aeronáuticas son una fuente completa oficial de datos de navegación que ofrece seguridad a la hora de completar las operaciones áreas. Se deben caracterizar por su sencillez, condensación y coordinación. Son muy utilizadas en numerosos campos, como control de tráfico aéreo, planificación de rutas y aeropuertos y en navegación.

Se dividen en 3 grupos en función de su grado de obligatoriedad [1]:

- **Cartas obligatorias**. Son las siguientes: el Plano de obstáculos de aeródromo Tipo A, la Carta topográfica para aproximaciones de precisión, la Carta de navegación en ruta, la Carta de aproximación por instrumentos, el Plano de aeródromo/ helipuerto y la Carta aeronáutica mundial.

- **Cartas opcionales**. Son producidas si su disposición contribuye a una mayor seguridad, regularidad y eficiencia de las operaciones. Son las siguientes: el Plano de obstáculos de aeródromo Tipo B, el Plano de aeródromo para movimientos en tierra, el Plano de estacionamiento y atraque de aeronaves, las Cartas aeronáuticas 1:500 000, la Carta de navegación aeronáutica y la Carta de posición.

- **Cartas condicionalmente necesarias**. Solo es necesario producirlas si se dan condiciones o circunstancias determinadas. Son las siguientes: el Plano de obstáculos de aeródromo Tipo C, la Carta de área, la Carta de salida normalizada - vuelo por instrumentos, la Carta de llegada normalizada - vuelo por instrumentos y la Carta de aproximación visual.

Además las cartas también se clasifican en 4 grupos:

- **Grupo 1: Cartas destinadas exclusivamente a La planificación**. Son los Plano de obstáculos de aeródromo Tipo A, B y C y la Carta topográfica para aproximaciones de precisión.

- **Grupo 2: Cartas destinadas a las fases del vuelo comprendidas entre el despegue y el aterrizaje**. Son la Carta de navegación e ruta, la Carta de

área, la Carta de salida normalizada - vuelo por instrumentos (SID), la Carta de Llegada normalizada - vuelo por instrumentos (STAR), la Carta de aproximación por instrumentos y la Carta de aproximación visual.

- **Grupo 3: Cartas destinadas a los, movimientos de las aeronaves en la superficie del aeródromo**. Son el Plano de aeródromo/helipuerto, Plano de aeródromo para movimientos en tierra y el Plano de estacionamiento y atraque de aeronaves.
- **Grupo 4: Cartas destinadas a la navegación aérea visual planificación y determinación de la posición**. Son la Carta aeronáutica mundial, la Carta aeronáutica, la Carta de navegación aeronáutica y la Carta de posición.

Para este proyecto la carta aeronáutica que se utiliza como base es una carta de aproximación por instrumentos (IAC), una carta obligatoria y del Grupo 2. Esta carta contiene el trayecto que comprende el tramo desde el IAF al aeropuerto, es decir, todos los tramos de aproxiamción, inicial, intermedia y final.

2.2. PBN

El PBN (Navegación Basada en Prestaciones) es un concepto de navegación que promovió la OACI ante el aumento de la congestión del espacio aéreo por el aumento del tráfico. El Volumen I del Anexo 10 del Convenio sobre Aviación Civil Internacional [2]: "*Requisitos para la navegación de área basada en la performance que se aplican a las aeronaves que realizan operaciones en una ruta ATS, en un procedimiento de aproximación por instrumentos o en un espacio aéreo designado.* ".

Su función es mejorar la eficiencia de las operaciones. Como indica su nombre, establece una navegación basada en prestaciones que no requiere sensores concretos, sino una serie de requerimientos RNP (Required Navigation Performance) que están definidos en términos de precisión, integridad, disposición y continuidad. De esta manera con PBN no se requiere un sensor específico, sino uno que cumpla con los requisitos RNP, con lo que se puede hacer navegación con NAVAIDs o con GNSS cumpliendo con los estándares PBN. Los requerimientos se definen de la siguiente manera [3]:

- **Precisión.** Para un tiempo dado, es el grado de similitud entre la posición medida de un objeto y la real.
- **Disponibilidad.** Porcentaje de tiempos en los que el sistema de navegación está en disponible para el uso por el navegador.
- **Continuidad.** Habilidad del sistema para realizar su función sin interrupción durante la operación.
- **Integridad.** Grado de confianza que se puede depositar en la precisión de la información que proporciona un sistema de navegación.

El PBN promueve el cambio en la metodología de navegación de la navegación convencional (con rutas y normas restrictivas) a la navegación de área (RNAV). RNAV permite trazar trayectorias en las zonas de cobertura de las ayudas disponibles según las capacidades de las distintas aeronaves. RNAV está definida por el Anexo 10 de la OACI [2]: "*Método de navegación que permite la operación de aeronaves en cualquier trayectoria de vuelo deseada, dentro de la cobertura de las ayudas para la navegación basadas en tierra o en el espacio, o dentro de los límites de capacidad de las ayudas autónomas, o una combinación de ambas*"

RNP es una evolución de la navegación de área, que utiliza una función e vigilancia y alerta de conflictos, lo que permite volar trayectorias más complejas, y por tanto utilizar mejor el espacio aéreo.

2.3. NAVAIDs

Las ayudas de navegación (NAVAIDs) son equipos terrestres que emiten cualquier tipo de señal que ayude a los pilotos en navegación. En el caso de señales radioeléctricas se habla de radioayudas. Los tipos de radioayudas más importantes son los siguientes:

- **DME (Equipo Medidor de Distancia).** Mide la distancia entre una estación terrestre y el avión. Utiliza el retardo en la propagación de las señales de radio que envía para calcular esta distancia. En la figura 2.1 se muestra una estación DME.

- **VOR (Radiofaro Omnidireccional VHF).** Estaciones con antenas que emiten una señal VHF en todas las direcciones, que reciben los aviones que tengan el equipo adecuado y la frecuencia de la estación sincronizada. Permite a los aviones seguir un rumbo determinado. En la figura 2.2 se muestra una estación VOR.

- **ILS (Sistema de Aterrizaje Instrumental).** Sistema formado por una serie de antenas que sirve de ayuda a los aviones en el aterrizaje, haciendo posible realizar un aterrizaje de precisión. Proporciona guiado vertical (a través de una senda de planeo) y horizontal (mediante el localizador, que proporciona la posición respecto al eje de la pista). En la figura 2.3 se muestra parte de un sistema ILS.

Figura 2.1: Estación DME

Figura 2.2: Estación VOR

Figura 2.3: Sistema ILS

2.4. Navegación por GNSS

El GNSS es un sistema que emplea modelos matemáticos para calcular el posicionamiento, utilizando como referencia un geoide de referencia como el WGS84 (utilizado por EEUU) o el ETRS89 (utilizado por Europa).

Los algoritmos utilizados se basan en trilateración esférica, que consiste en lo siguiente: Con 3 satélites se trazan esferas con radio ρ del satélite al receptor. Con la intersección de las 3 esferas se tiene u sistema de 3 ecuaciones con 3 incógnitas. Los errores causados por la desviación de los relojes hacen necesario utilizar un 4º satélite, ya que la distancia obtenida sin este no es la verdadera. Además, si los satélites están demasiado cerca se tiene una mala geometría, lo que causa errores. Por tanto es necesario tener una geometría favorable de los satélites.

Existen múltiples sistemas GNSS, siendo el GPS (Global Positioning System) de Estados Unidos el más conocido. Pero también están en uso otros como Galileo (europeo y el único de ámbito civil), GLONASS (ruso) y BeiDou (chino).

El GNSS no ofrece por sí mismo las suficientes prestaciones como para ser utilizado en aeronavegación por sí mismo. Son necesarios los sistemas de aumentación, que permiten alcanzar los requisitos establecidos por la OACI.

El DGNSS (Differential GNSS) es una técnica de posicionamiento con pseudodistancias. Es utilizado por varios sistemas de aumentación, como el SBAS y el GBAAS [4].

Los sistemas de aumentación más importantes son:

- **ABAS (Sistema de Aumentación Basado en la Aeronave).** La propia aeronave realiza la aumentación. Se utiliza en ruta principalmente, también en aproximaciones de no precisión.
- **SBAS (Sistema de Aumentación Basado en Satélites).** Se utiliza una red de estaciones terrestres y estaciones maestras que transmiten mensajes a satélites geoestacionarios, que a su vez transmiten los mensajes SBAS.
- **GBAS (Sistema de Aumentación Basado en Tierra).** Emplea estaciones terrestres para comparar su posición real con la estimada y obtener el error de posicionamiento mediante correciones diferenciales. Se puede observar su arquitectura en la figura 2.4.

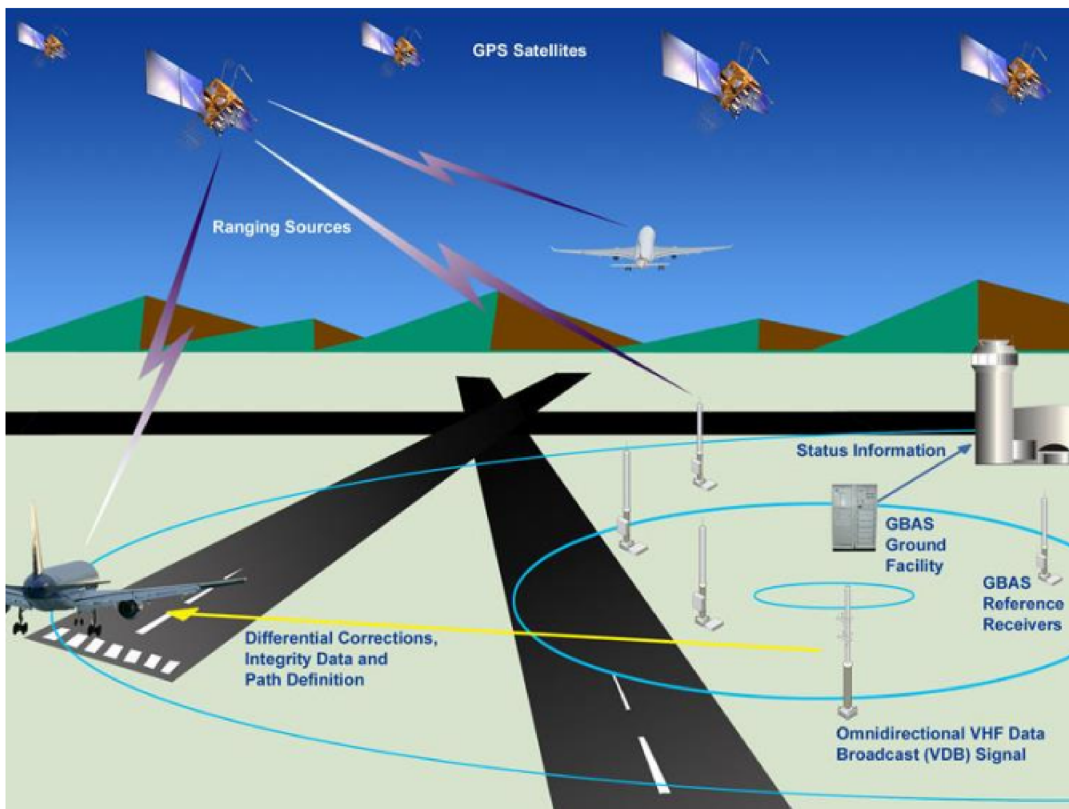

Figura 2.4: Arquitectura del GBAS (FAA)

2.5. Aproximación

La aproximación es la fase final del trayecto de una aeronave. La aproximación instrumental es el conjunto de maniobras que se realizan desde el IAF (Initial Approach Fix) hasta la pista de aterrizaje, mediante el uso de los instrumentos de abordo, con protección a los obstáculos obtenida gracias a los márgenes de franqueamiento. Se fivide en 4 fases:

- **Aproximación inicial.** Trayecto entre el IAF y el IF (Intermediate Fix).
- **Aproximación intermedia.** Trayecto entre el IF y el FAF (Final Approach Fix).
- **Aproximación final.** Trayecto entre el FAF y la pista.
- **Aproximación frustrada.** En caso de que no se pueda aterrizar con seguridad se realiza, entre el MAPt (Missed Approach Point) y el circuito de espera. Puede ser por cuestiones de visibilidad, por llevar una mala trayetoria, por estar en una posición incorrecta, etc. Cada procedimiento de aproximación debe tener al menos un procedmiento de aproximación frustrada.

En la figura 2.5 se muestran las fases gráficamente con su franqueamiento de obstáculos.

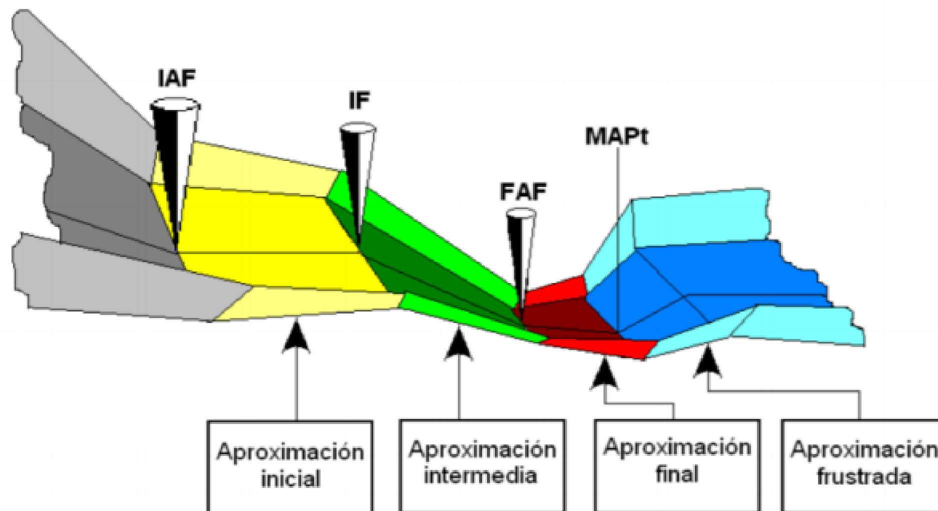

Figura 2.5: Fases de la aproximación (Obtenido de apuntes)

2.6. Fijos o waypoints

Los fijos son los puntos de los que se componen las rutas de los aviones. Marcan el recorrido que debe seguir el avión. En navegación tradicional los fijos se definen mediante instalaciones, como sobre la vertical de un VOR (Radiofaro Omnidireccional VHF) o combinación de radiales de un VOR y arcos de DME (Equipo de Medidor de Distancia). Estos métodos tienen una cierta área de tolerancia.

Para navegación PBN, sin embargo, esta tolerancia no es relevante, ya que los puntos se definen por coordenadas. Se llama entonces a los puntos de referencia waypoints.

Tanto waypoints como fijos pueden ser volados como fly-over o fly-by. El fly-over o sobrevuelo requiere que el avión pase por el waypoint o fijo obligatoriamente, mientras que el fly-by permite al piloto girar antes de llegar. En la figura 2.6 se puede observar la diferencia gráficamente.

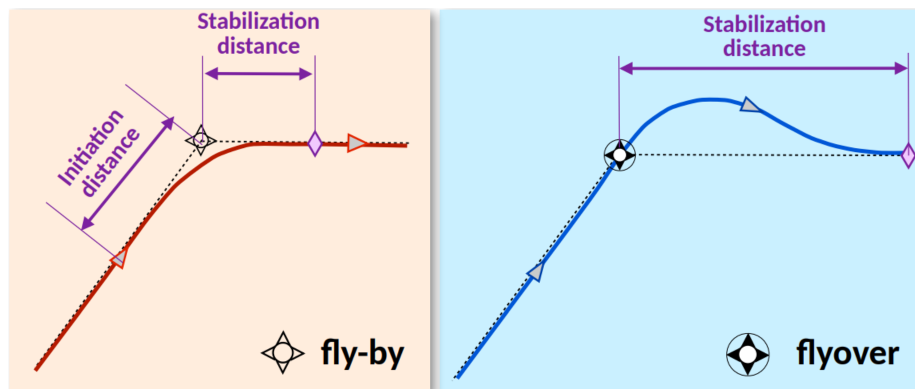

Figura 2.6: Giros con fly-by y fly-over (Obtenido de apuntes)

La simbología que se utiliza en las cartas aéronauticas se muestra en la figura 2.7. Los triángulos representan a los fijos, mientras que los círculos con puntas representan a los waypoints. A su vez, los que están oscurecidos representan fly-overs y los que son completamente blancos representan fly-bys.

Punto de notificación obligatorio Compulsory reporting point	▲
Punto de notificación facultativo On request reporting point	△
Punto de recorrido de sobrevuelo Flyover way-point	
Punto de recorrido de paso Fly-by way-point	

Figura 2.7: Simbología de fijos y waypoints en la cartografía aeronáutica [5]

2.7. Circuitos de espera

El Documento 8168 de OACI define el procedimiento de espera como: "*Maniobra predeterminada que mantiene a la aeronave dentro de un espacio aéreo especificado, mientras espera una autorización posterior*" [6]. Los circuitos de espera se colocan para permitir que las aeronaves mantengan la separación requerida, haciendo que en caso de que haya peligro de que no se cumpla la separación que requiere la capacidad del aeropuerto se pueda hacer esperar a las aeronaves un número de vueltas al circuito o hasta que reciban permiso de continuar.

Es evidente que aun si dos aviones cumplen la separación de seguridad requerida en ruta, no pueden aterrizar muy cerca el uno del otro, ya que tiene que dar tiempo a que el primer avión salga de la pista antes de que el siguiente pueda comenzar el aterrizaje. Es por esto que se colocan circuitos de espera en los IAC, para poder controlar la separación

de las aeronaves que confluyen hacia el aeropuerto desde las diferentes rutas de llegada.

También son necesarios los circuitos de espera para que los aviones que ejecutan aproximaciones frustradas los circulen hasta el nuevo intento de aproximación.

La entrada a un circuito de espera se realiza según el rumbo respecto a los tres sectores de entrada. Una vez la aeronave alcanza el punto de referencia de la espera, gira para seguir el rumbo de alejamiento. El tiempo que se sigue este rumbo suele indicarse mediante un tiempo cronometrado o una distancia. Después de alcanzar el nuevo punto de referencia según la distancia o el tiempo, la aeronave vuelve a girar y sigue el rumbo de acercamiento, durante el cual vuelve a alcanzar el punto de referencia inicial y ha completad una vuelta al circuito. El número de vueltas que sea necesario dar dependerá de las circunstancias de tráfico. En la figura 2.8 se muestra la estructura simplificada de un circuito de espera.

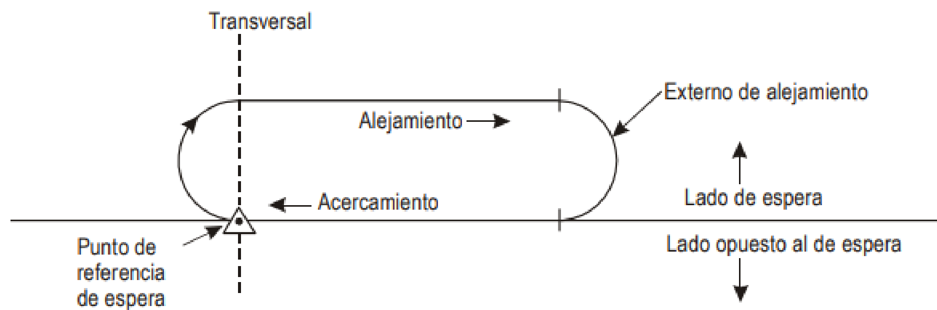

Figura 2.8: Ejemplo de circuito de espera con viraje a la derecha [6]

2.8. Control de Tráfico Aéreo

El Control de Tráfico Aéreo (ATC), es un servicio que se encarga de proporcionar seguridad a los vuelos de aeronaves, indicando separación y dando permisos para proceder a los aviones en espera, además de dar sugerencias u órdenes a los pilotos. Según la web de ENAIRE [7]: "*El servicio de control de tráfico aéreo (ATC) de ENAIRE es prestado por controladores aéreos, que aplican separaciones entre los aviones y emiten autorizaciones de control a petición de los pilotos, a propia iniciativa, en función de las condiciones del tránsito y del entorno*".

Hay 3 servicios ATC, como es establecido por ENAIRE:

- **Servicio de control de área** Control de los vuelos en un área de control terminal (TMA), en áreas de control o en aerovías. En España hay 5 centros de control (ACC), en Madrid, Barcelona, Sevilla, Palma de Mallorca y Gran Canaria.

- **Servicio de control de aproximación (APP)** Gestión de las entradas y salidas de los vuelos de una zona de control. Se controla el tráfico en la espera, aproximación,

despegue y aterrizaje, es decir, es las fases más cercanas al aeropuerto, y las más críticas por tanto.

- **Servicio de control de aeródromo (TWR)** Gestión de las aeronaves que hay en el aeropuerto y en las cercanías de este.

2.9. Secuenciación

2.9.1. Operaciones de descenso continuo

Las operaciones de descenso continuo (CDO) son una técnica de operación de aeronaves que permite la ejecución de un perfil de vuelo optimizado para la capacidad operacional del avión, con bajo empuje en el motor y una configuración de baja resistencia, para así reducir consumo de combustible y emisiones contaminantes, como viene definido en el Manual de Operaciones de descenso continuo (CD) [8]. Las CDO se apoyan en espacios aéreos y diseños de procedimiento adecuados, junto con ayuda y permisos de ATC.

El ángulo óptimo de descenso depende del tipo de avión y sus características, además de los factores ambientales. Un CDO puede ser volado con o sin soporte de una senda de planeo generada por ordenador y con o sin una trayectoria horizontal fija. Sin embargo, el máximo beneficio para un avión se consigue manteniendo el avión lo más alto posible hasta que llegue al punto óptimo de descenso.

Para poder realizar CDO se necesita que ATC provea una gestión segura y eficiente de los aviones que lleguen a la aproximación, que depende de la densidad de tráfico, los tipos de aviones y el clima. Para conseguir eficiencia en el manejo de llegadas y salidas se tiene que conseguir un balance entre un procesamiento rápido del tráfico, cumplir con la capacidad del aeropuerto, reducir tiempos y distancias de vuelo, combustible, emisiones y ruido, cumpliendo además con los requerimientos para la seguridad de las operaciones.

Los objetivos estratégicos más importantes son, según el Manual de CDO, la seguridad, la capacidad, la eficiencia, el acceso y el medio ambiente. En el caso del medio ambiente hay varias consideraciones en las que basar las decisiones al diseñar procedimientos. Se puede perseguir la reducción de ruido, un consumo de combustible más eficiente y, por tanto, una reducción de emisiones, o una combinación de ambas. El diseño de CDO tiene que hacer una balanza. En el despegue, por ejemplo, lo más eficiente en cuanto a consumo de combustible es un ascenso ininterrumpido, pero es necesario evadir áreas pobladas para evitar la contaminación acústica.

Es necesario además asegurar que los procedimientos con CDO no comprometen la seguridad y la capacidad del aeropuerto. Estos factores hacen que no siempre sea posible volar el CDO más óptimo posible. Puede ser necesario detener un descenso y mantener el nivel de vuelo para poder mantener la separación entre aeronaves indicada por ATC. Por tanto el objetivo no es conseguir el CDO más óptimo posible, sino el mejor CDO que no

comprometa la capacidad o la seguridad.

Normalmente el tráfico de llegada y de salida son independientes, y se puede asegurar por separado que ambos tengan rutas con la mayor eficiencia posible en el consumo de combustible.

2.9.2. Métodos de diseño de CDO

Según el Manual de CDO [8], existen dos métodos de diseño de procedimientos CDO que estén basados en rutas fijas lateralmente. Según el método la distancia de vuelo hasta la pista varía. Las dos metodologías de diseño son:

- Los diseños de trayectoria cerrada tienen definida la trayectoria de vuelo hasta el FAF/FAP, por lo que la distancia exacta hasta la pista es conocida antes de empezar la maniobra.
- Los diseños de trayectoria abierta son aquellos en los que la trayectoria no llega a estar definida hasta el FAF/FAP, y por tanto no se conoce la distancia hasta la pista.

En la figura 2.9 se muestra un ejemplo de diseño de trayectoria cerrada. Se puede comprobar que solo hay un posible camino hasta llegar a la pista, por lo que la distancia es conocida.

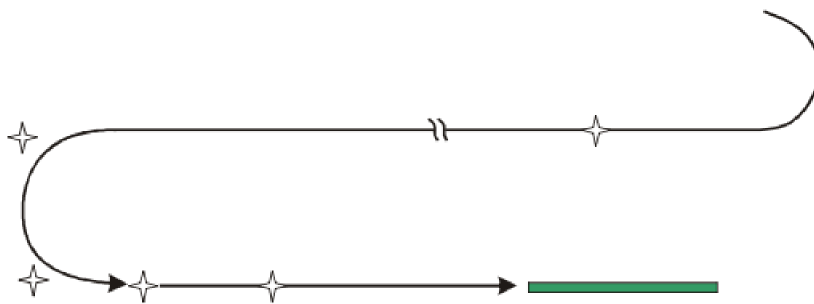

Figura 2.9: Ejemplo de diseño de trayectoria cerrada [8]

Para este proyecto son más relevantes los diseños de trayectoria abierta, sin embargo. En este tipo de diseños una parte o toda la ruta consisten en vectorización. En la figura 2.10 se muestra un ejemplo de procedimiento en el que toda la ruta es vectorizada. Se ve como no hay ninguna parte del trayecto que sea conocida antes de iniciar el CDO, es decir, cada avión volará una trayectoria diferente que le asigne ATC.

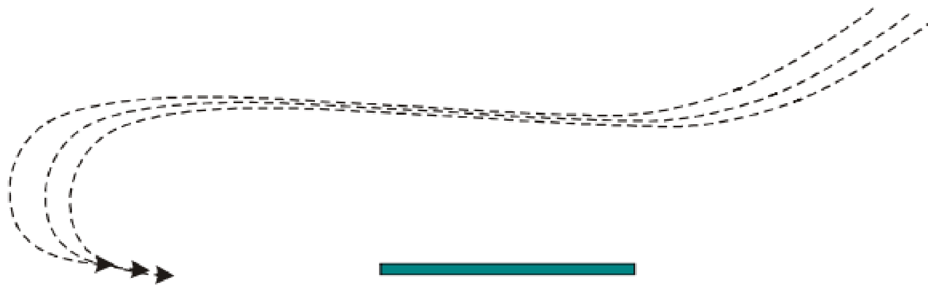

Figura 2.10: Ejemplo de diseño de trayectoria abierta vectorizada [8]

En los procedimientos CDO a sotavento se parte de una ruta fija que envía aviones a un segmento vectorizado, normalmente como una extensión del tramo a sotamento al FAF/FAP. Se muestra en la figura 2.11 el aspecto de este tipo de procedimiento.

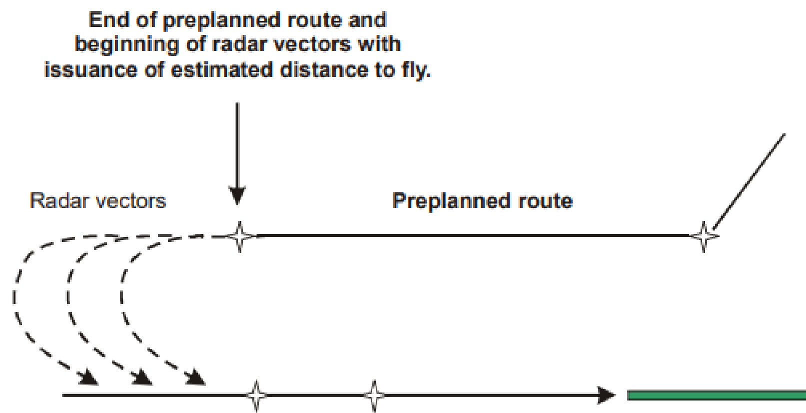

Figura 2.11: Ejemplo de diseño de trayectoria abierta vectorizada parcialmente [8]

2.9.3. Métodos de secuenciación

Para asegurar que los aviones lleguen al aeropuerto manteniendo una tasa de aterrizajes óptima o como mínimo que cumpla con la capacidad del aeropuerto es necesario aplicar secuenciación. La secuenciación es el método usado para asegurar que los aviones guarden la distancia suficiente para cumplir con este criterio. En escenarios de bajo tráfico no es necesario aplicar secuenciación, pero con tráfico elevado se vuelve imprescindible. El Manual de CDO [8] nombra los siguientes 3 métodos:

- **Métodos de secuenciación automatizados.** Son sistemas como un tiempo e llegada requerido, visualizaciones de asesoramiento de ATC e indicadores de posición relativa.

- **Velocidad.** El control de velocidad es el método más eficaz para pequeños ajustes al principio de un procedimiento. Permite establecer y mantener separaciones entre aviones de diferentes tipos. Ajustes de velocidad demasiado grandes fuerzan a

los aviones a mantener configuraciones de vuelo poco óptimas, por lo que no son recomendados.

- **Vectorización.** Es el método más flexible pero causa la menor predictibilidad a los pilotos. Se puede facilitar estimaciones de la distancia a los pilotos para mitigar las incertidumbres.

Además de los definidos por el Manual de CDO, se destaca la secuenciación mediante **circuitos de espera**, que es la forma más básica de secuenciar en aproximaciones. Consiste en hacer que las aeronaves se queden dando vueltas a los circuitos hasta que se consiga la distancia deseada con el resto de aeronaves que están volando la aproximación.

2.9.4. Vectorización

La vectorización consiste en asignar a las aeronaves una trayectoria o bien desviada de la original o, en el caso de las trayectorias abiertas, asignar la trayectoria en sí al no haber original. Hay dos tipos principales de vectorización en rutas:

- En la secuenciación mediante *point merge* los pilotos continúan atravesando los tramos de secuenciación hasta que reciben un permiso de ATC de "directo a", tras lo que se dirigen al punto de convergencia. Se muestra el aspecto de la maniobra en la figura 2.12.

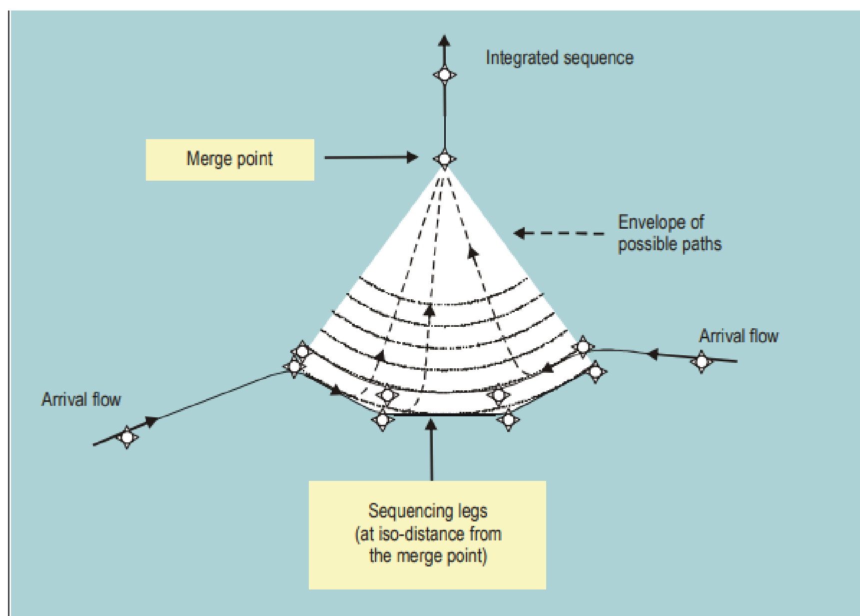

Figura 2.12: Maniobra de *point merge* [8]

- El *path stretching* es un método de vectorización que tiene waypoints predeterminados que son conocidos previamente por ATC y el piloto. Se muestra el aspecto básico de la maniobra en la figura 2.13. Según la distancia que se desee conseguir ATC envía al avión al tramo que añada los minutos necesarios.

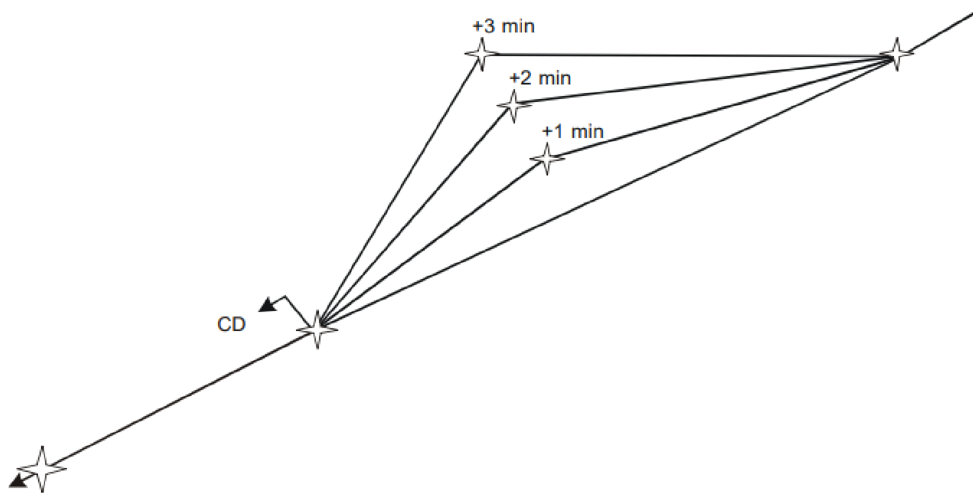

Figura 2.13: Maniobra de *path stretching* [8]

2.10. Tipos de aproximaciones

Una vez conocidos los métodos principales de secuenciación se exponen varios de los tipos de aproximaciones en función de cómo secuencia el tráfico.

2.10.1. Aproximación directa

Las aproximaciones directas son el tipo de aproximaciones más común para aeropuertos de bajo y medio tráfico. Un ejemplo es el Aeropuerto de Braunschweig/Wolfsburg (EDVE), que cuenta con 12000 movimientos IFR (instrumentales) por año. Otros aeropuertos más grandes como Halle/Leipzing (EDDP) utilizan este esquema por la noche cuando hay poco tráfico. [9]

Cuentan con la ventaja de tener un diseño simple y su adaptabilidad a la situación del tráfico, pero no son adecuados para situaciones de mayor tráfico. Se muestra el esquema en la figura 2.14.

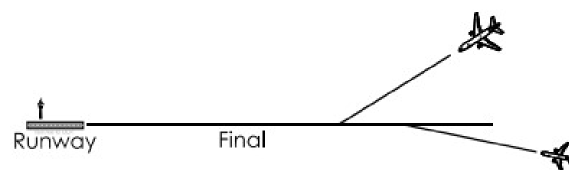

Figura 2.14: Esquema de una aproximación directa [9]

2.10.2. Patrones de apilamiento

Es un tipo de aproximación en el que los aviones son dirigidos a **holding stacks** verticales en los que descienden un nivel cada vez que reciben permiso, como se muestra en la figura 2.15.

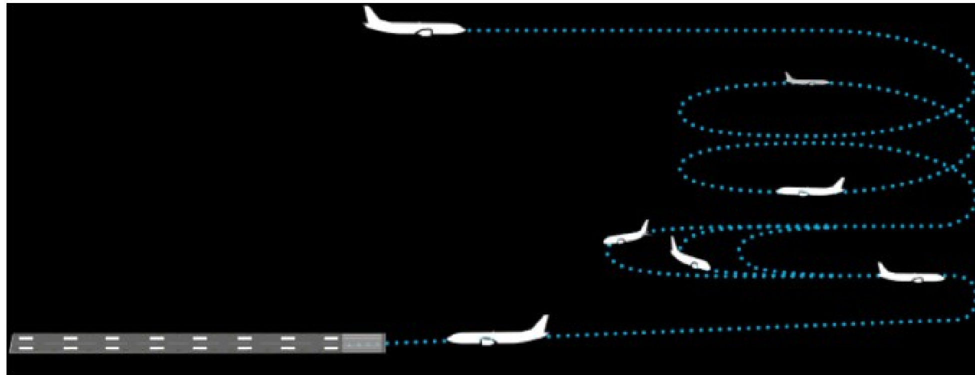

Figura 2.15: Esquema de un holding stack vertical [10]

Una vez que los aviones abandonan el *holding stack* son dirigidos por ATC a la aproximación final. Se muestra una vista cenital del procedimiento en la figura 2.16.

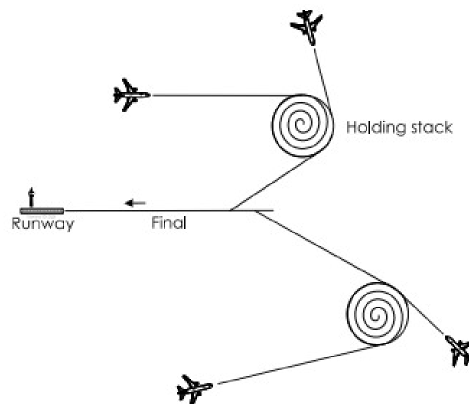

Figura 2.16: Esquema de una aproximación con stacking [9]

2.10.3. Point merge

Esta aproximación implementa la secuenciación *point merge* que se ha explicado anteriormente. Los aviones son secuenciados mediante la decisión de cuándo abandonan el arco de secuenciación. Cuando ATC considera que hay una separación adecuada, da instrucciones al piloto de que haga ruta directa al *merge point*.

La distancia al *merge point* es la misma a lo largo de todos los tramos de secuenciación, por lo que no son arcos circulares, sino segmentos que forman un casi arco centrado en el *merge point*. Se muestra el esquema en la figura 2.17.

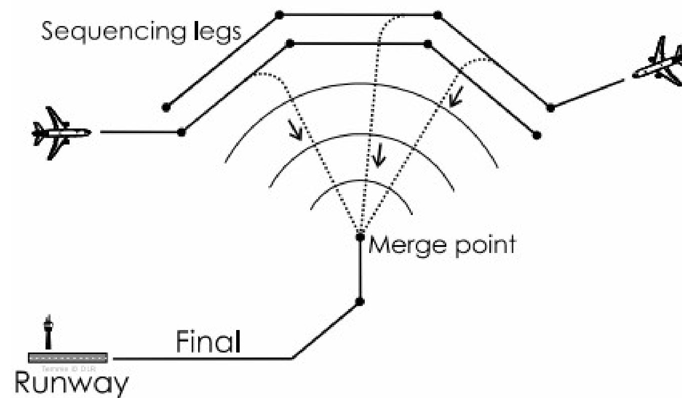

Figura 2.17: Esquema de una aproximación con point merge [9]

Los beneficios de este sistema se basan en la creación de espacio entre los aviones sin intervención de ATC, que girar independiemntemente el arco y luego solo reciben el permiso *direct to*. Al permanecer a alturas elevadas en el arco el procedimiento disminuye el ruido, y también reduce los contactos necesarios entre tierra y aire.

Sin embargo, tiene algunas desventajas. Por ejemplo, en situaciones de bajo tráfico se consume mucho tiempo de vuelo. Además puede procesar solamente dos flujos de tráfico por cada pista.

2.10.4. Aproximación Fan

Se utiliza para aeropuertos de medio y alto tráfico, y consiste en un fijo de medición a partir del cual ATC envía a los aviones en diferentes ángulos hacia el tramo recto de la aproximación final. Si es necesario se puede separar a viones por niveles de vuelo. Si la distancia entre dos aviones es demasiado baja, el primer avión en llegar toma el rumbo más corto y el segundo toma otro rumbo suficientemente separado. ATC también puede variar el tiempo en el que dan permiso para pasar al tramo fina, cambiando el ángulo de intersección. Se muestra el esquema en la figura 2.18.

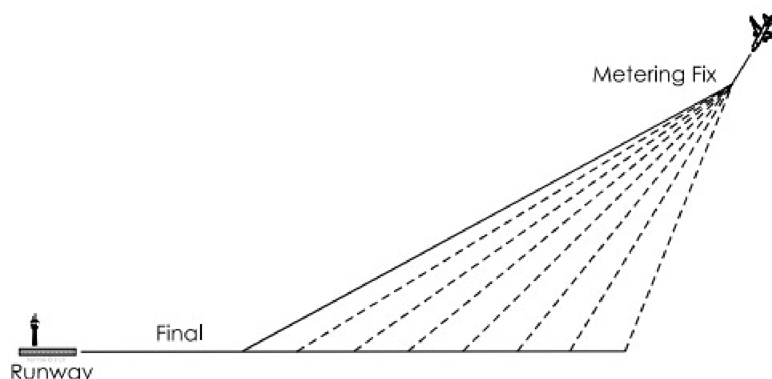

Figura 2.18: Esquema de una aproximación fan [9]

2.10.5. Aproximación en trombón

La aproximación en trombón es un tipo de aproximación que utiliza secuenciación *path stretching* y es muy útil es escenarios de alta intensidad de tráfico. Básicamente, consiste en secuenciar el tráfico haciendo que cada avión tenga un tramo de alejamiento diferente antes de hacer el giro para dirigirse al IF.

La estructura en trombón es una manera simple de conseguir el tiempo objetivo cuando los aviones llegan desde otra dirección hasta la final. Los aviones son ordenados desde los dos lados del flujo de tráfico de tráfico hacia el final. Cuando llegan a la posición ideal, los pilotos son instruidos para girar al final. Se muestra el esquema en la figura 2.19

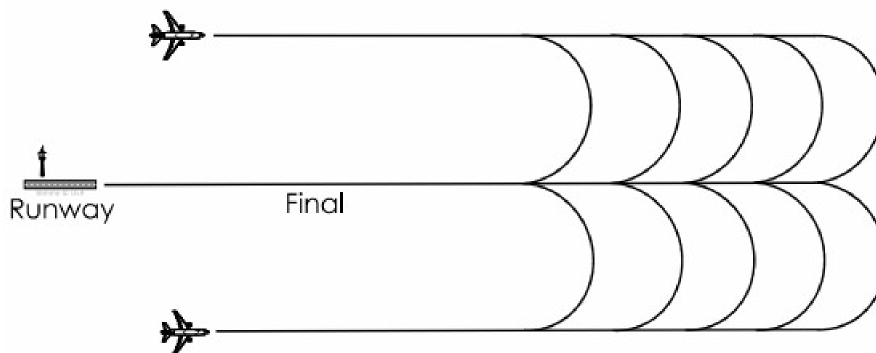

Figura 2.19: Esquema de una aproximación en trombón [9]

Hay variaciones de este esquema, como el caso del Aeropuerto Internacional de Munich (EDDM), en el que hay un trombón abierto, lo que significa que los aviones deben esperar en sotavento hasta que reciben el permiso de giro. El Aeropuerto de Frankfurt (EDDF) tiene una estructura cerrada, es decir, que si el avión no recibe permiso, vuela hasta el final del alejamiento y hacen el último giro posible. Estos dos tipos generan diferencias menores según la situación del tráfico.

Puede existir waypoints virtuales en las líneas centrales y final que ATC puede autorizar cuando el avión está aún en el alejamiento y lejos de su giro planificado. Su uso puede o no ser necesario para mantener la flexibilidad en el guiado.

Algunos aeropuertos grandes tienen un sistema de pistas paralelas dependientes o independientes que implementan un doble trombón con alejamientos separados para cada pista. En el ejemplo de la figura 2.20 los aviones que llegan del norte será dirigidos habitualmente al trombón norte y aviones del sur al trombón sur. Pero en el caso de que haya un desbalance en el tráfico que viene de ambas direcciones ATC tiene la posibilidad de autorizar a algunos aviones para que cambien al final de la pista paralela, aunque conlleva riesgos. Lo más habitual es llevar aviones alrededor del trombón hasta el final. Si la distancia entre las pistas es muy amplia pueden ser operadas independientemente, pero si están poco separadas se deben operar como una sola.

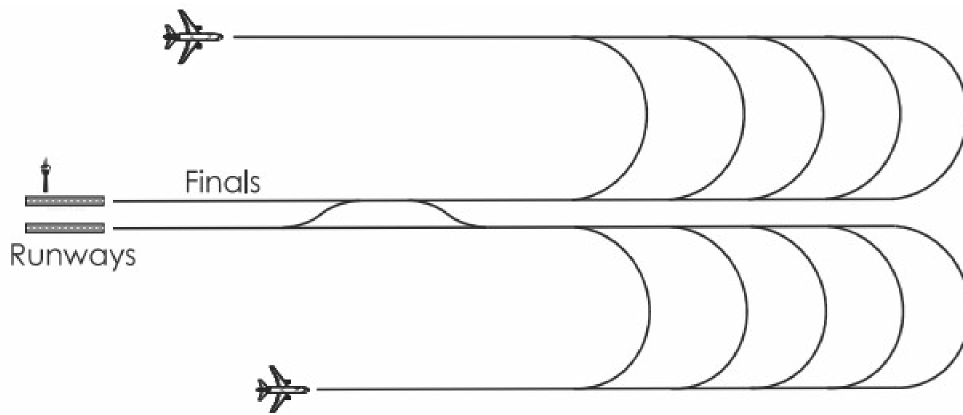

Figura 2.20: Esquema de una aproximación en trombón con pistas paralelas [9]

En la figura 2.21 se puede ver el tramo en trombón que se va a analizar en el proyecto, con giros marcados en rojo.

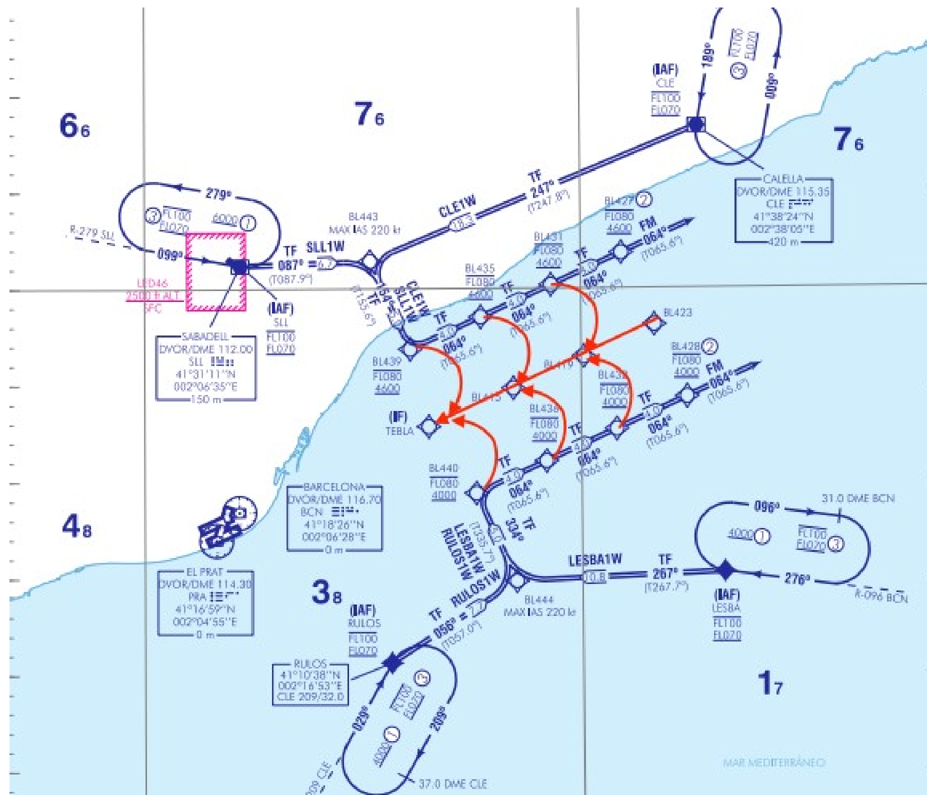

Figura 2.21: Tramo en trombón del Aeropuerto de Barcelona-El Prat [11]

2.11. Capacidad de un aeropuerto

La FAA define la capacidad como la medida del máximo número de operaciones de aeronaves que pueden ser realizadas en un aeropuerto o componente de aeropuerto en una

hora. La capacidad de un componente de un aeropuerto es independiente de la de otros componentes, por lo que puede ser calculada por separado [12].

La capacidad de un aeropuerto está afectada por muchos factores, entre ellos la configuración, número, espacio y orientación de los sistemas de las pistas, la configuración, número y tamaño de las vías de taxi, salidas de la pista, rutas de tráfico, el tamaño y mezcla de aeronaves usando las instalaciones, las condiciones meteorológicas, los procedimientos de salida y llegada de aeronaves, la presencia de aeropuertos cercanos, las características de los sistemas de radar, la adopción de soluciones anti ruido o la carga de trabajo de los controladores de la torre [13].

Los análisis de capacidad tienen en cuenta la influencia de estos elementos en temas de seguridad, economía y logística. Además, las modificaciones en la capacidad de un aeropuerto afectan a la política de medio ambiente del operador del aeropuerto.

La capacidad de la pista de un aeropuerto es calculada como la inversa de la separación en el tiempo entre dos aviones. Es necesario asumir ciertas condiciones para poder calcular la capacidad, ya que las condiciones climáticas, la velocidad de cada avión en la pista, interferencias en los despegues o aterrizajes pueden hacer variar la capacidad.

2.12. Aeropuerto de Barcelona-El Prat

Según el Plan Director del Aeropuerto de Barcelona [14] las primeras instalaciones de este se sitúan en el año 1916 cerca de El Remolar. Después, entre 1939 y 1941 se construye el Aeroclub de Cataluña. También en 1941 se inicia otra ampliación de las instalaciones, que acaba en 1946. Es en 1970 cuando dos de las pistas adquieren su estado actual, y en 1990 se realiza una gran actualización.

El Aeropuerto de Barcelona es el segundo aeropuerto con más tráfico de España. Cuenta con 3 pistas, dos en paralelo (06L/24R y 06R/24L) y una cruzada (02/20), y 2 terminales, T1 y T2, siendo T2 una combinanción de los anteriores terminales T2A, T2B y T2C [15]. En la figura 2.22 se muestra una imagen aérea del Aeropuerto de Barcelona.

Figura 2.22: Vista aérea del Aeropuerto de Barcelona-El Prat

2.12.1. Proyecto BRAIN

El proyecto BRAIN (Barcelona RNAV Approach Innovations) fromaba parte del Plan Verano de ENAIRE, y tenía por objeto según ENAIRE: "*optimizar y agilizar el flujo de tráfico, disminuyendo la complejidad de las maniobras en todas las configuraciones, mejorando la distribución de las llegadas y reduciendo el número de esperas en el aire y en las proximidades del aeropuerto*"[16].

Se trata de dotar al aeropuerto de procedimientos de aproximación en trombón mediante una estructura de waypoints para optimizar las trayectorias. Con esto se pretende mejorar la eficiencia medioambiental y en la operación de aeronaves, con una estandarización de las operaciones y una reducción de las comunicaciones con los controladores aéreos para poder reducir su carga de trabajo.

El 26 de abril de 2018 se inició la implantación de las nuevas maniobras de aproximación, después de que los controladores de ENAIRE hayan estado recibiendo formación mediante simuladores. BRAIN mejora la distribución de las legadas y disminuye el tiempo de esperas necesarias para la secuenciación de los aviones [17].

El 29 de enero de 2018 la DIrección General de Calidad y Evaluación Ambiental y Medio Natura emite la resolución que da un informe de impacto medioambiental favorable para el nuevo procedimiento.

En la figura 2.23 se puede observar el esquema que desarrolló ENAIRE para los nuevos procedimientos implementados con BRAIN.

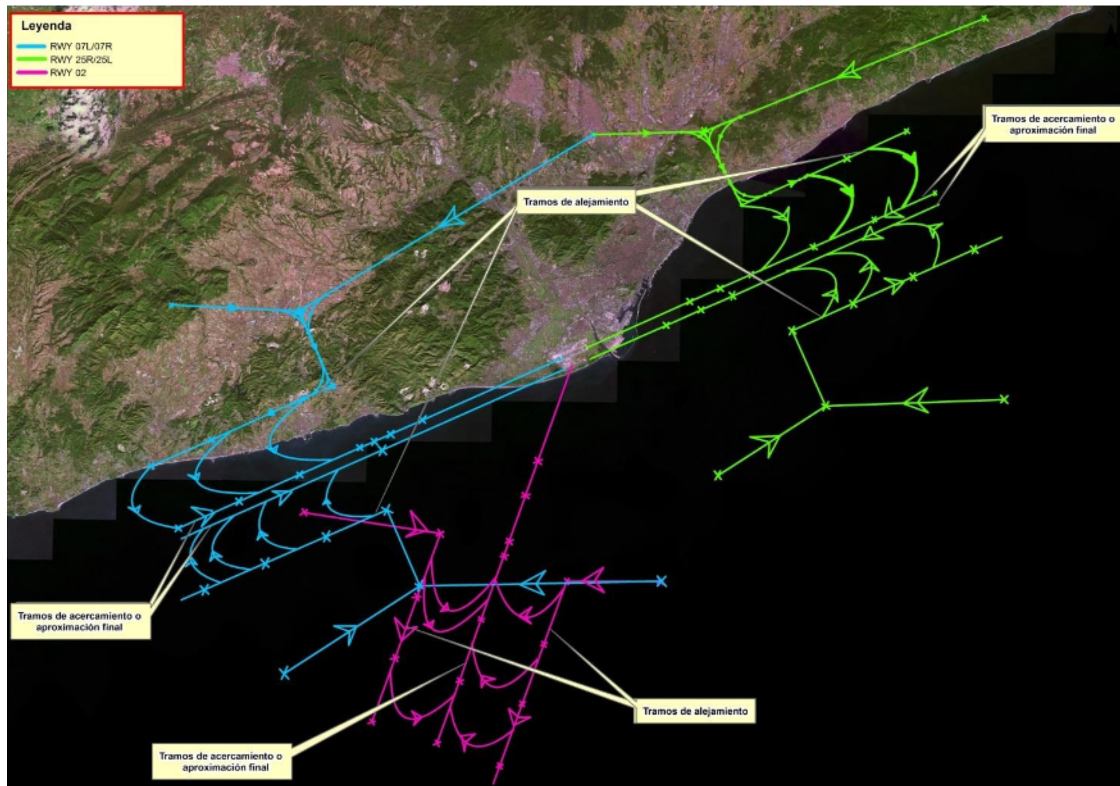

Figura 2.23: Esquema del proyecto BRAIN [16]

2.13. Sistemas de coordenadas

Un sistema de coordenadas es un marco en el que se definen las ubicaciones relativas de los objetos representados en un área concreta. Para poder implementar los procedimientos estudiados es necesario explicar los sistemas de coordenadas relevantes para el proyecto: las coordenadas geográficas y las ENU. Las coordenadas geográficas son las que aparecen en las cartas que se van a utilizar para obtener información y las ENU son en las que se desarrollará el proyecto.

2.13.1. Coordenadas geográficas

Las coordenadas geográficas utilizan latitud, longitud y altitud para representar la posición relativa a un elipsoide de referencia (representación de la superficie de la Tierra aproximándola a un elipsoide). Son las siguientes [18]:

- La latitud de un punto es el ángulo normal al elipsoide en ese punto que hace con el plano ecuatorial. Va de -90º (Sur) a +90º (Norte). Se muestra una representación en la figura 2.24.

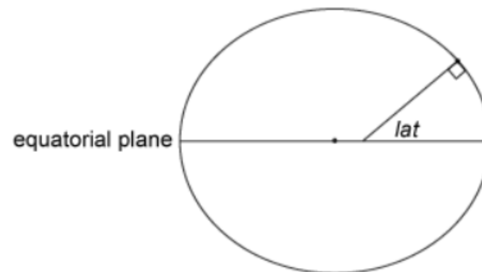

Figura 2.24: Representación de la latitud [18]

- La longitud de un punto es el ángulo entre un plano que contiene al centro del elisoide y el meridiano que contiene al punto y el plano que contiene el centro del elipsoide y el meridiano que se toma como 0º. Normalmente se usa el rango de -180º (Oeste) a +180º (Este). Se muestra en la figura 2.25.

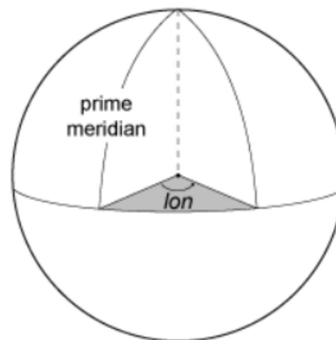

Figura 2.25: Representación de la longitud [18]

- La altitud es la longitud medida en la normal del punto al elipsoide. Se muestra en la figura 2.26

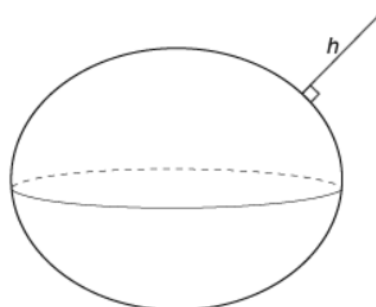

Figura 2.26: Representación de la altitud [18]

2.13.2. Coordenadas ENU

Las coordenadas ENU (East-North-Up) utilizan las coordenadas cartesianas (x, y, z) para representar la posición relativa a un origen local que puede o no estar en la superficie de elipsoide [18]. La x representa la dirección al este del paralelo de la latitud del punto de referencia. La y representa la dirección norte a lo largo del meridiano que contiene la longitud del punto. La z apunta al norte de la normal a la elipsoide. Se representa el sistema en la figura 2.27.

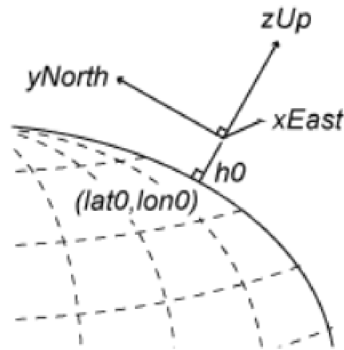

Figura 2.27: Representación de las coordenadas ENU [18]

Capítulo 3

Herramientas

En este capítulo se tratarán las herramientas informáticas que se han utilizado para desarrollar este proyecto, que son principalmente la programación orientada a objetos y Java como lenguaje de programación que implementa la programación orientada a objetos. También se utilizará brevemente MATLAB para hacer las transformaciones de coordenadas.

3.1. Programación orientada a objetos

La programación orientada a objetos (OOP) está basada en la interacción de los objetos, en la que está basado el diseño de aplicaciones y programas [19].

3.1.1. Clases y objetos

Una clase es una entidad que se utiliza en OOP para definir las características o propiedades de un concepto que se quiere representar. Las clases permite modalidad y flexibilidad a los programas, y utilizar una estructura diferente a la programación tradicional. Las clases contiene atributos y métodos:

- Los **atributos** son las variables que contienen información sobre el concepto que se representa con la clase.
- Los **métodos** son el equivalente a las funciones, que permite realizar cambios en los atributos del objeto o ejecutar operaciones y enviar información a otros objetos.

Los **objetos** son las instancias de las clases, con sus propios atributos para definir sus propiedades y con métodos para realizar cambios en estos. Los atributos y métodos permite a los objetos definir como ser usados por otras clases.

La programación basada en objetos tiene los siguientes beneficios [19]:

- **Modularidad.** El código de cada objeto se puede escribir y guardar independientemente.
- **Ocultación de información.** Si solo se permite la interacción con lo métodos de un objeto, con lo que desde otros objetos no se puede acceder a otros componentes del objeto.
- **Reutilización del código.** Los objetos creados pueden ser utilizados en cada nuevo programa, lo que hace innecesario crear un nuevo objeto para realizar la misma tarea.

- **Facilidad en insertabilidad y depuración.** Es posible reemplazar un objeto que falle por otro sin cambiar el resto del programa.

3.1.2. Pilares de la programación orientada a objetos

La programación orientada a objetos cuenta con los siguientes pilares:

- **Abstracción.** Se puede aislar a los objetos de los otros elementos del código, lo que permite expresar las características importantes que identifican a ese objeto y no necesitar los detalles irrelevantes para trabajar con estos objetos.

  La abstracción está nivelada, es más baja cuanto más se acerca al detalle concreto y más alta cuanto más restringido está el acceso a las características de los onjetos.

- **Encapsulamiento.** Es la manera en que se oculta la información interna de un objeto, haciendo más complicado que reciba modificaciones externas. El encapsulamiento no aumenta la complejidad del programa. Se puede elegir el tipo de encapsulamiento de los datos mediante los modificadores: public, protected, default y private.

- **Herencia.** Es un mecanismo jerárquico que permite crear objetos complejos desde otros más simples. Este nuevo objeto tiene los métodos y los atributos de la clase de la que hereda, y puede añadir los suyos propios. Es una forma útil de crear objetos más específicos del grupo que represente la clase original. Una clase abstracta es una clase que solo se puede untilizar para ser heredada.

  La clase que hereda se llama subclase y la original superclase. Con el comando extends se heredan las propiedades de la clase a la que se haga referencia. Se muestra un ejemplo de esto en la figura 3.1.

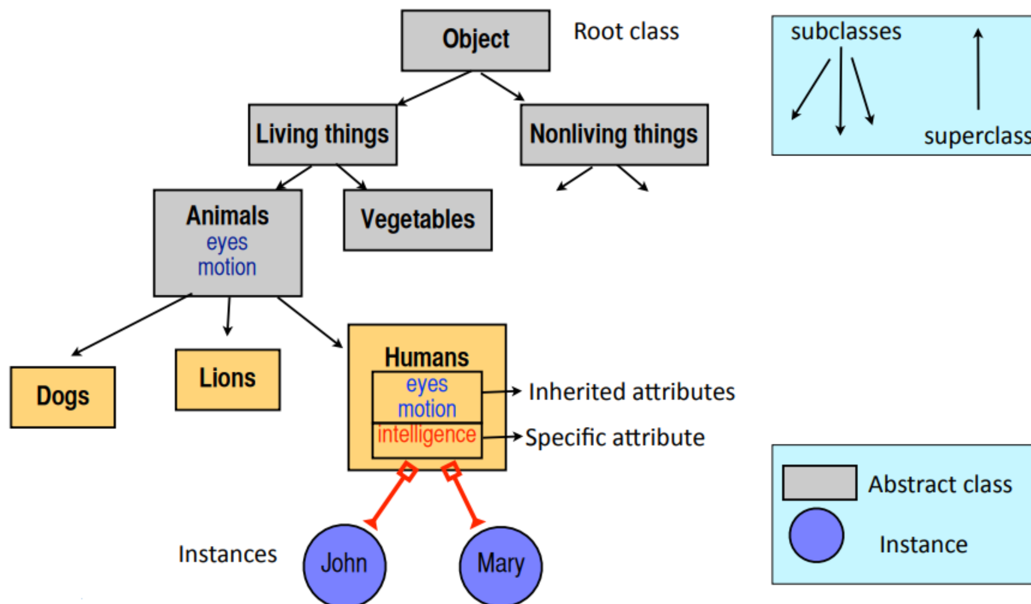

Figura 3.1: Ejemplo de herencia en Java [20]

- **Polimorfismo.** Es la capacidad de poder enviar mensajes idénticos sintácticamente a objetos de tipo diferente. Es decir, se puede manejar un objeto como se haría con uno genérico.

  Una propiedad del polimorfismo es que una referencia a una superclase puede apuntar a sus subclases dado que la operación se selecciona según la clase, no la referencia.

  Otra propiedad es que dos objetos de distintas jerarquías pueden responder a los mismos mensajes mediante las interfaces. Los objetos con la misma interfaz se pueden manejar igual.

Gracias a las propiedades de las clases, objetos, atributos y métodos los lenguajes de programación orientada a objetos comparten la propiedad de la **modularidad**. Esto significa que los programas se pueden dividir en bloques pequeños que realizan funciones concretas de manera independiente al resto del programa. Esta propiedad permite seccionar los diferentes aspectos del programa de manera separada y poder añadir nuevas funciones fácilmente mediante la introducción de nuevos bloques.

3.2. Java

Como se ha establecido en la introducción, se utilizará Java para desarrollar el proyecto. Los motivos por los que se ha elegido Java son:

- El **contexto gráfico** de Java es importante para la realización del proyecto, ya que el resultado va a ser una representación visual de las aproximaciones de diferentes aviones, y Java tiene una gran variedad de opciones sencillas de implementar mediante las Interfaces Gráficas de Usuario (GUI).

- Los **threads** (hilos) de Java permiten realizar varias tareas simultáneamente, lo que es muy conveniente para simular la gestión de ATC de los diferentes aviones que van llegando al circuito. Esto permite que ATC pueda manejar los aviones en tiempo real mientras los aviones realizan su circuito de espera.

- Los beneficios propios de utilizar **programación orientada a objetos** son importantes para el proyecto, ya que permiten crear diferentes clases y subclases para los waypoints, posiciones, rutas... que facilitan su implementación en la simulación.

3.2.1. Introducción a Java

Java es un lenguaje de programación concurrente basado en objetos, con las propiedades que se han explicado previamente, y que posee una alta abstracción del lenguaje máquina, lo que lo hace un lenguaje de alto nivel, más fácil de usar que uno de bajo nivel.

Presenta además un tipado fuerte, lo que indica que una variable de cierto tipo no se puede utilizar en operaciones que requieran otro tipo sin realizar una conversión previamente.

Al compilar Java es capaz de detectar errores, lo que ayuda a la hora de refinar el código. También permite traducir los programas a *bytecode*, que es un código guardado en un archivo binaro que permite ser ejecutado en una máquina virtual Java independientemente de la arquitectura del ordenador.

Se suele trabajar con Java mediante un Entorno de Desarrollo Integrado (IDE), que es la aplicación que proporciona las herramientas para facilitar la programación. Se suele componer de un editor de código fuente, un depurador y herramientas de construcción automáticas. El IDE utilizado en este caso es Apache NetBeans IDE, específicamente la versión 17 del programa. Netbeans IDE es de código abierto y permite ser utilizado en Windows, Linux, Mac y Solaris.

A continuación se explicarán diferentes características de Java.

3.2.2. Concurrencia: threads

La concurrencia es las capacidad de ejecutar al mismo tiempo varias tareas, es decir, en paralelo.

Es la propiedad que usan los **threads** mencionados anteriormente. Los threads, como se ha comentado anteriormente, son objetos que se ejecutan concurrentemente con otros threads,. Ejecutar varios threads a la vez es una de las capacidades mñas importantes de Java.

Los threads permiten desarrollar tareas secundarias a las que realiza la parte principal del programa, aunque se puede realizar un programa en que la parte importante sean los threads en sí y el programa principal se encargue de crearlos únicamente. Los threads se crean mediante lo que se conoce conoce como hilo padre y comparte los recursos del programa, como la memoria y los archivos abiertos. En la figura 3.2 se muestra una representación simple de los threads.

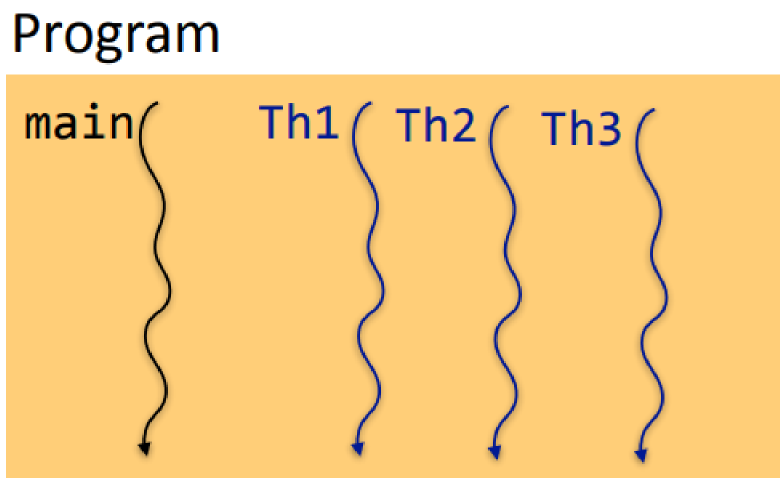

Figura 3.2: Representación de threads concurrentes [20]

La interacción entre los threads se puede explicar mediante el ejemplo de la figura 3.3. Se tiene un **producer** que crea los datos, un **consumer** que los utiliza para algo. Los dos threads se comunican mediante un objeto compartido llamado **Drop**. Es necesario que los threads se coordinen, el consumer no puede intentar recibir los datos antes de que el producer los haya entregado, y el producer no debe intentar entregar datos nuevos antes de que el consumer no haya recibido los antiguos.

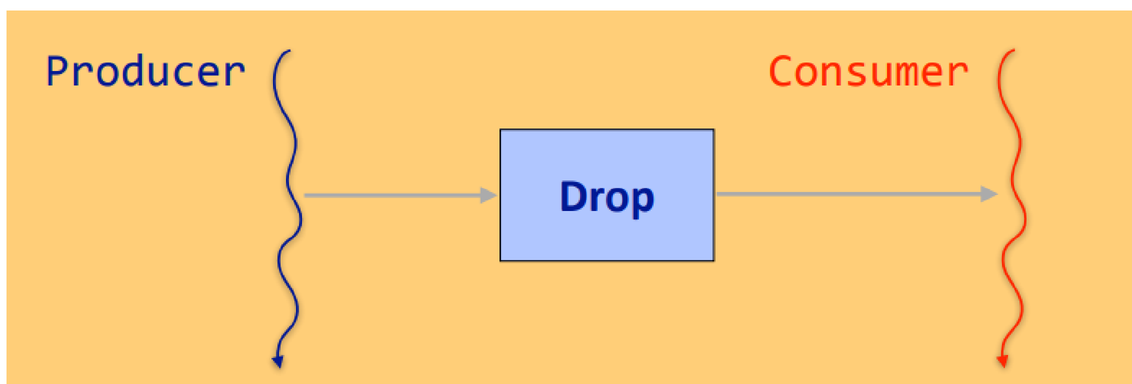

Figura 3.3: Ejemplo de comunicación entre threads [20]

3.2.3. Colecciones: List y Map

Las colecciones de Java son objetos que sirven para representar un grupo elementos, que son otros objetos. [20]

Contienen métodos para realizar operaciones básicas, como:

- **int size()** permite conocer el tamaño de la colección
- **boolean isEmpty()** indica si la colección está vacía. Es útil para comprobar que la colección con la que se trabaja no está vacía antes de realizar una operación con ella, para evitar causar un error.
- **boolean contains(Object element)** devuelve *true* si la colección contiene el elemento que se indica.
- **boolean add(E element)** añade el elemento a la colección.
- **boolean remove(Object element)** elimina la primera instancia del elemento, si la colección lo contiene.
- **Iterator¡E¿iterator()** devuelve un iterador sobre los elementos de la colección.

También contiene métodos que permite operar en las colecciones enteras, como:

- **boolean containsAll(Collection¡?¿c)** devuelve *true* si la colección contiene todos los elementos de la colección indicada.
- **boolean addAll(Collection¡? extends E¿c)** añade todos los elementos de la colección indicada a la colección.
- **boolean removeAll(Collection¡?¿c)** elimina la primera instancia de cada elemento de la colección c indicada, si la colección sobre la que se opera lo contiene. Es decir, itera el método remove() sobre la colección dada.
- **boolean retainAll(Collection¡?¿c)** Itera sobre los elementos de la colección c comprobando si la colección los contiene, y después elimina los elementos de esta colección que no estaban contenidos en la colección c.
- **void clear()** elimina todos los elementos de la colección, dejándola vacía.

Las colecciones más importantes son las listas y los mapas, de las cuales las listas son las más importantes para el proyecto.

Mapas

Son colecciones que asignan claves a valores. Se pueden entender como un diccionario que contiene grupos de par-valor, no pudiendo contener claves duplicadas y sin establecer un orden específico, lo que las diferencia de las listas.

Listas

Las listas son colecciones ordenadas en las que se tiene un control preciso sobre en que posición concreta de la lista se coloca un elemento.

Permite acceder elementos por su índice de tipo int (su posición en la lista) y buscar elementos en la listas. También puede contener elementos duplicados.

En este proyecto las listas presentan el mecanismo ideal para representar el orden de prioridad que ATC les otorgue a los aviones, estando estos ordenados según su prioridad.

Además de los métodos generales de las colecciones, los siguiente métodos de las listas son relevantes:

- **Element get(int index)** devuelve el elemento que se corresponde con el índice introducido.
- **Element remove(int index)** elimina el elemento que se corresponde con el índice introducido.
- **Elemente set(int index, E element)** reemplaza el elemento de la posición indicada por el índice por el nuevo elemento introducido.

3.2.4. Sockets

Los Sockets son una funcionalidad de Java que permite establecer un enlace entre programas independientes mediante una red con la que se comunican los dispositivos. Dado que los Sockets no se utilizan en este proyecto se explicarán brevemente.
Los sockets utilizan una arquitectura cliente-servidoe mediante los protocolos TCP y UDP. El TCP (Transmission COntrol Protocol) establece un conducto exclusivo entre los equipos que se comunican y lo cierra al finalizar la comunicación (comunicación punto a punto entre equipos). El cliente usa la clase Scoket y el servidor ServerSocket.

El protocolo UDP (User Datagram Protocol) envía los datos en datagramas con parte de la información a la red en espera de ser recibidos. El equipo receptor los recibe y debe reconstruir la información que llega sin orden. No se garantiza que llegue toda la información. El que envía datos usa la clase DatagramSockt y el paquete DatagramPacket.

En la figura 3.4 se muestra la estructura de la conexión de los dos protocolos. El diagrama de arriba representa el UDP y el de abajo el TCP.

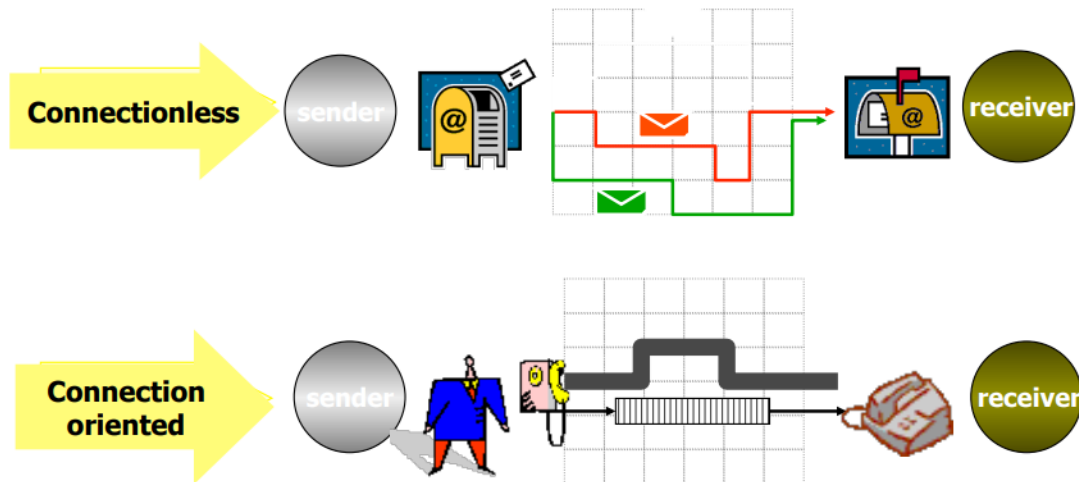

Figura 3.4: Tipos de conexión con sockets [20]

3.2.5. GUI

Las Interfaces Gráficas de Usuario (GUI), son, como se ha indicado anteriormente, una parte importante del motivo por el que se ha elegido Java como lenguaje de programación. Java utiliza *Java Foundation Classes* (JFC) para proporcionar interfaces gráficas a los programas. Tiene los siguientes componentes:

- AWT (Abstract Window Toolkit) es una GUI que depende de la plataforma usada.
- Swing es una GUI basada en IFC (Internet Foundation Classes) de Netscape, y no depende de la plataforma, manjeando de forma propia la interfaz. Es más rápido que AWT, tiene características más avanzadas, requiere el paquete javax.swing y soporta MVC. MVC es un modelo que determina cómo el componente es mostrado en pantalla teniendo en cuenta el estado actual del modelo y cómo reacciona el componente al usuario. Sigue la estructura de la fiura 3.5.

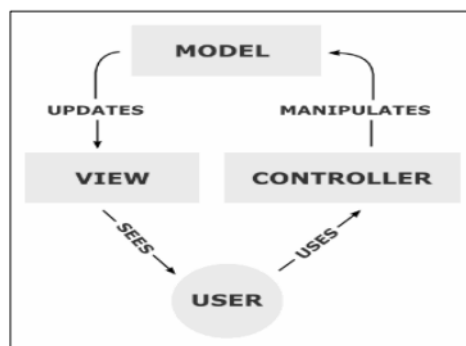

Figura 3.5: Modelo MVC [20]

- Java 2D es una GUI que sirve para dibujar gráficos en dos dimensiones. Al dibujar se trata de rellenar.en pantalla una "formaütilizando un "pincelz çomponiendo.el re-

sultado en pantalla.

La clase JPanel proporciona un área gráfica personalizada en la que se definen las coordenadas del objeto que se quiere pintar respecto al origen del JPanel, como se muestra en la figura 3.6. Se pueden establecer límites para el JPanel y convertir coordenadas reales al nuevo sistema mediante una posición gráfica, en el que cada píxel represente a un conjunto de coordenadas real.

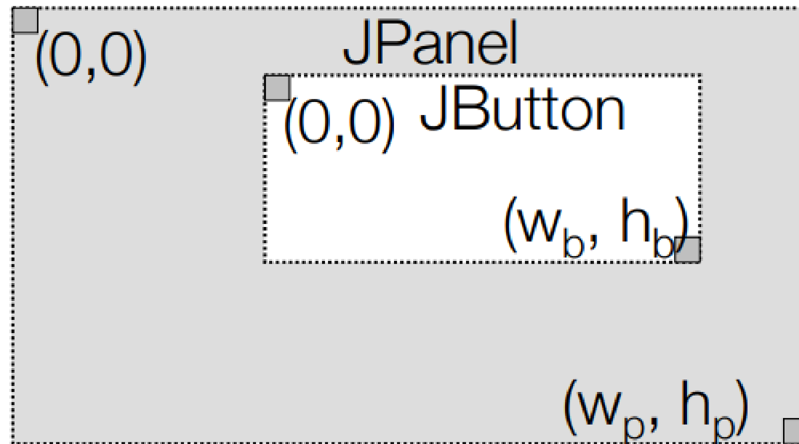

Figura 3.6: Coordenadas en el JPanel [20]

Todas las pantallas dibujan sobre la misma superficie y no recuerdan"lo que tienen debajo, por lo que se debe repintar cuando las porciones tapadas vuelven a exponerse en el frente.

Se debe diseñar el programa para que envíe eventos **repaint()** para volver al pintar si existe solapamiento de ventanas, redimensionamiento o cualquier otra actividad que cambie el dibujo. Estos eventos llaman al método de pintado **paintComponent**, que se puede configurar para que pinte lo que se desea que aparezca en pantalla.

Este proyecto utiliza tanto Swing como Java2D para realizar la interfaz gráfica.

3.3. MATLAB

MATLAB es una plataforma de programación y cálculo numérico desarrollada por MathWorks, con muchas funciones predefinidas y muy sencilla de utilizar, lo que la convierte en ideal para todos los niveles.

En el presente proyecto MATLAB solo se utiliza para realizar las conversiones de coordenadas, por lo que carece de sentido realizar una explicación extensa sobre el programa. Se explicará la función de MATLAB utilizada para realizar las conversiones.

3.3.1. Función lla2enu

La función **lla2enu** convierte de coordenadas geográficas (lat, lon, alt) a coordenadas ENU locales. Su sintaxis es de la forma: xyzENU = lla2enu(lla,lla0,method). Siendo:

- lla: coordenadas (lat, lon, alt) a convertir en una matriz de nx3.
- lla0: coordenadas (lat0, lon0, alt0) del punto de origen del sistema ENU local.
- method: es el tipo de método de transformación. En este caso se ha usado 'flat'.

Capítulo 4

Diseño

En este apartado se explicará la estructura general del programa y la interacción de sus componentes. También se tratarán las decisiones de diseño tomadas, las simplificaciones realizadas, el proceso de traslación de la información de las cartas aeronáuticas al código y similares aspectos no estrictamente relacionados con la interacción entre clases del programa.

4.1. Inserción de las coordenadas en el código

Para realizar la simulación de las aproximaciones tradicionales y de transición al aeropuerto de Barcelona es necesario obtener las coordenadas de los waypoints que forman los procedimientos, lo que se puede hacer a través de las cartas IAC de uno de los procedimientos. Dado que hay varios procedimientos de aproximación al aeropuerto de Barcelona es necesario elegir uno de ellos para cada tipo de aproximación, es decir, uno TRAN y otro tradicional.

En este caso se ha optado por elegir el procedimiento TRAN para RWY 24R y la aproximación VOR también para RWY 24R. Estos dos procedimientos se han elegido por varios motivos. Lo más importante es que los IAF son los mismos en el procedimiento TRAN que en el VOR tradicional, pero este no es el factor decisivo ya que las demás aproximaciones VOR a las otras pistas también lo cumplen respecto a sus TRAN para la misma pista. El principal motivo es que se ha considerado que los procedimientos IAC para RWY 24R eran algo más sencillos visualmente de interpretar, por cuestiones como la colocación de los IAF más a la derecha del aeropuerto, una separación más idónea entre ellos y cuestiones estéticas similares.

4.1.1. Procedimiento tradicional VOR

En la figura 4.1 se muestra el procedimiento de aproximación con VOR a la pista 24R del Aeropuerto de Barcelona [21].

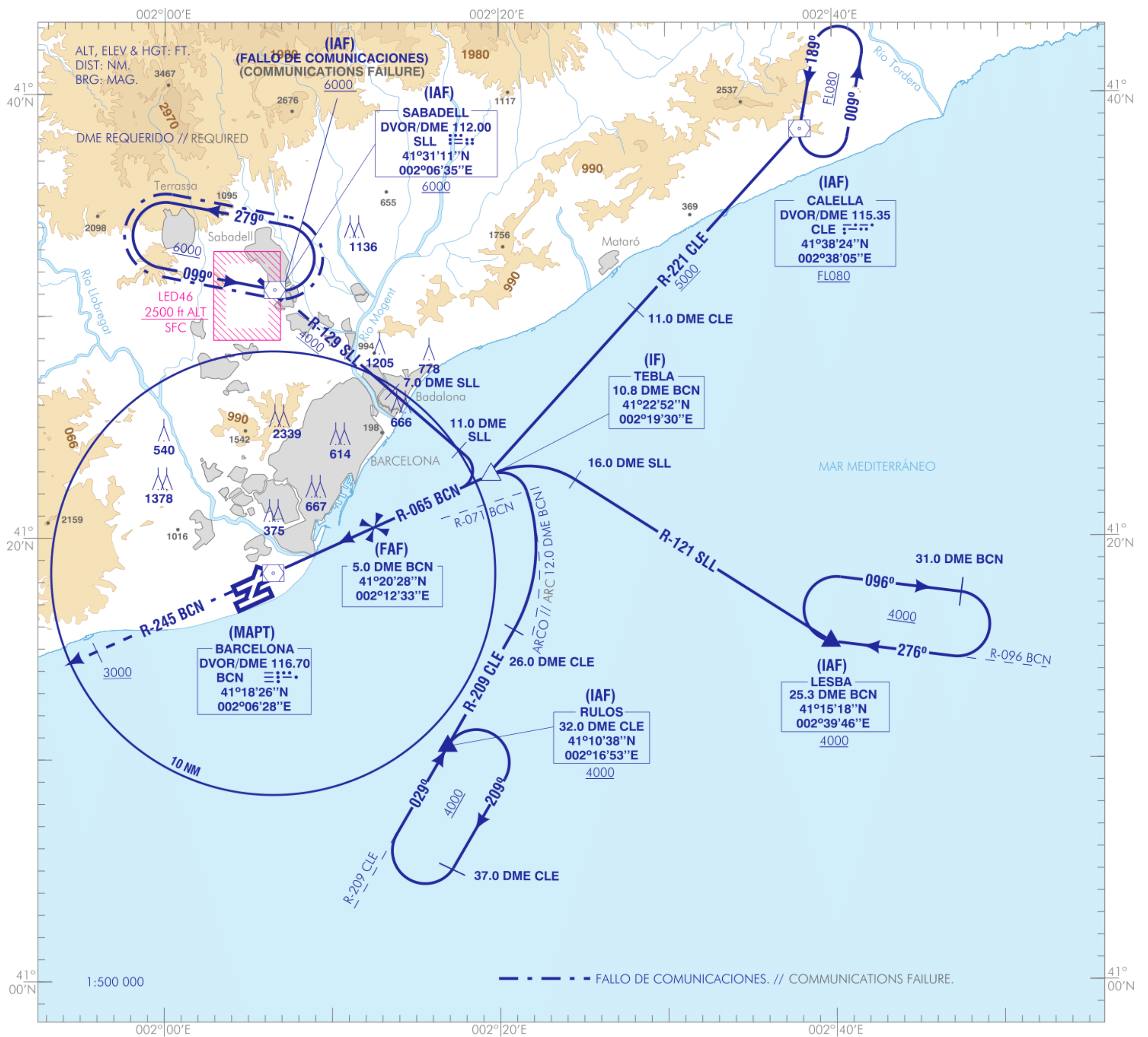

Figura 4.1: Carta IAC VOR RWY 24R [21]

De esta carta se puede obtener la información necesaria para representar el procedimiento en el programa que se desarrolla. Se saca la conclusión de que los siguientes waypoints son necesarios para representar este procedimiento en el código: el propio Aeropuerto de Barcelona, los 4 IAF, el IF y los 4 puntos marcados por distancia del DME más cercanos al IF, dado que a partir de ellos cambia la maniobra a curvas en la mayoría de los casos. El FAF no es necesario representarlo porque está en medio de una línea recta entre el IF y la pista, al igual que la distancia 7.0 DME SLL. También cumple esto la distancia 11.0 DME CLE, pero se ha decidido añadirlo como waypoint para que el resultado final sea más visualmente consistente, con un waypoint entre cada IAF y el IF. Los tramos curvos entre las distancias DME y el FAF no es posible representarlas con las clases que se han diseñado para dirigir el vuelo del avión y requeriría de un diseño complicado para implementarse sin alterar demasiado la fidelidad de la simulación, por lo

que se ha optado por representarlas por tramos rectos.

Las coordenadas de los waypoints aparecen con mayor precisión en la tabla aportada por la carta, que se muestra en la tabla 4.1.

VOR RWY 24R

PUNTO POINT	LAT	LONG	AZIMUT VERDADERO TRUE BEARING	DISTANCIA DME DME DISTANCE (NM)
LESBA (IAF)	41°15'17.7"N	002°39'45.5"E	096.93° (BCN)	25.29 DME BCN
RULOS (IAF)	41°10'38.2"N	002°16'53.3"E	210.00° (CLE)	32.00 DME CLE
DVOR/DME SLL (IAF)	41°31'11.5"N	002°06'35.1"E	–	–
DVOR/DME CLE (IAF)	41°38'24.1"N	002°38'04.8"E	–	–
TEBLA (IF)	41°22'52.2"N	002°19'30.4"E	065.57° (BCN)	10.77 DME BCN
FAF	41°20'27.8"N	002°12'32.6"E	066.00° (BCN)	5.00 DME PRA
DVOR/DME BCN (MAPT)	41°18'25.8"N	002°06'28.7"E	–	–
Aproximación final de no precisión - Pendiente (Ángulo de descenso) // Non-precision final approach - Slope (Descent angle)				5.66% (3.24°)

Cuadro 4.1: Coordenadas de los waypoints de la carta IAC VOR RWY 24R [21]

Se puede observar que las coordenadas de las distancias marcadas por DME no están disponibles, lo cual era esperado ya que el piloto utiliza el DME para saber si ha llegado. Pero en la simulación es necesario insertar sus coordenadas para dibujar el punto en el mapa, por lo que se ha implementado la clase Waypoint, de la que se hablará más adelante, de forma que le puedas dar como datos una posición y una distancia con un rumbo, de manera que sitúe el waypoint en la posición marcada por la distancia DME y el rumbo que marca la carta IAC.

Con las coordenadas obtenidas en latitud y longitud se insertan en MATLAB y con la función lla2enu que se ha explicado anteriormente se consiguen las coordenadas en ENU de cada uno de los waypoints, utilizando la posición del Aeropuerto de Barcelona como punto de referencia. Se muestra un ejemplo del código usado en la figura 4.2, en la que se ve cómo se introducen los datos de cada waypoint. Como se comentará más adelante, se insertan las posiciones marcadas por las coordenadas obtenidas en una clase que contenga las rutas que siguen los aviones desde cada IAF hasta la pista.

```
latdeg=41;
latmin=24;
latsec=50.6;
londeg=2;
lonmin=25;
lonsec=19.7;
lla = [latdeg+latmin/60+latsec/3600 londeg+lonmin/60+lonsec/3600 0]; % [lat lon alt]
lla0 = [41+18/60+26/3600 2+6/60+28/3600 0]; % [lat0 lon0 alt0]
xyzENU = lla2enu(lla,lla0,'flat');
```

Figura 4.2: Código en MATLAB utilizado para convertir de LLA a ENU

4.1.2. Procedimiento de transición a la aproximación final

En la figura 4.3 se muestra el procedimiento de transición a la aproximación final a la pista 24R del Aeropuerto de Barcelona [11].

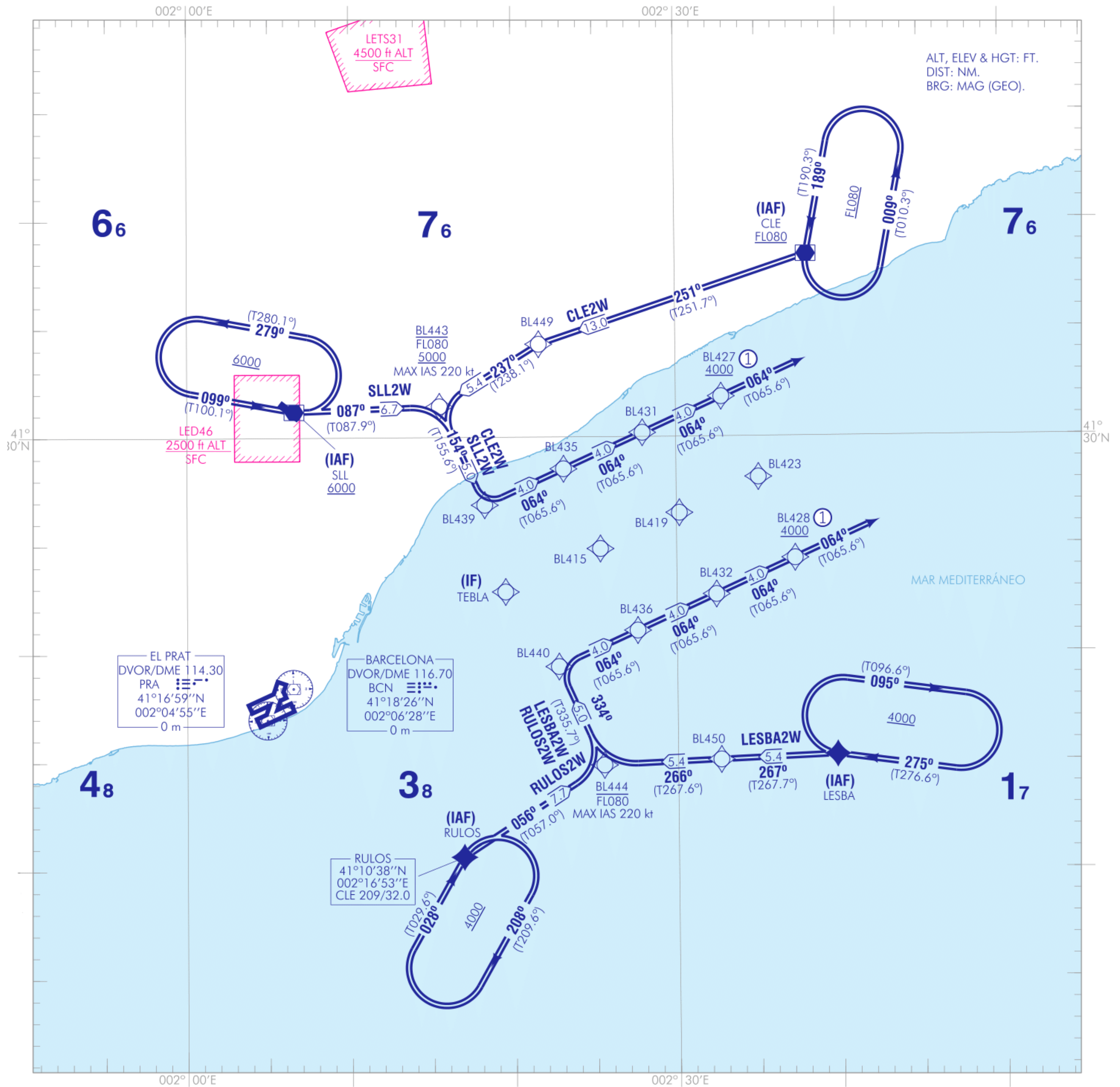

Figura 4.3: Carta IAC TRAN RWY 24R [11]

La primera diferencia destacable con el procedimiento tradicional es que no hay ninguna distancia definida por DME, por lo que resulta más sencillo insertar los waypoints en el código. La diferencia más destacable sin embargo es que el número de waypoints es mucho mayor que el del procedimiento VOR, y que todos deben ser incluidos en la simulación ya que básicamente todos son imprescindibles para poder simular la aproximación

en trombón que se intenta representar, dado que el avión debe poder girar en cualquiera de los waypoints que le indique ATC. También se nota que el FAP no está presente en la carta, ya que es de transición a la aproximación final, pero como el tramo es recto no afecta en cualquier caso.

Las coordenadas de los waypoints aparecen con mayor precisión de nuevo en la tabla aportada por la carta, que se muestra en la tabla 4.2.

COORDENADAS WAYPOINTS // WAYPOINTS COORDINATES	
WPT	COORD
BL415	41°24'50.6"N 002°25'19.7"E
BL419	41°26'29.0"N 002°30'11.0"E
BL423	41°28'07.3"N 002°35'02.7"E
BL427	41°31'50.7"N 002°32'47.7"E
BL428	41°24'22.3"N 002°37'15.8"E
BL431	41°30'11.5"N 002°27'56.8"E
BL432	41°22'43.1"N 002°32'25.5"E
BL435	41°28'32.2"N 002°23'06.0"E
BL436	41°21'03.9"N 002°27'35.2"E
BL439	41°26'53.0"N 002°18'15.3"E
BL440	41°19'24.6"N 002°22'45.1"E
BL443	41°31'26.3"N 002°15'30.4"E
BL444	41°14'51.2"N 002°25'29.1"E
BL449	41°34'17.6"N 002°21'36.8"E
BL450	41°15'04.7"N 002°32'38.2"E
CLE	41°38'24.0"N 002°38'04.2"E
LESBA	41°15'17.7"N 002°39'45.5"E
RULOS	41°10'38.2"N 002°16'53.3"E
SLL	41°31'12.0"N 002°06'35.1"E
TEBLA	41°22'52.2"N 002°19'30.4"E

Cuadro 4.2: Coordenadas de los waypoints de la carta IAC TRAN RWY 24R [11]

De nuevo, como se comentará más adelante, se introducen las posiciones marcadas por las coordenadas de la tabla en MATLAB para convertirlas a ENU y después en una clase, diferente en este caso a la de la aproximación VOR. Algo reseñable es que las coordenadas de los IAF varía en décimas de segundo de las de la aproximación VOR, pero esto no tendrá influencia en la simulación.

4.2. Capacidad del aeropuerto

Para programar el comportamiento de ATC es necesario establecer la capacidad del Aeropuerto de Barcelona. Para establecer la capacidad de pista se utilizan los datos de AECFA (Asociación Española de Coordinación de Franjas Horaria) para la temporada de 2023. En la tabla 4.3 se muestra la capacidad del Aeropuerto de Barcelona en movimientos por hora para la temporada de verano de 2023 [22]. De esta tabla se hace la media de los movimientos de llegada por hora, que da como resultado 26 llegadas por hora, lo cual equivale a una llegada cada 2.31 minutos. Para este proyecto se aproximará este número a una llegada cada 2 minutos.

CÓDIGO	AEROPUERTO	CAPACIDAD DE PISTA					
		MOVIMIENTOS / HORA			MOVIMIENTOS / SLOT		
		LLEGADA	SALIDA	TOTAL	LLEGADA	SALIDA	TOTAL
ACE	LANZAROTE CM	L-X-S			3	3	5
		12	12	22			
		M-J-V-D					
		14	14	24			
AGP	MÁLAGA-COSTA DEL SOL						
	07:00 - 18:59	25	25	46	5	5	8
	19:00 - 06:59	24	24	37	5	5	7
ALC[1]	ALICANTE-ELCHE MH						
	00:00 - 04:29	19	19	36	4	4	7
	04:30 - 05:59	19	19	36	4	6	7
	06:00 - 23:59	19	19	36	4	4	7
BCN[2]	JT BARCELONA-EL PRAT						
	00:00 - 03:59	24	24		4	4	
	04:00 - 04:59	18	30		4	6	
	05:00 - 20:59	38	40		7	8 (C)	
	21:00 - 22:59	26	22		5	4	
	23:00 - 23:59	24	24		4	4	

Cuadro 4.3: Fragmento de la tabla de la capacidad aeroportuaria para el verano de 2023 [22]

4.3. Arquitectura del software

La parte principal del diseño del proyecto viene dada por las clases y su relación entre sí. Las diversas clases se utilizan para distintas funciones. Dentro de estas clases se encuentran los métodos y atributos que permiten la simulación que se persigue.

El programa se encuentra en el proyecto **Approach** de NetBeans, que cuenta con 3 paquetes, que se muestran en la figura 4.4, y con la librería EJML para las operaciones con matrices, de la que se hablará más adelante.

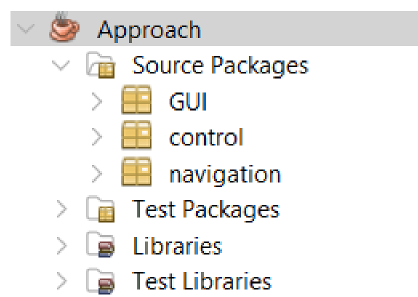

Figura 4.4: Paquetes del proyecto

Las clases del paquete **navigation** representan simulaciones de conceptos de la navegación aérea, como posiciones, rutas, aviones, waypoints, etc. y funciones que permiten que estas operen las clases de tipos de vuelo y las que contienen los waypoints de los procedimientos. Por otra parte están las que cumplen la función de ATC y las que generan el tráfico, que tienen diferentes versiones según el tipo de aproximación o el protocolo

de control deseado. Por último están las clases relacionadas con la interfaz gráfica, cuya función es realizar la simulación que aparece en pantalla.

4.3.1. Paquete navigation

El paquete **navigation**, como se ha comentado antes, es el que contiene los elementos "básicos"de una simulación de aviones volando en el espacio aéreo. En él están todas las clases necesarias para generar un vuelo que no sea representado gráficamente si se decidiese hacer una clase ejecutable para ello. En la figura 4.5 se muestra la relación de las clases entre ellas. En los siguientes apartados se explicará cada una de las clases.

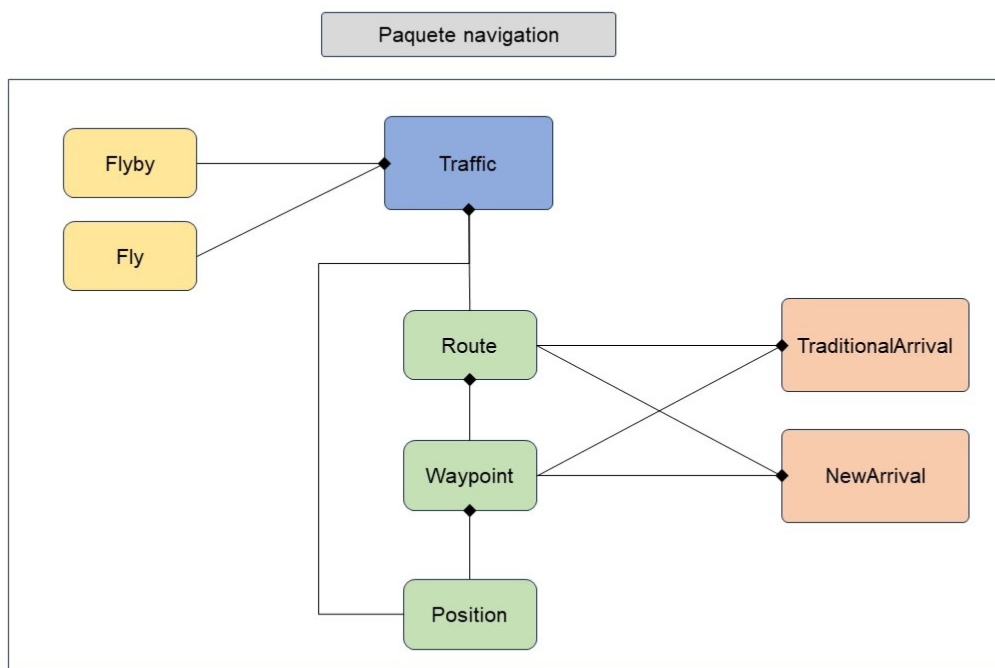

Figura 4.5: Estructura del paquete navigation

Traffic

La clase **Traffic** es la clase más importante del paquete **navigation**, ya que es la que simula a la aeronave. Extiende la clase Thread, ya que necesita ser ejecutada el paralelo al resto del programa. No tendría sentido que ATC tenga que esperar a que una aeronave llegue a la pista para dar permisos a las siguientes. Requiere una ruta o una posición para funcionar, ya que la aeronave debe encontrarse en una posición inicial, que viene dada por la clase **Position** que se le introduce, o, alternativamente, por la primera posición del primer waypoint de la **Route** que se introduzca, ya que se puede definir de ambas maneras. La aeronave también requiere una ruta que recorrer, lo que le proporciona el objeto de la clase **Route** que se le introduzca. Sin embargo, en el procedimiento en trombón la aeronave no sabe qué ruta va a recorrer hasta que ATC le dé permiso, ya que ATC calculará qué ruta es la más óptima de entre las que mantiene la separación necesaria.

En este caso las clases encargadas del control le asignará un objeto de la clase **Route** después de haber sido definido.

Traffic utiliza las clases **Flyby** y **Fly** para volvar entre dos waypoints de la ruta, definidos por objetos de la clase **Waypoint**. Utiliza un flyby cuando no es necesario sobrevolar el waypoint, y utiliza la clase **Fly** para hacer un vuelo directo entre dos waypoints.

El código de la clase **Traffic** se puede decir que está dividido en dos partes: el circuito de espera en el IAF y el vuelo hasta la pista.

- En el contexto de la simulación, las aeronaves van a ser generadas en los IAF y necesitan permiso de ATC para comenzar la aproximación, por lo que **Traffic** se quedará dando vueltas al circuito de espera del IAF hasta que se le dé permiso. En el código esto se representa de la siguiente manera: comprueba si tiene persimo, si no lo tiene estará esperando un tiempo, y al finalizar esta espera vuelve a comprobar si le han dado permiso. Si sigue sin tener permiso vuelve a repetir el proceso. El tiempo exacto que la aeronave debería estar esperando se ha ido cambiando según se ha decidido que es más realista. Si el tiempo de espera fuese 0 la aeronave empezaría la ruta en cuanto reciba permiso de ATC y el circuito de espera no estaría modelado, lo cual no es deseable. Se estima que una aeronave tardaría unos 6 minutos en realizar el circuito de espera completo, pero la aeronave puede abandonarlo dependiendo de su posición dentro del mismo en caso de haber recibido autorización. En la práctica, si se hacía esperar ciclos de 6 minutos a la aeronave los resultados eran muy poco realistas. Se ha decidido suponer que cada 3 minutos la aeronave es capaz de abandonar el circuito si ya tiene autorización, estando esta decisión únicamente basada en la calidad de los resultados obtenidos y no en estudios reales.

- Una vez que se le da permiso, la clase **Traffic** procede a volar la ruta. Para esto utiliza la clase **Fly** si el siguiente waypoint no está marcado como flyby, o la clase **Flyby** si lo está. En ambos casos estas clases le generan una serie de puntos separados entre sí según la velocidad de la aeronave, moviéndose cada segundo un punto. El código está dividido en 3 partes: primero vuela hasta el primer waypoint con o sin flyby, después hace un bucle for para el resto de waypoints menos el último y se vuela hasta la pista en vuelo recto. Esto se hizo de esta manera porque la clase **Flyby**, como se explicará más adelante, requiere 3 waypoints, y antes del bucle se utiliza el waypoint 1 de la ruta como primero y en el bucle se usa la posición actual de la aeronave.

La clase **Traffic** además de con la ruta o la posición está definida con una variable que cambia la velocidad de la simulación, una variable de tipo string con el nombre del avión que se mostrará en pantalla, una variable que cambia el radio con el que la aeronave gira al volar en flyby y una variable que indica su TAS (*true* Air Speed).

Contiene un método que permite conocer la distancia entre la posición del avión y el waypoint de la ruta que se le indique, lo que resulta útil para que ATC compare las distancias a la hora de decidir si cumple la separación.

Position

Es una clase que contiene una posición ENU x, y, z. El proyecto no tiene implementada la altura en la simulación ya que no es relevante para la secuenciación de aviones que se simula, pero tiene comentados procedimientos que sirven de base para poder ampliarlo e implementar el perfil vertical de los aviones.

Además de guardar la posición ENU, la clase **Position** cuenta con 2 métodos. El primero le permite calcular la distancia entre la posición de un objeto **Position** con otro dado y el segundo obtiene la posición de destino si se introduce una distancia y un rumbo.

Es, por tanto, una clase bastante sencilla, pero que forma la base de todas las demás clases, ya que la simulación al final se basa en la gestión de las posiciones de los aviones y los waypoints, y todas las operaciones están basadas en posiciones.

Waypoint

La clase **Waypoint** define los puntos que definen las rutas que van a seguir los aviones. Por tanto, está definida por un objeto **Position**, pero también por si se vuela flyby o no, si contiene un circuito de espera, por su nombre y por el color que se usa para representarlo.

Está pensada para poder obtener la posición mediante una distancia y un rumbo desde un objeto **Position**, lo que permite representar los puntos definidos por distancias DME.

Dado que no contiene ninguna función mas que representar la posición del waypoint, es una clase aún más sencilla que **Position**.

Route

Esta clase simplemente contiene un array de objetos de clase **Waypoint**. Es la clase con menos código por tanto, Se utiliza para definir las rutas de los aviones. Como se ha comentado antes, en la aproximación tradicional los aviones empiezan con su ruta establecida y en la de transición ATC se la asigna mientras esperan en el IAF.

Flyby

Esta clase permite obtener el recorrido que realiza el avión dados 3 waypoints en dos dimensiones (x, y), el radio de giro y la distancia entre los puntos del recorrido, siendo el segundo waypoint volado como flyby.

Es una adaptación a Java de un código de MATLAB proporcionado por el tutor del

proyecto, cuyo aspecto se muestra en la figura 4.6. Como se puede observar, la función genera un trayecto que empieza en las coordenadas del primer waypoint, hace un giro sin pasar por el segundo y acaba después de realizar el giro. Por tanto, si se quiere llegar hasta el siguiente waypoint la clase **Traffic** debe tomar la posición actual del avión como primer waypoint después de ejecutar la clase **Flyby** por primera vez, ya que si no se hiciese así no tendría un recorrido continuo. También es claro que no se puede volar solo utilizando la función flyby, ya que aun si todos los puntos fuesen flyby sería necesario llegar desde la última posición del avión en el dibujo hasta el punto final del recorrido mediante vuelo recto.

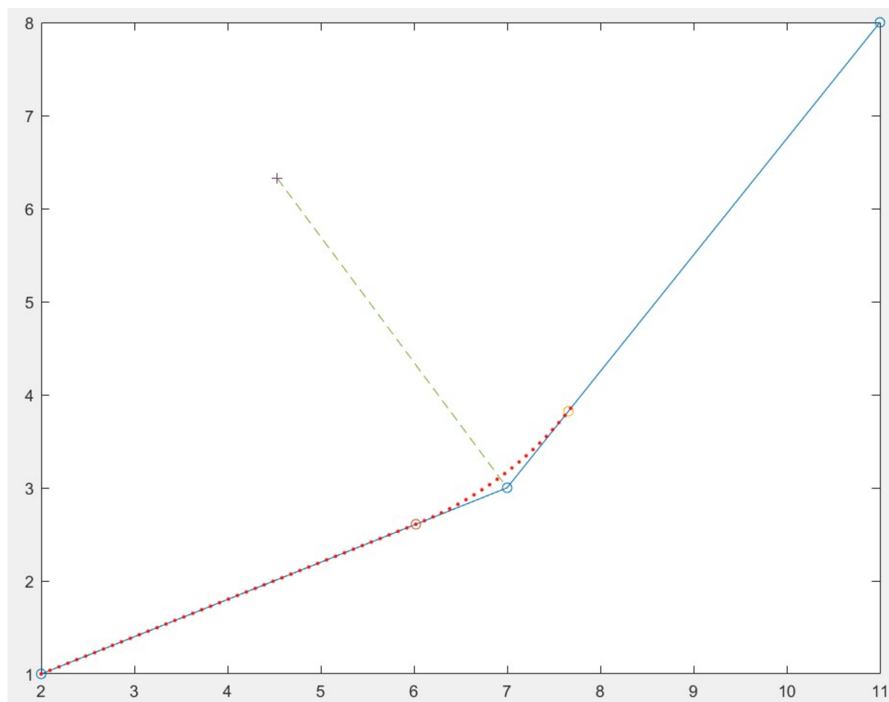

Figura 4.6: Aspecto de la función flyby en MATLAB

A la hora de traducir el código de MATLAB a Java se le ha de añadir un método linspace, ya que Java, al contrario que MATLAB, no tiene uno de forma nativa. Además, es esta clase la que hace necesario el uso de la librería EJML, ya que es necesario trabajar con matrices para las operaciones que se realizan en el código. La alternativa sería programar a mano las operaciones de las matrices, pero resulta mucho más costoso y poco intuitivo de esa manera. La librería EJML puede resultar algo complicada de entender a primera vista, pero después de probarla un tiempo resulta más cómoda que cualquier alternativa. EJML proporciona varias clases para definir las matrices, de las que se elige **SimpleMatrix** al ser la más sencilla. Una vez que se entiende cómo definir los objetos SimpleMatrix y convertir de SimpleMatrix a double[] y viceversa y cómo extraer los datos de las filas y columas de una SimpleMatrix, es posible introducir los datos en double[] y que **Flyby** los devuelva en SimpleMatrix, de donde luego se pueden extraer los datos de cada par de coordenadas x e y.

Fly

La clase **Fly** es una adaptación del código de la clase **Flyby**, a la que se le ha eliminado el código que calcula el inicio del giro y el giro en sí, y se ha convertido en una clase a la que se le dan dos waypoints y una distancia delta entre los puntos de la trayectoria que calcula una trayectoria de puntos recta entre los dos waypoints. Se muestra en la figura 4.7 el aspecto de la función fly en MATLAB, donde se puede comprobar que en efecto simplemente calcula un array de puntos entre los dos waypoints.

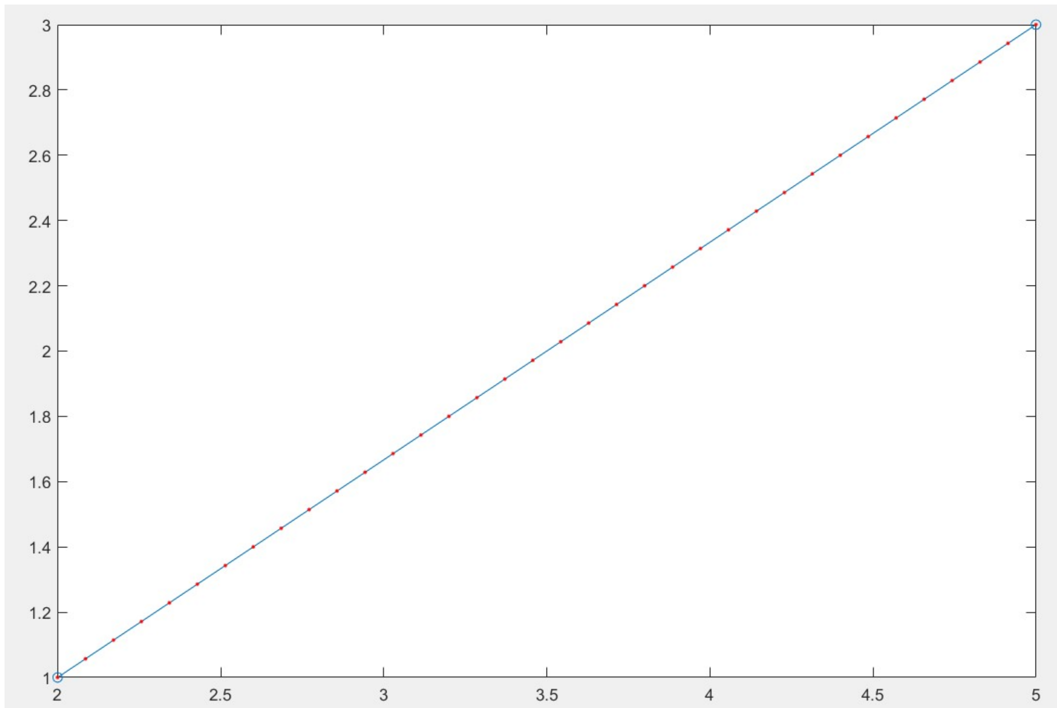

Figura 4.7: Aspecto de la función fly en MATLAB

En Java utiliza el método linspace y la clase SimpleMatrix de la misma manera que **Flyby**, ya que es una adaptación de su código. La clase **Traffic** utilizará esta clase cuando el waypoint hacia el que esté dirigiéndose no esté identificado como waypoint, y al final de la ruta siempre, ya que el trayecto del penúltimo waypoint a la pista va a ser siempre recto.

TraditionalArrival

Esta es la clase en la que se introducen las posiciones en ENU de los waypoints obtenidos de las cartas. Como estos van a ser siempre los mismos, están definidos dentro de la propia función como variables internas de valor fijo. Esta clase funciona como una base de datos, que contiene las definiciones de todos los waypoints de la aproximación VOR tradicional, definidos a partir de las posiciones en ENU, y de todas las rutas posibles que pueden tomar los aviones en esta aproximación. Se pueden crear versiones diferentes de waypoints que sea técnicamente el mismo pero que en un caso requieran que el avión los

sobrevuele y en otro puedan ser volados como flyby.

En la aproximación tradicional VOR los aviones que llegan a cada IAF conocen su ruta antes de llegar, por lo que en total hay almacenados 4 objetos de clase **Route**, uno para la ruta de cada IAF. La clase requiere un entero como dato, y devuelve la ruta que corresponda al entero. Cada tuta tiene asignado un número del 1 al 4, contando cualquier otro número como la ruta 1. La clase también permite obtener un array con todos los waypoints, para que estos puedan ser dibujados en la simulación.

NewArrival

Esta clase es la equivalente a **TraditionalArrival** pero con los datos de la aproximación en trombón. En este caso hay muchos más waypoints que en la tradicional, por lo que es necesario introducir muchas más posiciones ENU. Además, según cuanto se aleje el avión en la maniobra un mismo waypoint será volado como flyby o como sobrevuelo, ya que si gira en un waypoint hará un flyby, pero si gira en el siguiente tendrá que pasar por el anterior antes de girar. Esto significa que hay que definir varios waypoints dos veces.

En el procedimiento de trombón el piloto no sabe que ruta va a seguir cuando llega al IAF, ya que le pueden hacer alejarse más o menos antes de girar hacia la pista. Hay 4 waypoints en los que se puede girar para cada IAF, por lo que el número de rutas que se tienen que definir son 4 por cada IAF, es decir 16 objetos **Route**.

En este caso también se le introduce un entero que selecciona la ruta que se quiere recibir de la clase, pero en este caso hay 16 opciones. Los número posibles son de dos dígitos, siendo el primero del 1 al 4, indicando cada uno el IAF de la ruta, y el segundo también del 1 al 4, indicando en esta ocasión el waypoint en el que se gira. Por ejemplo, introducir un 21 devolvería el IAF 2 girando en el primer waypoint, y un 43 el IAF 4 girando en el waypoint 3. Si se introduce cualquier otro número que no sea una de las opciones devolverá la ruta 11.

4.3.2. Paquete control

Este paquete contiene las clases que simulan la actuación de ATC y las clases que generan los aviones cada cierto tiempo. El motivo por el que estos dos tipos de clases están en el mismo paquete es porque su filosofía de diseño es parecida. Esto es porque las clases que generan aeronaves con una distribución estadística se diseñan a la vez que las que contienen los protocolos de control de forma que sean compatibles pero independientes, de manera que se pueda sustituir cómo funciona cada una de ellas sin cambiar nada de las otras.

Es necesario tener al menos un protocolo de control y un generador de aviones para cada tipo de aproximación, siendo de especial importancia el protocolo de control. En los siguientes apartados se nombrará los 3 protocolos de control utilizados

Filosofía de diseño de los protocolos de control

A la hora de diseñar el método de secuenciamiento que se va a usar para cada tipo de aproximación se pueden tomar varias filosofías, teniendo cada una de ellas sus ventajas pero también sus problemas. Por ejemplo, podría parecer lo más lógico que si hay dos aviones esperando recibir permiso de ATC para iniciar la aproximación se le diese preferencia al que estuviese más cerca para optimizar la distancia entre aviones. Sin embargo, este protocolo podría llevar a una situación en la que un avión se quede esperando demasiado tiempo porque hay constantemente aviones en IAFs más cercanos, lo que naturalmente haría que se desperdiciase mucho combustible y conduciría a reclamaciones de la compañía y de los pasajeros.

Otra opción es que tengan preferencia los aviones según su orden de llegada, es decir, que si hay dos aviones esperando se le dé permiso siempre primero al que ha llegado antes al IAF. Esto tiene como inconveniente que inevitablemente va a ocasionar que un avión que esté mejor posicionado para comenzar la aproximación tenga que esperar a que otro más lejano reciba permiso, y por tanto no se aproveche la capacidad del aeropuerto y se obtenga más distancia entre aviones de la necesaria, alejándose de la óptima.

También es posible realizar un protocolo mixto, que tenga 2 waypoints en una cola y los otros 2 en otra, y dentro de esas colas se le dé permiso al que ha llegado antes. Este protocolo tiene bastante sentido en la aproximación en trombón, ya que 2 de los waypoints comparten uno de los lados de alejamiento del trombón y los otros 2 comparten el otro, por lo que tiene sentido que haya dos colas de dos waypoints cada una, una cola para cada lado del trombón. Como este esquema permite que dos aviones en lados opuestos del trombón tomen caminos más cortos o largos, pueden abandonar antes el circuito de espera, ya que la secuenciación se realiza mediante vectorización, alargando el camino si es necesario. Pero en el caso de los dos IAFs que comparten lado del trombón esto no es posible, ya que necesitan estar separados antes de llegar al trombón.

En este proyecto se ha decidido implementar 3 esquemas de control, 1 para la aproximación VOR y 2 para la de trombón.

1. Aproximación tradicional

Para la aproximación tradicional se ha implementado un sistema de cola única en el que ATC le da permiso al primero de la cola antes de comprobar si el segundo cumple la separación o no. Esto significa que ATC le da permiso primero al avión que haya llegado antes a alguno de los IAFs del procedimiento, es decir, a la cola. Se puede explicar también como que el avión que lleva más tiempo en la espera tiene preferencia para recibir autorización para realizar la aproximación. Es un protocolo cuya filosofía es evitar que ningún avión se quede esperando mucho más tiempo que el resto.

Su implementación consiste en que los aviones que llegan se meten en la cola, y ATC calcula si el primer avión de la cola cumple con los requerimientos de separación dictados por la capacidad del aeropuerto. Si el avión cumple con los requerimientos nada más llegar

puede iniciar la aproximación sin perder tiempo en el circuito de espera, pero si no los cumple tendrá que estar circulando la espera hasta volver a poder iniciar la aproximación si ahora sí cumple la separación.

2. Aproximación en trombón con cola única

Este sistema de control consiste en aplicar el esquema de cola única que utiliza la aproximación tradicional a la aproximación en trombón. Es decir, los aviones llegan y se meten en la cola, y ATC tiene preferencia para dar permiso al primero en llegar. Este sistema no tiene realmente sentido en una aproximación en trombón, ya que los aviones a cada lado del trombón se secuencian mediante vectorización, lo que hace que no sea necesario que tengan que esperar a aviones del otro lado para poder iniciar la aproximación. Aun así, se realiza este esquema de control para poder comprobar cómo de eficiente resulta.

Al igual que en el caso anterior, los aviones que llegan a los IAFs entra en una cola y ATC calcula si alguna de las posibles rutas que ofrece el trombón le permite mantener la separación con los aviones del procedimiento. Cómo se realiza la asignación de rutas se explicará más adelante en sus clases correspondientes.

3. Aproximación en trombón con dos colas

Como se ha indicado, este es el esquema más coherente para una aproximación en trombón, y es el que más sentido tiene comparar con la aproximación tradicional, ya que en la aproximación adicional hay una sola cola de aviones porque deben estar secuenciados hasta llegar al IF, que es donde se juntan los 4 caminos, pero con trombón los aviones del lado norte y sur solo necesitan esperar para estar separados hasta antes de entrar en el trombón, ya que con los de la otra cola se secuencian mediante vectorización, no mediante alargar las esperas. Como en la aproximación tradicional no se puede vectorizar, los aviones están en una sola cola en la que esperan hasta que les dan permiso.

Consiste en tener una cola norte y una cola sur, que se corresponde con los dos lados del trombón, en la que se meten a los aviones que lleguen a los respectivos IAFs. ATC le da permisos a la vez al primer avión de cada cola si se le puede asignar una ruta que cumpla la separación marcada por la capacidad. El funcionamiento de estos cálculos se explicará más adelante en la clase correspondiente.

Distribución de llegadas

Para modelizar de manera adecuada la distribución de llegadas de aviones a la aproximación es necesario implementar una distribución probabilística, dado que sería poco realista y generaría peores resultados suponer los aviones aparecen siguiendo un intervalo constante.

Para este proyecto se ha elegido un distribución de Poisson de media a elegir. Se considera que una distribución de Poisson es adecuada debido a la naturaleza de esta. Por

ejemplo, la distribución de llegadas cumple el supuesto de que la ocurrencia de un suceso no afecte a la probabilidad de que ocurra un segundo, ya que el hecho de que llegue un avión no reduce la probabilidad de que llegue otro poco después.

También se considera que la tasa media a la que se producen los sucesos es constante, ya que se quiere mantener en la media que se escoja al simular, como se ha mencionado. Otro supuesto que se cumple es que dos sucesos no pueden ocurrir exactamente en el mismo instante, ya que aunque teóricamente dos aviones pueden llegar a la vez, por la forma en la que se genera el código esto es imposible que suceda. Por último, siendo k el intervalo de tiempo en segundos entre llegadas puede tomar los valores 0, 1, 2...

Por tanto se ajusta a todos los supuestos de la distribución de Poisson, por lo que se asume como válida.

Esquema de las interacciones de clases

El esquema de las interacciones de las clases de control se muestra en la figura 4.8. Se ha diseñado de manera que sea sencillo implementar nuevos esquemas de control sin tener que cambiar ninguna de las otras clases, para que sea posible poder expandir la simulación en el futuro. La interfaz es sencilla. La clase **Traffic Control** creará un **Traffic Generator** al que le introducirá un objeto de tipo Lista llamado **Queue**. **Traffic Generator** crea objetos **Traffic** y los mete en la cola **Queue** de **Traffic Control**, que puede entonces pasar a decidir cuándo darle permiso para iniciar la aproximación utilizando el método **setPermission(*true*)**, que pone la variable permission de **Traffic** a *true*, lo que hace que cuando acabe el presente circuito de espera pueda empezar a moverse.

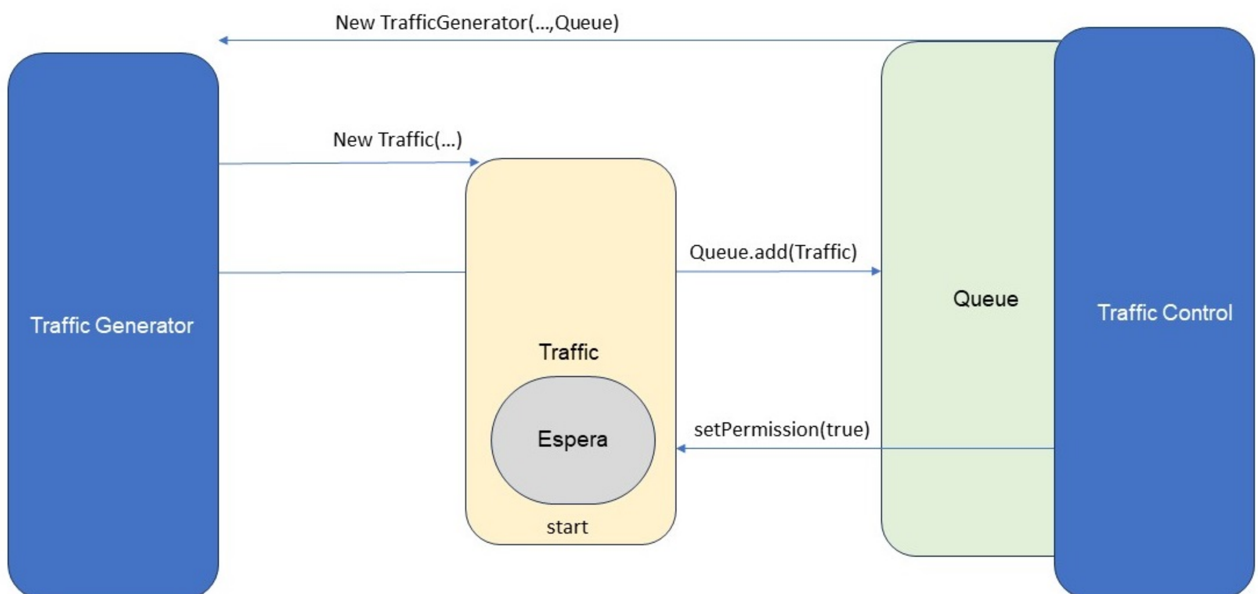

Figura 4.8: Interfaz simplificada de los protocolos de control

Se trata de un esquema reactivo, ya que la clase **Traffic** se introduce en la cola como señal de que quiere empezar la aproximación, y debe esperar a que **Traffic Control** le dé permiso para poder empezar la aproximación. No se trata de un sistema reactivo real, porque la clase **Traffic** no se pone disponible por ella misma a **Traffic Control**, sino que requiere que **Traffic Generetor** la meta en la Lista que representa la cola de ATC. Esto se diseñó de esta manera porque, aunque tiene menos sentido desde un punto de vista de realismo, es más sencillo de programar, ya que se ahorra introducir dentro de la clase **Traffic** el objeto **Queue** para que se pudiese introducir ella misma y así simular mejor que el avión anuncia su intención de iniciar la aproximación, lo que generaría una interacción extra sin beneficio para la simulación y complicaría un diseño en el que se quisiese aumentar el número de colas.

Tras explicar la filosofía de diseño general de los protocolos de control se pasará a definir las clases concretas que forma el paquete.

TrafficControl

Esta clase representa la simulación de ATC en la aproximación tradicional. Extiende la clase Thread, al ser necesario que sea ejecutada en paralelo al resto del proyecto. Utiliza **TraditionalArrival** para obtener los datos de las rutas y define **TrafficGenerator**, pasando como datos de entrada el número de aviones que se quiere que tenga la simulación y el objeto **Queue** de clase LinkedList<Traffic>, que es una Lista de objetos de clase **Traffic**.

La clase está construida respecto a dos objetos LinkedList<Traffic>, **Queue**, ya mencionado, que representa la cola de aviones esperando en los IAFs a recibir permiso para iniciar la aproximación, y **Flying**, que representa a los aviones que están volando en ese momento. **TrafficGenerator** va introduciendo los aviones en la cola al ser creados, y **TrafficControl** realiza los cálculos con el primer avión de la cola. Primero comprueba si **Queue** está vacía para que no haya errores, y pasa a verificar si la diferencia entre el tiempo que tardaría el primer avión de la cola hasta llegar al IF y el tiempo de todos los aviones que están volando (los **Traffic** contenidos en **Flying**) es superior a 2 minutos, el tiempo establecido como objetivo de la secuenciación. Esto quiere decir que la clase da permiso al primera avión de la cola si calcula que llegaría al IF con una separación de al menos 2 minutos con respecto a todos los aviones de la Lista **Flying**.

Esto es importante destacarlo, dado que es probable que en un mismo momento haya más de un avión volando la aproximación, y el avión que espera permiso debe cumplir la separación con todos ellos. Esto se implementa mediante un bucle for que recorre toda la Lista **Flying** y calcula la diferencia del tiempo de llegada al IF. La clase luego compara todas las diferencias de tiempo obtenidas y selecciona la menor. Si la menor diferencia de tiempo es superior a 2 minutos significa que el avión cumple la separación y tiene permiso para iniciar la aproximación.

Sin embargo, el avión no puede empezar la aproximación en el instante en el que recibe

permiso, ya que debe poder llegar a un punto del circuito de espera en el que pueda abandonarlo, pero puede que cuando lo haya abandonado ya no se cumpla la separación de 2 minutos de cuando le habían dado permiso. Por este motivo, **TrafficControl** comprueba constantemente que el avión cumple los requisitos de separación, y en caso de que deje de cumplirlos cambia su permiso a negativo.

Una vez que el objeto **Traffic** sale del circuito de espera con permiso positivo, pone su atributo start a *true*. **TrafficControl**, además de comprobar constantemente que el primer objeto de **Queue** cumple la separación también está comprobando si su atributo start es *true* o *false*. En caso de ser *true*, significa que el avión ha abandonado el IAF, por lo que **TrafficControl** lo quita de **Queue** y lo añade a **Flying**, escribiendo en pantalla el tiempo de separación más bajo y pasando a hacer la comprobaciones para el nuevo objeto que queda en primer lugar en la Lista **Queue.**

Además, si un **Traffic** de **Flying** llega a la pista, se lo elimina de la Lista, ya que ya ha aterrizado. Esto se hace comprobando el atributo **currentWp**, que indica en qué waypoint de la ruta se encuentra.

TrafficGenerator

La clase **TrafficGenerator** se encarga de generar un número indicado de objetos de clase **Traffic** siguiendo una distribución de Poisson para decidir el intervalo de tiempo e introducirlos en la Lista **Queue** con la que se la define. Utiliza también la clase **TraditionalArrival** para obtener los datos de las rutas de la aproximación tradicional. Trabaja de forma pareja por tanto con la clase **TrafficControl**, que es la que crea los objetos **TrafficGenerator**. Extiende la clase Thread, debido a que crea los aviones en paralelo a la gestión que ejercita **TrafficGenerator**.

Está diseñada para que cada vez que se genera un avión haya un 25 % de posibilidades de que sea generado en cada uno de los 4 IAFs.

TrafficControlNew

Es la clase equivalente a **TrafficControl** para la aproximación en trombón, utilizando el criterio de que el avión que llega primero tiene preferencia. Utiliza **NewArrival** en lugar de **TraditionalArrival** para obtener los datos, **Queue** para la cola de aeronaves en espera en los IAFs y **Flying** como Lista de aviones que están volando y extiende Thread. Crea de la misma manera un objeto **TrafficGeneratorNew**, que genera cada cierto tiempo un avión y lo asigna a un IAF.

Las diferencias surgen de que en la aproximación en trombón el piloto no conoce su ruta antes de realizar la operación, por lo que los **Traffic** son generados sin especificar rutas, utilizando la posición del IAF en su lugar. **TrafficControlNew** debe asignar la ruta más adecuada a cada avión además de darle permiso para iniciar la aproximación. La clase implementa la selección de la ruta de la siguiente manera:

- Primero comprueba en qué IAF está localizado el primer objeto de la Lista **Queue**. Esto lo realiza comparando la posición x del objeto con la posición x de cada uno de los IAF. Se podría comparar la y también, pero no es necesario ya que todas las posiciones x de los IAFs son diferentes.
- Después le asigna al objeto **Traffic** la ruta **Route** más corta de su IAF, es decir, la que gira en el primer waypoint posible en la zona del trombón.
- Con la ruta asignada, calcula el tiempo que el avión tardaría en llegar al IF y lo compara con el de todos los aviones que están volando, es decir, contenidos en **Flying**. Si la menor diferencia de tiempo es superior a 2 minutos, el avión tiene permiso para iniciar la aproximación.
- Si no cumple la diferencia de 2 minutos, se le cambia la ruta asignada a la segunda más corta, es decir, girando en el segundo waypoint. Se repite el proceso con esta nueva ruta. Si cumple con la separación recibe permiso, y si no se le asigna la siguiente ruta.
- Si se han probado todas las rutas y no obtiene permiso con ninguna se le pasa un permiso negativo y se vuelve a comprobar de nuevo. Al igual que en el case de **TrafficControl**, se comprueba constantemente para que el resultado esté actualizado cuando el avión sale del circuito de espera.

De esta manera **TrafficControlNew** le asigna rutas y concede permisos a los primeros objetos de **Queue** de manera constante, simulando la operación de ATC en la aproximación en trombón con el criterio de cola única.

TrafficControlNew2

Esta clase es una modificación de **TrafficControlNew** para implementar el protocolo de control de 2 colas, una a cada lado del trombón. Ya no se tiene solamente a la Lista **Queue**, sino que además existen las Listas **QueueN** y **QueueS**, que representa la cola norte y la cola sur respectivamente. La norte incluye a los dos waypoints que están más al norte, y la sur a los que están más al sur. Se mantiene la Lista Queue por si se quiere comprobar el orden de llegada general y para comprobar que no está vacía al principio del código.

La metodología de control es similar a la del caso de cola única, pero en este caso las comprobaciones son diferentes. El procedimiento queda de la siguiente manera:

- Se comprueba que **QueueN** no está vacía, y en caso de no estarlo se compara la posición x del primer objeto con la de los IAF de esa cola para identificar en cuál está esperando el avión.
- Se le asigna la ruta más corta que empiece en el IAF del mismo modo que antes.

- Se calcula el tiempo que tardaría un punto de la ruta que comparta con el otro IAF antes del trombón. Nótese que las rutas de ambos IAF de la misma cola siempre se van a combinar a partir de cierto punto, ya que su tramo de alejamiento es el mismo. Por este motivo se calcula la diferencia del tiempo que tardaría el avión en la cola con el que tardarán en llegar a ese mismo punto los aviones que han salido de esa cola. Se realiza además una comprobación con la situación del primer avión que está en la cola sur, es decir, con el otro que espera permiso, calculando la diferencia de tiempo hasta el IF de ambos. Si la menor diferencia de tiempo de entre todas las calculadas es superior a 2 minutos se continúa, si no, se da un permiso negativo y se vuelve a empezar el progreso.

- Si se continua con el proceso se realizan las mismas comprobaciones de **TrafficControlNew**, comprobando si la ruta más corta cumple la separación de 2 minutos hasta el IF con los aviones que están volando en ese momento. Si no la cumple se asigna la siguiente ruta más corta. Se repite el proceso hasta que una de las rutas asignadas cumpla con la separación de 2 minutos. Si ninguna cumple, se le da permiso negativo al avión.

- Se repite este proceso para **QueueS**.

Al igual que en el resto de clases ATC, extiende Thread, se van eliminando de **Flying** los aviones que han llegado al aeropuerto y cuando el avión tiene permiso se escribe en pantalla la separación más pequeña.

Esta clase es probablemente la más complicada de entender, ya que hay que realizar comprobaciones previas a las comprobaciones de las rutas, y su código es muy largo (741 líneas de código) y repetitivo (con versiones para cada IAF y cada cola de todos los procedimientos), por lo que puede resultar confuso y poco intuitivo si no se explica previamente su funcionamiento básico.

TrafficGeneratorNew

Esta clase es la versión de TrafficGenerator para la aproximación en trombón. Es igual en todos los aspectos salvo las siguientes diferencias:

- Utiliza los datos de **NewArrival**.

- Requiere que se le introduzcan **QueueN** y **QueueS**. Aunque en el caso de **TrafficControlNew** no se utilizan estas colas, como la interfaz debe ser la misma se deben introducir en la declaración igualmente.

- Asigna a todos los **Traffic** a **Queue**, pero también asigna a los que se generen en IAF norte o sur a su correspondiente cola.

4.3.3. Paquete GUI

El paquete **GUI** comprende las clases que se utilizan para la representación visual de la simulación, es decir, las relacionadas con la interfaz gráfica del proyecto. En este paquete se encuentran los ejecutables del proyecto, siendo estos 3, uno por cada protocolo de control.

En la figura 2.9 se muestra la estructura del paquete. La clase **PositionGraphic** es la base, ya que es la que permite representar las posiciones gráficas. A partir de ella se construyen las clases Plot, y utilizando esta los JFrame Panel. A partir de cada Panel se construye un Frame ejecutable.

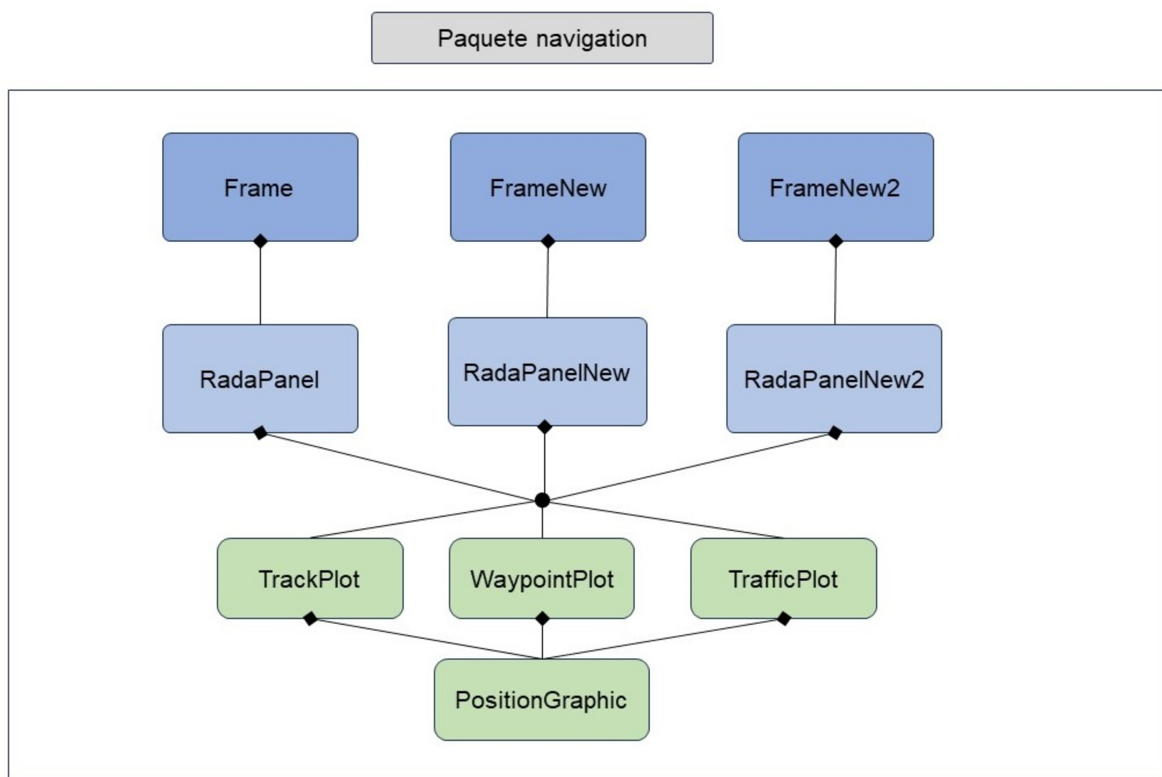

Figura 4.9: Estructura del paquete GUI

PositionGraphic

PositionGraphic define una posición gráfica (xg, yg) respecto a un rectángulo 2D. Se introduce una posición y se ajusta al marco dado por el área, la altura y la anchura de un rectángulo sobre el que se va a representar. Básicamente reescala las coordenadas respecto de los márgenes sobre los que se va a representar.

WaypointPlot

Es una clase que extiende JPanel, teniendo como variables de entrada un **Waypoint** y una versión de **RadarPanel**. Tiene 3 posibles definiciones para acomodar el planel de cada tipo de protocolo.

Utiliza los márgenes del panel que se le indica para definir una posición gráfica para el **Waypoint** introducido, dibujando sobre ella un triángulo para representar al waypoint reescribiendo paintComponent().

TrafficPlot

Es una clase similar a **WaypointPlot**, extendiendo JPanel y requiriendo como variables de entrada un **Traffic** y un **RadarPanel**, **RadarPanelNew** o **RadarPanelNew2**. Define un objeto **PositionGraphic** con la posición del **Traffic** y los márgenes del panel utilizado. Define el color, la opacidad y demás características gráficas.

En paintComponent() dibuja un óvalo en la posición del avión, y una línea que apunta a un texto con el código del avión.

TrackPlot

Representa la trayectoria recorrida por un **Traffic** en tiempo real. Tiene las mismas variables de entrada que **TrafficPlot** y también extiende JPanel.

La diferencia principal con **TrafficPlot** se da en paintComponent(), donde utiliza un bucle con un iterador de las posiciones del Log definido en **Traffic**, y para cada posición creá un objeto **PositionGraphic**. Dibuja una polilínea con todas las posiciones gráficas definidas.

RadarPanel

Es la clase que define el panel sobre el que se representa la simulación. Extiende JPanel, define un objeto **TrafficControl** y objetos **TrafficPlot**, **TrackPlot** y **WaypointPlot** para representar todos los waypoints, aviones y trayectorias que hay en la simulación en cualquier momento. Los waypoints son permanentes, pero los aviones y trayectorias cambian constantemente.

Tiene una función redundante con las clases de control, borrando de la Lista **Flying** los aviones que han acabado, para que sea más robusto el sistema.

Cuando todos los aviones han llegado al aeropuerto, es decir, cuando **TrafficControl** ha terminado su ejecución, comprueba los datos obtenidos para calcular el tiempo de espera medio y la capacidad del aeropuerto, escribiéndolos en pantalla.

RadarPanelNew y RadarPanelNew2

Funcionan exactamente igual que **RadarPanel** pero para las respectivas clases ATC.

Frame, FrameNew y FrameNew2

Son las clases ejecutables del proyecto, con un código muy básico. Extienden JFrame, y su diseño gráfico consiste en un JPanel con el código personalizado para que sea el **RadarPanel** correspondiente. Este JPanel tiene un JButton que pasa la información de los JTextField a **RadarPanel**.

Capítulo 5

Implementación

En este capítulo se mostrará una síntesis de los métodos y atributos de todas las clases del programa. Se estructura la información en diferentes tablas de manera que sea sencillo buscar información sobre los atributos, métodos o constructores de una clase determinada.

5.1. Paquete navigation

5.1.1. navigation.Traffic

public class **Traffic** extends Thread

La clase **Traffic** se encarga de la simulación de una aeronave que debe esperar a recibir permiso de **TrafficControl** para iniciar una aproximación. Utiliza una **Route** y cambia su **Position** mediante **Flyby** y **Fly** hasta llegar al destino.

Síntesis de atributos

Modificador y tipo	Atributo y descripción
private Position	**pos** Guarda la posición ENU de **Traffic**.
private double	**tas** Guarda la TAS de **Traffic**.
private double	**velSimulacion** Guarda el factor por el que se acelera la simulación.
private Color	**color** Guarda el color con el que se representa **Traffic**.
private double	**traveled** Guarda la distancia recorrida por **Traffic**.
private Route	**route** Guarda la ruta asignada a **Traffic**.

private Waypoint[]	**wp** Guarda el conjunto de **Waypoints** de la ruta asignada.
private double	**delta** Guarda la distancia que se introduce en **Flyby** y **Fly** entre los puntos de la trayectoria.
private double	**R** Guarda el radio de giro que se introduce en **Flyby**.
private double	**espera** Guarda el tiempo que **Traffic** se mantiene en el circuito de espera.
private int	**currentWp** Guarda el último **Waypoint** que ha atravesado **Traffic**.
private boolean	**start** Indica si **Traffic** ha comenzado a volar la ruta.
private boolean	**permission** Indica si **Traffic** ha recibido permiso de **TrafficControl** o clases equivalentes para comenzar a volar.
private String	**code** Guarda el nombre de **Traffic**.
private String	**status** Guarda el estado de **Traffic**, pudiendo ser *Flying* o *In circuit.*
private boolean	**selected** Indica si **Traffic** está seleccionado (volando) para las clases gráficas.

Síntesis de constructores

Constructor y descripción
public Traffic(Route route, double tas, double R, double velSimulacion, String code) Construye un Thread que simula un avión y le otorga los atributos identificados en los argumentos del constructor. El avión tiene la ruta definida desde el principio.
public Traffic(Position pos, double tas, double R, double velSimulacion, String code) Construye un Thread que simula un avión y le otorga los atributos identificados en los argumentos del constructor. El avión necesita que ATC le asigne la ruta.

Síntesis de métodos

Modificador y tipo	Método y descripción
private void	**run()** Ejecuta el código principal de la clase.
public double	**getDistance(int destWp)** Calcula la distancia entre la posición actual y el waypoint indicado.
public LinkedList<Position>	**getLog()** Devuelve una Lista con todas las posiciones de la trayectoria recorrida hasta ese punto.
public void	**setLog(LinkedList** *t* **<Position> log)** Cambia el log por la Lista de posiciones indicada.
public void	**addLog()** Añade la posición actual del avión al log de posiciones.
public Position	**getPosition()** Devuelve la posición actual del avión.
public void	**setPosition(Position pos)** Cambia la posición del avión por la indicada.
public double	**getX()** Devuelve la posición X actual del avión.
public double	**getY()** Devuelve la posición Y actual del avión.
public double	**getZ()** Devuelve la posición Z actual del avión.
public double	**getVel()** Devuelve la velocidad TAS en m/s del avión.
public double	**getTas()** Devuelve la velocidad TAS en nudos del avión.
public void	**setTas(double tas)** Cambia la velocidad TAS en nudos del avión por la indicada.
public Color	**getColor()** Devuelve el color asignado a **Traffic**.
public void	**setColor(Color color)** Cambia el color de **Traffic** por el indicado.
public boolean	**getSelected()** Devuelve si **Traffic** está seleccionado.
public void	**setSelected(boolean selected)** Cambia el estado de selected al indicado.

public double	**getTraveled()** Devuelve la distancia recorrida por el avión.
public int	**getCurrentWp()** Devuelve el número del último waypoint que el avión ha atravesado.
public Route	**getRoute()** Devuelve la ruta que **Traffic** tiene asignada.
public void	**setRoute(Route route)** Cambia la ruta que **Traffic** tiene asignada por la indicada.
public void	**setPermission(boolean permission)** Cambia el estado del permiso de ATC para iniciar el procedimiento por el indicado.
public boolean	**isStart()** Indica si el avión ha comenzado la aproximación.
public String	**getCode()** Devuelve el nombre que **Traffic** tiene asignado.
public String	**getStatus()** Devuelve el estado de **Traffic**. Puede ser *In citcuit* o *Flying*.
public void	**setStatus(String status)** Cambia el status de **Traffic** por el indicado.
public double	**getEspera()** Devuelve el tiempo de espera total de **Traffic**.
public boolean	**isPermission()** Indica si **Traffic** tiene permiso para iniciar la aproximación.

5.1.2. navigation.Waypoint

public class **Waypoint**

La clase **Waypoint** define uno de los puntos de una **Route** (ruta) que marcan la trayectoria que debe seguir **Traffic**. Puede ser volado en flyby.

Síntesis de atributos

Modificador y tipo	Atributo y descripción
public Position	**pos** Guarda la posición ENU del **Waypoint**.
public String	**name** Guarda el nombre del **Waypoint**.
public Color	**color** Guarda el color con el que se representa al **Waypoint**.
public int	**size** Guarda el tamaño de representación del **Waypoint**.
public boolean	**flyby** Indica si el **Waypoint** se vuela como flyby.
private String	**icon** Guarda el icono con el que se representan **Waypoints** pequeños. Es el utilizado en el proyecto.
private String	**icon2** Guarda el icono con el que se representan **Waypoints** grandes.
double	**espera** Indica si el **Waypoint** no inicial tiene una espera (no utilizado).

Síntesis de constructores

Constructor y descripción
public Waypoint(Position pos, String name, Color color, int size, **boolean flyby)** Construye un waypoint al que otorga los atributos identificados en los argumentos del constructor, guardando su posición y sus elementos gráficos.
public Waypoint(Position pos, String name, Color color, int size,) **boolean flyby, double espera)** Construye un waypoint al que otorga los atributos identificados en los argumentos del constructor, guardando su posición y sus elementos gráficos y otorgándole una espera si no es el inicial (no utilizado).
public Waypoint(Position pos, String name, Color color, int size,) **boolean flyby, boolean flyby)** Construye un waypoint al que otorga los atributos identificados en los argumentos del constructor, guardando su posición y sus elementos gráficos e indicando que se vuela como flyby..

Síntesis de métodos

Modificador y tipo	Método y descripción
public String	**getIcon()** Devuelve el icono que corresponde al tamaño del **Waypoint**.
public Position	**getPos()** Devuelve la posición ENU del **Waypoint**.
public boolean	**isFlyby()** Indica si el **Waypoint** se vuela como flyby.
public double	**getEspera()** Indica si el **Waypoint** (no inicial) tiene una espera. No utilizado.

5.1.3. navigation.Route

public class **Route**

La clase **Route** define un conjunto de **Waypoints** que debe seguir un **Traffic** hasta llegar al aeropuerto de destino.

Síntesis de atributos

Modificador y tipo	Atributo y descripción
Waypoint[]	**wp** Guarda el conjunto de waypoints de la ruta.

Síntesis de constructores

Constructor y descripción
public Route(Waypoint[] wp) Construye una ruta con el conjunto de waypoints indicado.

Síntesis de métodos

Modificador y tipo	Método y descripción
public Waypoint[]	**getWp()** Devuelve el array de **Waypoints** qu forman la ruta.

5.1.4. navigation.Position

public class **Position**

La clase **Position** define una posición en coordenadas ENU.

Síntesis de atributos

Modificador y tipo	Atributo y descripción
private double	**x** Guarda la coordenada x de la posición ENU.
private double	**y** Guarda la coordenada y de la posición ENU.
private double	**z** Guarda la coordenada z de la posición ENU.

Síntesis de constructores

Constructor y descripción
public Position(double x, double y, double z) Construye una posición con las coordenadas ENU proporcionadas en los argumentos.

Síntesis de métodos

Modificador y tipo	Método y descripción
public doube	**getDistance(Position location)** Devuelve la distancia entre la posición de este objeto y la posición indicada.
public Position	**getDestination(double distance, double bearing)** Devuelve la posición que se encuentra a la distancia y el rumbo indicados.
public double	**getX()** Devuelve la coordenada x de la posición ENU.
public void	**setX(double x)** Cambia la coordenada x de la posición ENU por la indicada.
public double	**getY()** Devuelve la coordenada y de la posición ENU.
public void	**setY(double y)** Cambia la coordenada y de la posición ENU por la indicada.
public double	**getZ()** Devuelve la coordenada z de la posición ENU.
public void	**setZ(double z)** Cambia la coordenada z de la posición ENU por la indicada.

5.1.5. navigation.Fly

public class **Fly**

La clase **Fly** define una matriz de puntos que representan la trayectoria recta entre dos waypoints. La utiliza **Traffic** para tramos rectos sin flyby.

Síntesis de atributos

Modificador y tipo	Atributo y descripción
private double[]	**w1** Guarda la posición ENU 2D del primer waypoint.
private double[]	**w2** Guarda la posición ENU 2D del segundo waypoint.
private double	**delta** Guarda la distancia deseada entre puntos de la trayectoria.
private SimpleMatrix	**p** Guarda el array de puntos de la trayectoria generada por la clase.

Síntesis de constructores

Constructor y descripción
public Fly(double[] w1, double[] w2, double delta) Calcula un array de posiciones separadas delta metros de la trayectoria entre dos puntos.

Síntesis de métodos

Modificador y tipo	Método y descripción
public SimpleMatrix	**fly()** Calcula un array de posiciones separadas delta metros de la trayectoria entre dos puntos..
public static double[]	**linspace(double init, double end, int n)** Crea un double[] de tamaño n con los límites establecidos.
public SimpleMatrix	**getP()** Devuelve la matriz p de las posiciones calculadas.

5.1.6. navigation.Flyby

public class **Flyby**

La clase **Flyby** define una matriz de puntos que representan la trayectoria que un avión realiza cuando un waypoint se vuela como flyby. La utiliza **Traffic** para tramos rectos con flyby.

Síntesis de atributos

Modificador y tipo	Atributo y descripción
private double[]	**w1** Guarda la posición ENU 2D del primer waypoint.
private double[]	**w2** Guarda la posición ENU 2D del segundo waypoint.
private double[]	**w2** Guarda la posición ENU 2D del tercer waypoint.
private double	**R** Guarda el radio de giro objetivo de la maniobra.
private double	**delta** Guarda la distancia deseada entre puntos de la trayectoria.
private SimpleMatrix	**p** Guarda el array de puntos de la trayectoria generada por la clase.

Síntesis de constructores

Constructor y descripción
public Flyby(double[] w1, double[] w2, double[] w3, double R, double delta) Calcula un array de posiciones separadas delta metros de una maniobra flyby entre tres puntos.

Síntesis de métodos

Modificador y tipo	Método y descripción
public SimpleMatrix	**fly()** Calcula un array de posiciones de la maniobra flyby.
public static double[]	**linspace(double init, double end, int n)** Crea un double[] de tamaño n con los límites establecidos.
public SimpleMatrix	**getP()** Devuelve la matriz p de las posiciones calculadas.

5.1.7. navigation.TraditionalArrival

public class **TraditionalArrival**

La clase **TraditionalArrival** sirve como base de datos que contiene los **Waypoints** y **Routes** de una maniobra de aproximación tradicional VOR al Aeropuerto de Barcelona. Otras clases la utilizan para obtener los datos de las rutas.

Síntesis de atributos

Modificador y tipo	Atributo y descripción
private Route	**route** Guarda la ruta seleccionada según los argumentos del constructor.
private Waypoint[]	**wp** Guarda el array de **Waypoints** de la ruta seleccionada.
private Waypoint[]	**WPtotal** Guarda el array de todos los **Waypoints** del procedimiento.

Síntesis de constructores

Constructor y descripción
public TraditionalArrival(double RouteSelector) Guarda los datos de todas las posiciones, waypoints y rutas de la aproximación tradicional VOR. Selecciona los datos de una de las rutas según el valor del argumento.

Síntesis de métodos

Modificador y tipo	Método y descripción
public Route	**getRoute()** Devuelve la ruta seleccionada.
public Waypoint[]	**getWp()** Devuelve el array de **Waypoints** de la ruta seleccionada.
public Waypoint[]	**getWPtotal()** Devuelve el array de **Waypoints** del procedimiento.

5.1.8. navigation.NewArrival

public class **NewArrival**

La clase **NewArrival** sirve como base de datos que contiene los **Waypoints** y **Routes** de una maniobra de aproximación en trombón al Aeropuerto de Barcelona. Otras clases la utilizan para obtener los datos de las rutas.

Síntesis de atributos

Modificador y tipo	Atributo y descripción
private Route	**route** Guarda la ruta seleccionada según los argumentos del constructor.
private Waypoint[]	**wp** Guarda el array de **Waypoints** de la ruta seleccionada.
private Waypoint[]	**WPtotal** Guarda el array de todos los **Waypoints** del procedimiento.

Síntesis de constructores

Constructor y descripción
public TraditionalArrival(double RouteSelector) Guarda los datos de todas las posiciones, waypoints y rutas de la aproximación en trombón. Selecciona los datos de una de las rutas según el valor del argumento.

Síntesis de métodos

Modificador y tipo	Método y descripción
public Route	**getRoute()** Devuelve la ruta seleccionada.
public Waypoint[]	**getWp()** Devuelve el array de **Waypoints** de la ruta seleccionada.
public Waypoint[]	**getWPtotal()** Devuelve el array de **Waypoints** del procedimiento.

5.2. Paquete control

5.2.1. control.TrafficControl

public class **TrafficControl** extends Thread

La clase **TrafficControl** simula el comportamiento de ATC en una aproximación tradicional VOR. Secuencia los **Traffic** que son creados para que mantengan una distancia que cumpla con la capacidad del aeropuerto.

Síntesis de atributos

Modificador y tipo	Atributo y descripción
private LinkedList<Traffic>	**Queue** Guarda la Lista de **Traffics** que esperan permiso para iniciar la aproximación.
private LinkedList<Traffic>	**Flying** Guarda la Lista de **Traffics** que están haciendo la aproximación.
private boolean	**running** Indica si **TrafficControl** ha sido iniciado. Mientras sea *true* se mantiene ejecutando.
private boolean	**finnish** Indica si **TrafficControl** ha finalizado.
private Traffic[]	**List** Guarda el array de todos los **Traffic** cuando acaba la ejecución.
private double	**rate** Guarda la media de la tiempo de llegada entre aviones.
private double	**velSim** Guarda la velocidad de la simulación respecto al tiempo real.
private int	**X** Indica el número de **Traffics** que deben ser creados

Síntesis de constructores

Constructor y descripción
public TrafficControl(int X, double rate, double VelSim) Crea un Thread que implementa un protocolo de control para la secuenciación de **Traffics** en la aproximación VOR. Los argumentos definen las características de la simulación.

Síntesis de métodos

Modificador y tipo	Método y descripción
ppublic void	**run()** Ejecuta el código principal del Thread.
public LinkedList<Traffic>	**getQueue()** Devuelve la Lista de **Traffics** que están en el circuito de espera.
public LinkedList<Traffic>	**getFlying()** Devuelve la Lista de **Traffics** que están volando la aproximación.
public Traffic[]	**getList()** Devuelve el array de **Traffics** que han sido controlados.
public boolean	**isFinnish()** Indica si el Thread ha terminado su ejecución.
public double	**getRate()** Devuelve el tiempo de separación medio entre de **Traffics**.
public void	**getVelSim()** Devuelve el valor de la velocidad de la simulación.
public void	**getX()** Devuelve el número de **Traffics** simulados.

5.2.2. control.TrafficControlNew

public class **TrafficControlNew** extends Thread

La clase **TrafficControlNew** simula el comportamiento de ATC en una aproximación en tormbón con cola única. Secuencia los **Traffic** que son creados para que mantengan una distancia que cumpla con la capacidad del aeropuerto.

Síntesis de atributos

Modificador y tipo	Atributo y descripción
private LinkedList<Traffic>	**Queue** Guarda la Lista de **Traffics** que esperan permiso para iniciar la aproximación.
private LinkedList<Traffic>	**QueueS** Guarda la Lista de **Traffics** que esperan permiso para iniciar la aproximación en la cola sur.
private LinkedList<Traffic>	**QueueN** Guarda la Lista de **Traffics** que esperan permiso para iniciar la aproximación en la cola norte.
private LinkedList<Traffic>	**Flying** Guarda la Lista de **Traffics** que están haciendo la aproximación.
private boolean	**running** Indica si **TrafficControl** ha sido iniciado. Mientras sea *true* se mantiene ejecutando.
private boolean	**finnish** Indica si **TrafficControl** ha finalizado.
private Traffic[]	**List** Guarda el array de todos los **Traffic** cuando acaba la ejecución.
private double	**rate** Guarda la media de la tiempo de llegada entre aviones.
private double	**velSim** Guarda la velocidad de la simulación respecto al tiempo real.
private int	**X** Indica el número de **Traffics** que deben ser creados

Síntesis de constructores

Constructor y descripción
public TrafficControlNew(int X, double rate, double VelSim) Crea un Thread que implementa un protocolo de control para la secuenciación de **Traffics** en la aproximación de trombón con cola única. Los argumentos definen las características de la simulación.

Síntesis de métodos

Modificador y tipo	Método y descripción
ppublic void	**run()** Ejecuta el código principal del Thread.
public LinkedList<Traffic>	**getQueue()** Devuelve la Lista de **Traffics** que están en el circuito de espera.
public LinkedList<Traffic>	**getFlying()** Devuelve la Lista de **Traffics** que están volando la aproximación.
public Traffic[]	**getList()** Devuelve el array de **Traffics** que han sido controlados.
public boolean	**isFinnish()** Indica si el Thread ha terminado su ejecución.
public double	**getRate()** Devuelve el tiempo de separación medio entre de **Traffics**.

5.2.3. control.TrafficControlNew2

public class **TrafficControlNew2** extends Thread

La clase **TrafficControlNew2** simula el comportamiento de ATC en una aproximación en tormbón con dos colas. Secuencia los **Traffic** que son creados para que mantengan una distancia que cumpla con la capacidad del aeropuerto.

Síntesis de atributos

Modificador y tipo	Atributo y descripción
private LinkedList<Traffic>	**Queue** Guarda la Lista de **Traffics** que esperan permiso para iniciar la aproximación.
private LinkedList<Traffic>	**QueueS** Guarda la Lista de **Traffics** que esperan permiso para iniciar la aproximación en la cola sur.
private LinkedList<Traffic>	**QueueN** Guarda la Lista de **Traffics** que esperan permiso para iniciar la aproximación en la cola norte.
private LinkedList<Traffic>	**Flying** Guarda la Lista de **Traffics** que están haciendo la aproximación.
private boolean	**running** Indica si **TrafficControl** ha sido iniciado. Mientras sea *true* se mantiene ejecutando.
private boolean	**finnish** Indica si **TrafficControl** ha finalizado.
private Traffic[]	**List** Guarda el array de todos los **Traffic** cuando acaba la ejecución.
private double **private double**	**rate** Guarda la media de la tiempo de llegada entre aviones. **velSim** Guarda la velocidad de la simulación respecto al tiempo real.
private int	**X** Indica el número de **Traffics** que deben ser creados

Síntesis de constructores

Constructor y descripción
public TrafficControlNew2(int X, double rate, double VelSim) Crea un Thread que implementa un protocolo de control para la secuenciación de **Traffics** en la aproximación de trombón con dos colas. Los argumentos definen las características de la simulación.

Síntesis de métodos

Modificador y tipo	Método y descripción
ppublic void	**run()** Ejecuta el código principal del Thread.
public LinkedList<Traffic>	**getQueue()** Devuelve la Lista de **Traffics** que están en el circuito de espera.
public LinkedList<Traffic>	**getFlying()** Devuelve la Lista de **Traffics** que están volando la aproximación.
public Traffic[]	**getList()** Devuelve el array de **Traffics** que han sido controlados.
public boolean	**isFinnish()** Indica si el Thread ha terminado su ejecución.
public double	**getRate()** Devuelve el tiempo de separación medio entre de **Traffics**.

5.2.4. control.TrafficGenerator

public class **TrafficGenerator** extends Thread

La clase **TrafficGenerator** crea **Traffic** cada cierto intervalo de tiempo que sigue una distribución de Poisson para la aproximación VOR.

Síntesis de atributos

Modificador y tipo	Atributo y descripción
private LinkedList<Traffic>	**Queue** Guarda la Lista de **Traffics** que esperan permiso para iniciar la aproximación.
private int	**X** Indica el número de **Traffics** que deben ser creados
private boolean	**finnish** Indica si **TrafficControl** ha finalizado.
private Traffic[]	**tg** Guarda el array de todos los **Traffic** generados.
private double	**rate** Guarda la media de la tiempo de llegada entre aviones.
private double	**velSim** Guarda la velocidad de la simulación respecto al tiempo real.

Síntesis de constructores

Constructor y descripción
public TrafficGenerator(int X, LinkedList<Traffic>Queue) Crea un Thread que crea X aviones separados un tiempo decidido por una distribución de Poisson y los guarda en la Lista **Queue**.

Síntesis de métodos

Modificador y tipo	Método y descripción
ppublic void	**run()** Ejecuta el código principal del Thread.
public Traffic[]	**getTg()** Devuelve el array de **Traffics** que han sido generaods.
public boolean	**isFinnish()** Indica si el Thread ha terminado su ejecución.
public double	**getRate()** Devuelve el tiempo de separación medio entre de **Traffics**.
public double	**getVelSim()** Devuelve la velocidad de la simulación.

5.2.5. control.TrafficGeneratorNew

public class **TrafficGeneratorNew** extends Thread

La clase **TrafficGeneratorNew** crea **Traffic** cada cierto intervalo de tiempo que sigue una distribución de Poisson para la aproximación en trombón.

Síntesis de atributos

Modificador y tipo	Atributo y descripción
private LinkedList<Traffic>	**Queue** Guarda la Lista de **Traffics** que esperan permiso para iniciar la aproximación.
private LinkedList<Traffic>	**QueueS** Guarda la Lista de **Traffics** que esperan permiso para iniciar la aproximación en la cola sur.
private LinkedList<Traffic>	**QueueN** Guarda la Lista de **Traffics** que esperan permiso para iniciar la aproximación en la cola norte.
private int	**X** Indica el número de **Traffics** que deben ser creados
private boolean	**finnish** Indica si **TrafficControl** ha finalizado.
private Traffic[]	**tg** Guarda el array de todos los **Traffic** generados.
private double	**rate** Guarda la media de la tiempo de llegada entre aviones.
private double	**velSim** Guarda la velocidad de la simulación respecto al tiempo real.

Síntesis de constructores

Constructor y descripción
public TrafficGeneratorNew(int X, LinkedList<Traffic> Queue, LinkedList<Traffic> QueueN,LinkedList<Traffic> QueueS) Crea un Thread que crea X aviones separados un tiempo decidido por una distribución de Poisson y los guarda en la Lista **Queue**, en la Lista **QueueS** si son de los IAFs sur y en la Lista **QueueN** si son de los IAFs norte.

Síntesis de métodos

Modificador y tipo	Método y descripción
ppublic void	**run()** Ejecuta el código principal del Thread.
public Traffic[]	**getTg()** Devuelve el array de **Traffics** que han sido generaods.
public boolean	**isFinnish()** Indica si el Thread ha terminado su ejecución.
public double	**getRate()** Devuelve el tiempo de separación medio entre de **Traffics**.
public double	**getVelSim()** Devuelve la velocidad de la simulación.

5.3. Paquete GUI

5.3.1. GUI.WaypointPlot

public class **WaypointPlot** extends JPanel

La clase **WaypointPlot** define la representación gráfica de un **Waypoint**.

Síntesis de atributos

Modificador y tipo	Atributo y descripción
private Waypoint	**wp** Guarda el **Waypoint** que se quiere representar.
private RadarPanel	**rp** Guarda el **RadarPanel** en el que se quiere representar el **Waypoint**.
private RadarPanelNew	**rp1** Guarda el **RadarPanelNew** en el que se quiere representar el **Waypoint**.
private RadarPanelNew2	**rp2** Guarda el **RadarPanelNew2** en el que se quiere representar el **Waypoint**.
private Rectangle2D	**rp_area** Guarda el área rectangular 2D del panel guardado.
private int	**rp_width** Guarda la anchura del panel guardado.
private int	**rp_height** Guarda la altura del panel guardado.
private Dimension	**preferredSize** Guarda el tamaño preferido del componente.
private int	**fontSize** Guarda el tamaño de letra con el que se escribe en pantalla.

Síntesis de constructores

Constructor y descripción
public WaypointPlot(Waypoint wp, RadarPanel rp) Crea una representación gráfica de un **Waypoint** en un **RadarPanel** determinado.
public WaypointPlot(Waypoint wp, RadarPanelNew rp1) Crea una representación gráfica de un **Waypoint** en un **RadarPanelNew** determinado.
public WaypointPlot(Waypoint wp, RadarPanelNew2 rp2) Crea una representación gráfica de un **Waypoint** en un **RadarPanelNew2** determinado.

Síntesis de métodos

Modificador y tipo	Método y descripción
public void	**setPreferredSize(Dimension preferredSize)** Cambia el tamaño preferido por el indicado.
public Dimension	**getPreferredSize()** Devuelve el tamaño preferido.
public void	**paintComponent(Graphics g)** Dibuja en pantalla con los comandos establecidos.

5.3.2. GUI.TrafficPlot

public class **TrafficPlot** extends JPanel

La clase **TrafficPlot** define la representación gráfica de un **Traffic**.

Síntesis de atributos

Modificador y tipo	Atributo y descripción
private Traffic	**tg** Guarda el **Traffic** que se quiere representar.
private RadarPanel	**rp** Guarda el **RadarPanel** en el que se quiere representar el **Traffic**.
private RadarPanelNew	**rp1** Guarda el **RadarPanelNew** en el que se quiere representar el **Traffic**.
private RadarPanelNew2	**rp2** Guarda el **RadarPanelNew2** en el que se quiere representar el **Traffic**.
private Rectangle2D	**rp_area** Guarda el área rectangular 2D del panel guardado.
private int	**rp_width** Guarda la anchura del panel guardado.
private int	**rp_height** Guarda la altura del panel guardado.
private Dimension	**preferredSize** Guarda el tamaño preferido del componente.
private int	**RADIUS** Guarda el radio del óvalo que representa al avión.

Síntesis de constructores

Constructor y descripción
public TrafficPlot(Traffic tg, RadarPanel rp) Crea una representación gráfica de un **Traffic** en un **RadarPanel** determinado.
public TrafficPlot(Traffic tg, RadarPanelNew rp1) Crea una representación gráfica de un **Traffic** en un **RadarPanelNew** determinado.
public TrafficPlot(Traffic tg, RadarPanelNew2 rp2) Crea una representación gráfica de un **Traffic** en un **RadarPanelNew2** determinado.

Síntesis de métodos

Modificador y tipo	Método y descripción
public void	**setPreferredSize(Dimension preferredSize)** Cambia el tamaño preferido por el indicado.
public Dimension	**getPreferredSize()** Devuelve el tamaño preferido.
public void	**paintComponent(Graphics g)** Dibuja en pantalla con los comandos establecidos.

5.3.3. GUI.TrackPlot

public class **TrackPlot** extends JPanel

La clase **TrackPlot** define la representación gráfica de la trayectoria de un **Traffic**.

Síntesis de atributos

Modificador y tipo	Atributo y descripción
private Traffic	**tg** Guarda el **Traffic** del cual se quiere representar la trayectoria.
private RadarPanel	**rp** Guarda el **RadarPanel** en el que se quiere representar la trayectoria.
private RadarPanelNew	**rp1** Guarda el **RadarPanelNew** en el que se quiere representar la trayectoria.
private RadarPanelNew2	**rp2** Guarda el **RadarPanelNew2** en el que se quiere representar la trayectoria.
private Rectangle2D	**rp_area** Guarda el área rectangular 2D del panel guardado.
private int	**rp_width** Guarda la anchura del panel guardado.
private int	**rp_height** Guarda la altura del panel guardado.
private Dimension	**preferredSize** Guarda el tamaño preferido del componente.

Síntesis de constructores

Constructor y descripción
public TrackPlot(Traffic tg, RadarPanel rp) Crea una representación gráfica de la trayectoria de un **Traffic** en un **RadarPanel** determinado.
public TrackPlot(Traffic tg, RadarPanelNew rp1) Crea una representación gráfica de la trayectoria de un **Traffic** en un **RadarPanelNew** determinado.
public TrackPlot(Traffic tg, RadarPanelNew2 rp2) Crea una representación gráfica de la trayectoria de un **Traffic** en un **RadarPanelNew2** determinado.

Síntesis de métodos

Modificador y tipo	Método y descripción
public void	**setPreferredSize(Dimension preferredSize)** Cambia el tamaño preferido por el indicado.
public Dimension	**getPreferredSize()** Devuelve el tamaño preferido.
public void	**paintComponent(Graphics g)** Dibuja en pantalla con los comandos establecidos.

5.3.4. GUI.RadarPanel

public class **RadarPanel** extends JPanel implements Runnable

La clase **RadarPanel** define un panel en el que se representa gráficamente la simulación de la aproximación VOR. En ella se crean un **TrafficControl** y las clases gráficas de los elementos simulados.

Síntesis de atributos

Modificador y tipo	Atributo y descripción
private double	**xWest** Guarda la posicón x del margen oeste del panel.
private double	**xEast** Guarda la posicón x del margen este del panel.
private double	**yNorth** Guarda la posicón y del margen norte del panel.
private double	**ySud** Guarda la posicón y del margen sur del panel.
long	**tiempo** Guarda el tiempo del reloj interno que tarda la simulación.
long	**tiempo2** Guarda el tiempo basado en ciclos del código que tarda la simulación.
private Traffic[]	**tg** Guarda el array de **Traffics** que son representados en cada momento.
private TraditionalArrival	**traditional1, traditional2, traditional13, traditional4** Guardan los diferentes tipos de rutas tradicionales de los aviones.
private TrafficControl	**TFC** Genera un objeto **TrafficControl** para que secuencie el tráfico.
private double	**rate** Guarda la media de la tiempo de llegada entre aviones.
private double	**velSim** Guarda la velocidad de la simulación respecto al tiempo real.
private int	**X** Indica el número de **Traffics** que deben ser creados
private boolean	**start** Indica si la simulación debe empezar.

Síntesis de constructores

Constructor y descripción
public RadarPanel() Construye una JPanel que crea y ejecuta un objeto **TrafficControl** y representa con las clases gráficas los elementos de la simulación en pantalla.

Síntesis de métodos

Modificador y tipo	Método y descripción
public void	**run()** Ejecuta el código principal de la clase.
public void	**updateRadarPanel()** Actualiza constantemente la representación gráfica en pantalla.
public Rectangle2D	**getArea()** Devuelve el área del mapa.
public void	**paintComponent()** Llama a Graphics2D para pintar en pantalla.
public void	**setStart()** Cambia el estado de start. Se usa para empezar la simulación.
public void	**setVelSim()** Cambia el valor de la velocidad de la simulación.
public void	**setX()** Cambia el número de **Traffics** simulados.
public void	**setRate()** Cambia el tiempo entre llegadas de nuevos **Traffics**.

5.3.5. GUI.RadarPanelNew

public class **RadarPanelNew** extends JPanel implements Runnable

La clase **RadarPanelNew** define un panel en el que se representa gráficamente la simulación de la aproximación en trombón con cola única. En ella se crean un **TrafficControlNew** y las clases gráficas de los elementos simulados.

Síntesis de atributos

Modificador y tipo	Atributo y descripción
private double	**xWest** Guarda la posicón x del margen oeste del panel.
private double	**xEast** Guarda la posicón x del margen este del panel.
private double	**yNorth** Guarda la posicón y del margen norte del panel.
private double	**ySud** Guarda la posicón y del margen sur del panel.
long	**tiempo** Guarda el tiempo del reloj interno que tarda la simulación.
long	**tiempo2** Guarda el tiempo basado en ciclos del código que tarda la simulación.
private Traffic[]	**tg** Guarda el array de **Traffics** que son representados en cada momento.
private NewArrival	**new11, new12, new13, new14, new21, new22, new24, new23, new31, new32, new33, new34, new41, new42, new43, new44** Guardan los diferentes tipos de rutas en trombón de los aviones.
private TrafficControlNew	**TFC** Genera un objeto **TrafficControlNew** para que secuencie el tráfico.
private double	**rate** Guarda la media de la tiempo de llegada entre aviones.
private double	**velSim** Guarda la velocidad de la simulación respecto al tiempo real.
private int	**X** Indica el número de **Traffics** que deben ser creados
private boolean	**start** Indica si la simulación debe empezar.

Síntesis de constructores

Constructor y descripción
public RadarPanelNew() Construye una JPanel que crea y ejecuta un objeto **TrafficControlNew** y representa con las clases gráficas los elementos de la simulación en pantalla.

Síntesis de métodos

Modificador y tipo	Método y descripción
public void	**run()** Ejecuta el código principal de la clase.
public void	**updateRadarPanel()** Actualiza constantemente la representación gráfica en pantalla.
public Rectangle2D	**getArea()** Devuelve el área del mapa.
public void	**paintComponent()** Llama a Graphics2D para pintar en pantalla.
public void	**setStart()** Cambia el estado de start. Se usa para empezar la simulación.
public void	**setVelSim()** Cambia el valor de la velocidad de la simulación.
public void	**setX()** Cambia el número de **Traffics** simulados.
public void	**setRate()** Cambia el tiempo entre llegadas de nuevos **Traffics**.

5.3.6. GUI.RadarPanelNew2

public class **RadarPanelNew2** extends JPanel implements Runnable

La clase **RadarPanelNew2** define un panel en el que se representa gráficamente la simulación de la aproximación en trombón con dos colas. En ella se crean un **TrafficControlNew2** y las clases gráficas de los elementos simulados.

Síntesis de atributos

Modificador y tipo	Atributo y descripción
private double	**xWest** Guarda la posicón x del margen oeste del panel.
private double	**xEast** Guarda la posicón x del margen este del panel.
private double	**yNorth** Guarda la posicón y del margen norte del panel.
private double	**ySud** Guarda la posicón y del margen sur del panel.
long	**tiempo** Guarda el tiempo del reloj interno que tarda la simulación.
long	**tiempo2** Guarda el tiempo basado en ciclos del código que tarda la simulación.
private Traffic[]	**tg** Guarda el array de **Traffics** que son representados en cada momento.
private NewArrival	**new11, new12, new13, new14, new21, new22, new24, new23, new31, new32, new33, new34, new41, new42, new43, new44** Guardan los diferentes tipos de rutas en trombón de los aviones.
private TrafficControlNew2	**TFC** Genera un objeto **TrafficControlNew2** para que secuencie el tráfico.
private double	**rate** Guarda la media de la tiempo de llegada entre aviones.
private double	**velSim** Guarda la velocidad de la simulación respecto al tiempo real.
private int	**X** Indica el número de **Traffics** que deben ser creados
private boolean	**start** Indica si la simulación debe empezar.

Síntesis de constructores

Constructor y descripción
public RadarPanelNew2() Construye una JPanel que crea y ejecuta un objeto **TrafficControlNew2** y representa con las clases gráficas los elementos de la simulación en pantalla.

Síntesis de métodos

Modificador y tipo	Método y descripción
public void	**run()** Ejecuta el código principal de la clase.
public void	**updateRadarPanel()** Actualiza constantemente la representación gráfica en pantalla.
public Rectangle2D	**getArea()** Devuelve el área del mapa.
public void	**paintComponent()** Llama a Graphics2D para pintar en pantalla.
public void	**setStart()** Cambia el estado de start. Se usa para empezar la simulación.
public void	**setVelSim()** Cambia el valor de la velocidad de la simulación.
public void	**setX()** Cambia el número de **Traffics** simulados.
public void	**setRate()** Cambia el tiempo entre llegadas de nuevos **Traffics**.

5.3.7. GUI.Frame, GUI.FrameNew y GUI.FrameNew2

Estas clases solo son JFrame que contienen un JPanel con el código de la versión correspondiente de **RadarPanel** y varios JTextField y un JButton para asignar los valores a la simulación, por lo que no aporta mucho valor incluirlas en el API.

Capítulo 6

Manual de usuario

La interfaz gráfica de usuario para este proyecto es extremadamente sencilla e intuitiva, cumpliendo con lo marcado en los objetivos del proyecto. Cuenta con 3 ejecutables, que se muestran en la figura 6.1.

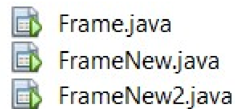

Figura 6.1: Ejecutables del proyecto

Para comenzar la simulación se debe iniciar uno de estos ejecutables, según el tipo de aproximación que se desee. Para una aproximación VOR tradicional se utiliza la clase **Frame**, para una aproximación en trombón con protocolo de ATC de cola única se utiliza **FrameNew** y para una aproximación en trombón con protocolo de ATC de dos colas se utiliza **FrameNew2**.

Una vez que se ejecuta una de estas clases se abre un panel con el aspecto que se observa en la figura 6.2. En este panel se debe establecer el número de aviones que se quiere simular, la aceleración respecto al tiempo real y la media de la separación temporal entre las entradas de los aviones a la aproximación.

Figura 6.2: Aspecto del panel de la simulación

Cuando los valores sean los deseados se hace clic sobre Start y la simulación comenzará. A partir de este momento la simulación se ejecuta y ATC secuencia los aviones independientemente. Se muestra en la figura 6.3 el aspecto de la simulación de la aproximación en trombón en funcionamiento en pantalla completa. En la figura 6.4 se muestra lo mismo pero para la aproximación tradicionak.

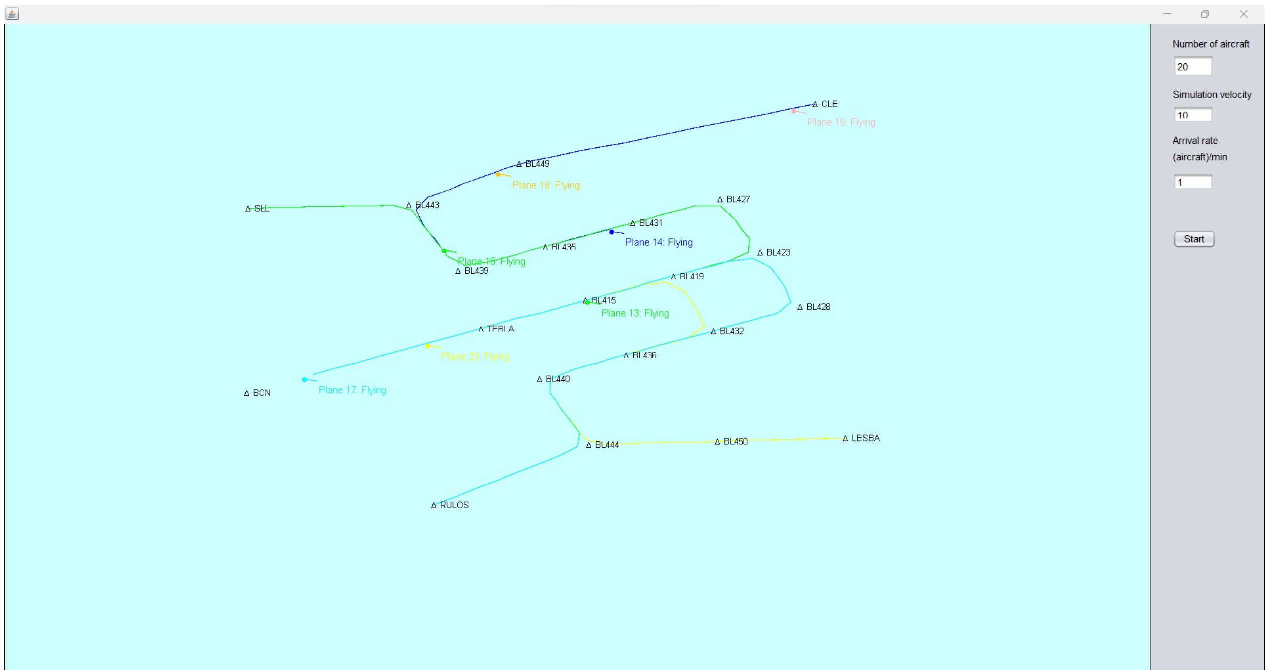

Figura 6.3: Aspecto de la aproximación en trombón en funcionamiento en pantalla completa

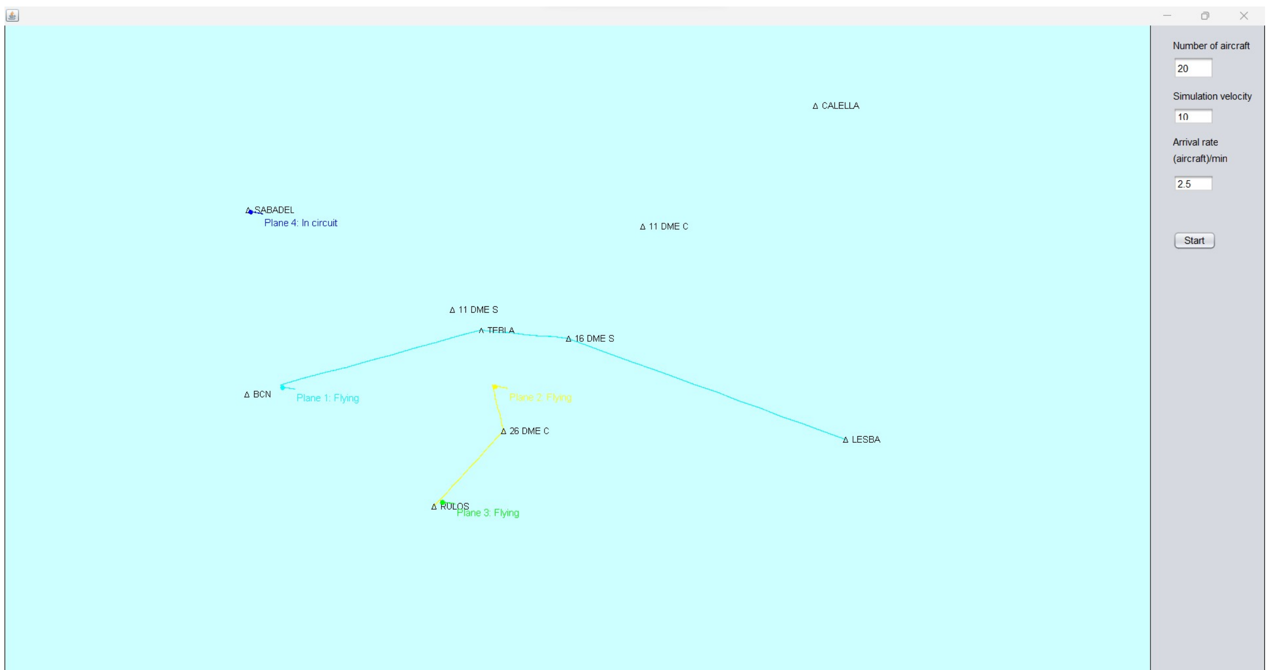

Figura 6.4: Aspecto de la aproximación tradicional en funcionamiento en pantalla completa

En el cuadro de outputs se van escribiendo las separaciones temporales de cada avión que inicia la aproximación con los otros aviones que están en vuelo. Cuando la simulación acaba se escriben el tiempo total, la espera media, la utilización de la capacidad y la la

utilización de la capacidad normalizada. En la figura 6.5 se muestra un ejemplo de los datos del cuadro de outputs.

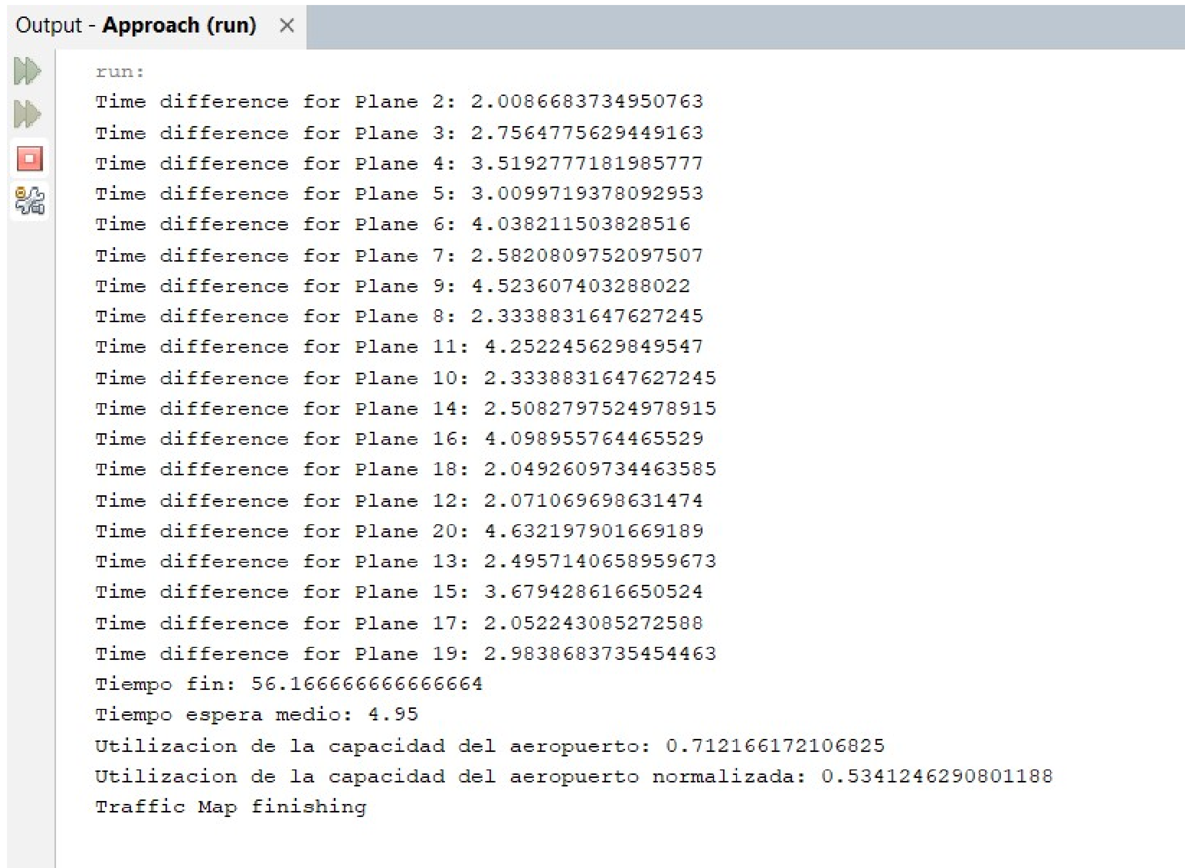

Figura 6.5: Aspecto del cuadro de outputs

Capítulo 7

Experimentos

En este capítulo se realizarán diferentes experimentos para probar la simulación que se ha desarrollado y poder obtener conclusiones respecto de los tipos de aproximaciones y protocolos de control utilizados.

7.1. Caracterización de la carga

Para los experimentos realizados se han 20 aviones y velocidad de 10, para que las diferencias vengan dadas solamente por la eficiencia de ATC para cada intervalo de tiempo de llegada. El motivo por el que se ha elegido esta velocidad es que velocidades más bajas tardan mucho tiempo en ejecutarse y permite realizar menos experimentos en el mismo tiempo, pero velocidades más elevadas pierden algo de precisión al hacer que los puntos de la trayectoria estén más separados. El número de aviones se ha elegido para que sea lo suficientemente grande pero sin que lleve a una simulación demasiado larga.

La velocidad de los aviones se toma como 180 nudos y es constante en toda la simulación, ya que es más sencillo de programar y no afecta a las conclusiones extraídas sobre los resultados de la simulación. La capacidad del aeropuerto, como se comentó en el apartado 4.2, se considera como de 0.5 aviones por minutos, o lo que es lo mismo, un avión cada 2 minutos.

El parámetro más importante es la separación entre aviones, ya que si el intervalo es grande ATC no va a tener problemas para secuenciar a los aviones, pero si el intervalo se va reduciendo le irá costando más secuenciar, y los aviones se pueden ir acumulando en los circuitos de espera.

El objetivo de los experimentos es comprobar la eficacia de los distintos esquemas en situaciones de tráfico diferentes, por lo que se realizarán 4 experimentos variando La media del intervalo de llegada. El intervalo de llegada sigue una distribución de Poisson como se ha explicado en el subapartado **Distribución de llegadas** del apartado 4.3.2, caracterizada por la media introducida en cada simulación. Los experimentos realizados tendrán separaciones de 1.5, 2, 1.2, 2.5 y 3.5 minutos, con las que se pretende comprobar situaciones con tráfico superior a la capacidad, igual a la capacidad, extremadamente superior a la capacidad, inferior a la capacidad y muy poco tráfico respectivamente.

7.2. Experimento 1: separación de 1.5 minutos

7.2.1. Resultados

Para este experimento se realizan varias simulaciones con 1.5 minutos de media de separación entre llegadas de aviones a la aproximación. En la tabla 7.1 se muestra la media de los resultados de las múltiples simulaciones realizadas para la aproximación tradicional, en la tabla 7.2 para la aproximación en trombón con cola única y en la tabla 7.3 para la aproximación en trombón con dos colas.

Métrica	**Valor**
Separación media entre aviones	3.68 min
Tiempo total de la operación	80.10 min
Tiempo medio de los aviones en la espera	20.96 min
Utilización de la capacidad de la pista	50 %
Utilización normalizada de la capacidad de la pista	38 %

Cuadro 7.1: Media de los resultados del experimento 1 para aproximación VOR tradicional

Métrica	**Valor**
Separación media entre aviones	3.50 min
Tiempo total de la operación	70.44 min
Tiempo medio de los aviones en la espera	14.50 min
Utilización de la capacidad de la pista	57 %
Utilización normalizada de la capacidad de la pista	43 %

Cuadro 7.2: Media de los resultados del experimento 1 para aproximación en trombón con cola única

Métrica	**Valor**
Separación media entre aviones	2.94 min
Tiempo total de la operación	59.72 min
Tiempo medio de los aviones en la espera	6.00 min
Utilización de la capacidad de la pista	67 %
Utilización normalizada de la capacidad de la pista	50 %

Cuadro 7.3: Media de los resultados del experimento 1 para aproximación en trombón con dos colas

7.2.2. Análisis de resultados

Esta situación representa un nivel de tráfico superior al que la capacidad del aeropuerto (estimada como de 2 min por avión) puede soportar, por lo que la secuenciación de ATC es muy necesaria. Dado que el intervalo medio establecido entre llegadas de aviones es un 25 % inferior a la capacidad, los resultados representan una situación de tráfico más intenso de lo normal, lo que hace bastante interesante este análisis.

Lo primero que llama la atención es que la separación media entre aviones no se consigue acercar a mucho 2 minutos en niguno de los casos. El que más cerca está es el esquema de trombón con dos colas, con 2.94 minutos de separación entre aviones. Esto lo separa de los otros dos procedimientos, que tienen ambos valores superiores a 3.5 minutos, con el tradicional teniendo la separación más elevada.

Los tiempos de finalización son superiores en aproximadamente 10 minutos al anterior, con de nuevo el esquema tradicional siendo el peor y el de trombón con dos colas el mejor. El tiempo de espera medio de los aviones es muy alto en el esquema tradicional, mejorando en el trombón de una cola pero siendo aún deficiente. El único con un tiempo medio de espera más razonable es el trombón de dos colas, que tiene 6 minutos de espera, lo que equivaldría a hacer el circuito de espera completo una vez. En la utilización y la utilización normalizada los esquemas siguen el mismo orden en cuanto a eficacia, y de nuevo los resultados del tradicional y del trombón de una cola están más cerca entre ellos que los dos trombones entre ellos, con el esquema de dos colas siendo claramente superior y el único que consigue una utilización normalizada del 50 %.

El tiempo de espera medio se comprueba no tiene una traducción proporcional con la mejor utilización de la capacidad del aeropuerto y con el tiempo total. Esto se debe a la naturaleza del esquema en trombón, que reduce mucho los tiempos de espera al secuenciar parte de los aviones con vectorización. Los aviones necesitan pasar menos tiempo en espera pero tienen que realizar una ruta más larga hasta la pista. Sin embargo se demuestra que es rentable el intercambio, ya que se consigue una mejor eficacia en todos los parámetros para los esquemas en trombón, y que siempre es mejor utilizar el esquema con dos colas, que no tiene ninguna desventaja con respecto al de una cola. No se puede decir que el de dos colas le de más prioridad a aviones que han llegado más tarde, ya que las dos colas se autorizan a la vez y si hay que secuenciar entre aviones de ambas colas se realiza mediante vectorización.

7.3. Experimento 2: separación de 2 minutos

7.3.1. Resultados

Para este experimento se realizan varias simulaciones con 2 minutos de media de separación entre llegadas de aviones a la aproximación. En la tabla 7.4 se muestra la media de los resultados de las múltiples simulaciones realizadas para la aproximación

tradicional, en la tabla 7.5 para la aproximación en trombón con cola única y en la tabla 7.6 para la aproximación en trombón con dos colas.

Métrica	**Valor**
Separación media entre aviones	3.19 min
Tiempo total de la operación	62.50 min
Tiempo medio de los aviones en la espera	10.20 min
Utilización de la capacidad de la pista	53 %
Utilización normalizada de la capacidad de la pista	53 %

Cuadro 7.4: Media de los resultados del experimento 2 para aproximación VOR tradicional

Métrica	**Valor**
Separación media entre aviones	2.84 min
Tiempo total de la operación	70.89 min
Tiempo medio de los aviones en la espera	9.90 min
Utilización de la capacidad de la pista	57 %
Utilización normalizada de la capacidad de la pista	57 %

Cuadro 7.5: Media de los resultados del experimento 2 para aproximación en trombón con cola única

Métrica	**Valor**
Separación media entre aviones	2.94 min
Tiempo total de la operación	62.44 min
Tiempo medio de los aviones en la espera	2.90 min
Utilización de la capacidad de la pista	64 %
Utilización normalizada de la capacidad de la pista	64 %

Cuadro 7.6: Media de los resultados del experimento 2 para aproximación en trombón con dos colas

7.3.2. Análisis de resultados

Los resultados de esta sección son muy interesantes. El primer dato que se puede extraer comparando estos resultados con los de media de 1.5 minutos es que el único esquema que mejora sus resultados en utilización y tiempo total de la operación es el tradicional. Esto se debe principalmente a que en este esquema las acumulaciones en la cola de las esperas al tener un ritmo mayor de llegada son el mayor problema de un esquema con espera como método de secuenciación. En el esquema de trombón parte de la secuenciación se realiza con vectorización, por lo que el ritmo mayor de llegadas puede

permitir una utilización mayor de la pista si se dan condiciones adecuadas de tráfico. Esto no es posible sin la vectorización, dado que llegadas más frecuentes que la capacidad del aeropuerto hacen que los aviones esperando en la espera se acumulen.

El único esquema que no mejora la separación media al aumentar el intervalo de llegada de aviones es el trombón de dos colas, que se mantiene igual. Esto es porque este tipo de secuenciación tiene la capacidad de absorber el aumento en tráfico sin aumentar la separación entre aviones, al utilizar la vectorización para separar gran parte del tráfico. Es notable que la única mejora al reducir la frecuencia de aviones es la reducción del tiempo en la espera, lo que no mejora en sí la eficiencia, porque el tiempo de finalización es incluso algo mayor que antes. Esto significa que los aviones están menos tiempo en la espera pero hacen recorridos más largos en el trombón. Este fenómeno también ocurre en el trombón de una cola, ya que disminuye el tiempo de espera medio pero esto no causa una disminución del tiempo total.

La utilización normalizada es igual que la utilización debido a que se normaliza respecto a 2 minutos de capacidad. Como el intervalo de llegada es de 2 minutos (de media de Poisson), se anulan al dividir. Aun así, se nota que aumenta, lo que significa que el procedimiento está mejorando su eficiencia al absorber el tráfico. Aunque los datos son muy similares, es debido a que los aviones en el experimento 1 se están acumulando en las colas de espera, y para los datos absolutos acaba siendo más importante el tiempo que ATC tarda en sacar un avión de la cola que el tiempo que tardan en llegar los aviones. El parámetro más importante para diferenciarlos es entonces la utilización normalizada, ya que es el que mejor refleja la eficiencia.

7.4. Experimento 3: separación de 1.2 minutos

7.4.1. Resultados

Para este experimento se realizan varias simulaciones con 1.2 minutos de media de separación entre llegadas de aviones a la aproximación. En la tabla 7.7 se muestra la media de los resultados de las múltiples simulaciones realizadas para la aproximación tradicional, en la tabla 7.8 para la aproximación en trombón con cola única y en la tabla 7.9 para la aproximación en trombón con dos colas.

Métrica	**Valor**
Separación media entre aviones	3.50 min
Tiempo total de la operación	77.67 min
Tiempo medio de los aviones en la espera	22.95 min
Utilización de la capacidad de la pista	52 %
Utilización normalizada de la capacidad de la pista	31 %

Cuadro 7.7: Media de los resultados del experimento 3 para aproximación VOR tradicional

Métrica	Valor
Separación media entre aviones	3.19 min
Tiempo total de la operación	62.50 min
Tiempo medio de los aviones en la espera	10.20 min
Utilización de la capacidad de la pista	64 %
Utilización normalizada de la capacidad de la pista	38 %

Cuadro 7.8: Media de los resultados del experimento 3 para aproximación en trombón con cola única

Métrica	Valor
Separación media entre aviones	2.84 min
Tiempo total de la operación	58.50 min
Tiempo medio de los aviones en la espera	6.00 min
Utilización de la capacidad de la pista	68 %
Utilización normalizada de la capacidad de la pista	41 %

Cuadro 7.9: Media de los resultados del experimento 3 para aproximación en trombón con dos colas

7.4.2. Análisis de resultados

En este experimento se considera una situación de tráfico extremadamente intenso, llegando los aviones con un intervalo medio de 1.2 minutos. Se pueden extraer conclusiones para situaciones en las que el tráfico es mucho más frecuente de lo que permite la capacidad del aeropuerto.

La aproximación tradicional tiene unos resultados bastante similares a los del experimento 1. Sin embargo, hay que tener en cuenta que los aviones están llegando mucho más rápido, por lo que el hecho de que la simulación tarde más o menos lo mismo significa que la eficacia es mucho menor. Se puede observar en el hecho de que la espera media es superior. Básicamente se crea una situación en la que todos los aviones automáticamente se van acumulando en los circuitos de espera, mientras ATC da una autorización a un ritmo inferior, lo que hace que haya cada vez más aviones en los circuitos de espera en una situación no sostenible si se prolongase. Si se aumentase el número de aviones simulados se comprobaría de forma más clara esta acumulación.

Comparando la utilización normalizada con los otros casos se nota que aunque el resto de datos sugieren buena eficiencia ninguno de los esquemas llega al 50 %, por lo que en realidad los aviones se están acumulando en la cola y los datos parecen positivos porque no se está teniendo en cuenta el ritmo de llegada. El ritmo de aterrizajes parece bueno, pero esto es porque ATC secuencia a los aviones de las colas para que lleguen bien al aeropuerto. Se nota al observar que el tiempo de finalización es casi igual a los anteriores.

7.5. Experimento 4: separación de 2.5 minutos

7.5.1. Resultados

Para este experimento se realizan varias simulaciones con 2.5 minutos de media de separación entre llegadas de aviones a la aproximación. En la tabla 7.10 se muestra la media de los resultados de las múltiples simulaciones realizadas para la aproximación tradicional, en la tabla 7.11 para la aproximación en trombón con cola única y en la tabla 7.12 para la aproximación en trombón con dos colas.

Métrica	**Valor**
Separación media entre aviones	3.71 min
Tiempo total de la operación	76.50 min
Tiempo medio de los aviones en la espera	12.60 min
Utilización de la capacidad de la pista	52 %
Utilización normalizada de la capacidad de la pista	65 %

Cuadro 7.10: Media de los resultados del experimento 4 para aproximación VOR tradicional

Métrica	**Valor**
Separación media entre aviones	3.53 min
Tiempo total de la operación	73.83 min
Tiempo medio de los aviones en la espera	6.75 min
Utilización de la capacidad de la pista	54 %
Utilización normalizada de la capacidad de la pista	68 %

Cuadro 7.11: Media de los resultados del experimento 4 para aproximación en trombón con cola única

Métrica	**Valor**
Separación media entre aviones	3.21 min
Tiempo total de la operación	66.67 min
Tiempo medio de los aviones en la espera	1.50 min
Utilización de la capacidad de la pista	60 %
Utilización normalizada de la capacidad de la pista	75 %

Cuadro 7.12: Media de los resultados del experimento 4 para aproximación en trombón con dos colas

7.5.2. Análisis de resultados

En este experimento se prueba con un intervalo de llegada superior a la capacidad del aeropuerto, para comprobar cómo funcionan los diferentes esquemas en condiciones normales.

En primer lugar, es relevante que los datos de la aproximación tradicional y en trombón de cola única son muy similares en todos los parámetros. Esto se explica porque al llegar con más margen los aviones no se acumulan en la cola, y por tanto la aproximación tradicional es capaz de absorber de manera adecuada el tráfico. Sigue siendo mejor, sin embargo, el esquema en trombón de dos colas. Aun con ambos esquemas funcionando sin presión por alto tráfico, el trombón de dos colas aprovecha la capacidad de secuenciar mediante vectorización y puede autorizar a dos aviones a la vez, separándolos mediante caminos más largos o cortos. El esquema de una cola, como se notó anteriormente, no es adecuado para la aproximación vectorizada, debido a que no es capaz de aprovechar el potencial de esta.

También cabe destacar que en este experimento la utilización normalizada es superior a la absoluta. Esto se debe a que el motivo por el que no se está utilizando tanto la capacidad del aeropuerto es que los aviones simplemente están tardando más en llegar y por tanto no es necesario utilizar toda la capacidad, no se debe a la incapacidad del procedimiento de absorber el tráfico.

7.6. Experimento 5: separación de 3.5 minutos

7.6.1. Resultados

Para este experimento se realizan varias simulaciones con 3.5 minutos de media de separación entre llegadas de aviones a la aproximación. En la tabla 7.13 se muestra la media de los resultados de las múltiples simulaciones realizadas para la aproximación tradicional, en la tabla 7.14 para la aproximación en trombón con cola única y en la tabla 7.15 para la aproximación en trombón con dos colas.

Métrica	**Valor**
Separación media entre aviones	3.77 min
Tiempo total de la operación	79.00 min
Tiempo medio de los aviones en la espera	1.05 min
Utilización de la capacidad de la pista	51 %
Utilización normalizada de la capacidad de la pista	89 %

Cuadro 7.13: Media de los resultados del experimento 5 para aproximación VOR tradicional

Métrica	Valor
Separación media entre aviones	3.84 min
Tiempo total de la operación	82.00 min
Tiempo medio de los aviones en la espera	0.30 min
Utilización de la capacidad de la pista	49 %
Utilización normalizada de la capacidad de la pista	85 %

Cuadro 7.14: Media de los resultados del experimento 5 para aproximación en trombón con cola única

Métrica	Valor
Separación media entre aviones	3.97 min
Tiempo total de la operación	81.50 min
Tiempo medio de los aviones en la espera	0.45 min
Utilización de la capacidad de la pista	49 %
Utilización normalizada de la capacidad de la pista	86 %

Cuadro 7.15: Media de los resultados del experimento 5 para aproximación en trombón con dos colas

7.6.2. Análisis de resultados

Este experimento simula situaciones de bajo tráfico, en las que básicamente todos los aviones pueden iniciar el procedimiento sin tener que realizar una espera. Por este motivo, no hay diferencias entre los dos esquemas en trombón, ya que al ir los aviones tan separados se van a autorizar automáticamente y van a tomar casi siempre el camino más corto en el trombón.

Las utilizaciones normalizadas son muy altas en todos los casos porque ATC absorbe básicamente todo el tráfico y no pierde eficiencia al secuenciar. Los otros datos parecen bajos, pero es porque los aviones llegan muy separados de por sí, no por culpa del proceso de secuenciación.

Al básicamente no haber circuito de espera, el que mejor resultados aporta es el esquema tradicional, ya que es el que tiene distancias más cortas entre los IAFs y el IF. Por este motivo mantener una aproximación tradicional en un aeropuerto es útil para situaciones de bajo tráfico.

Capítulo 8

Presupuesto

En este apartado se reflejará el valor económico que supondría la realización del presente proyecto. Se consideran gastos de personal contratado, emplazamiento y equipo.

Se supone que el tiempo de realización de este proyecto es de 330 horas durante 3 meses. Aunque estos datos no son reales, son realistas, dado que el proyecto es realizable con ese calendario..

8.1. Coste del material

En este apartado se mostrarán los costes aproximados de hardware y software utilizados para el desarrollo del proyecto, en la tabla 8.1. Se realiza la suposición de que se han pagado las licencias de todos los programas y que ninguno ha sido recibido de forma gratuita por ser estudiante de la UPV. Se consideran el coste únicamente en los períodos amortizados durante el proyecto. Se debe tener en cuenta que NetBeans es gratuito y que se utiliza Overleaf para escribir, que también es gratuito. Además, al utilizarse Excel para gestión de datos de experimentos se incluye Microsoft Office.

Costes de material				
Material	**Coste total (€)**	**Período de amortización (meses)**	**Período amortizado (meses)**	**Importe total (€)**
PcCom Revolt	1200.00	72	3	50.00
Licencia MATLAB	800.0	12	3	200.00
Licencia Microsoft 365 Personal	69.00	12	3	17.25
Subtotal				267.25

Cuadro 8.1: Costes de material

8.2. Coste de personal

En la tabla 8.2 se muestran los costes del ingeniero alumno y los dos tutores.

Costes de personal			
Recurso	**Coste (€/h)**	**Horas empleadas**	**Importe total (€)**
Ingeniero alumno	15.00	330.00	4950.00
Tutor	30.00	20.00	600.00
Cotutor	30.00	10.00	300.00
Subtotal			5850.00

Cuadro 8.2: Costes de personal

8.3. Costes indirectos

Estos costes incluyen los gastos de gestión, personal de limpieza, climatización, electricidad... que lleva a cabo la ETSID. De este gasto el proyecto asumiría solo una parte, ya que la ETSID cuenta con muchos alumnos y profesores que utilizan el mismo espacio. Se supone que este proyecto asumiría 400 € en total en costes indirectos.

8.4. Presupuesto total

Se suman los costes expuestos en los 3 apartados anteriores y se muestra el resultado en la tabla 8.3.

Costes de personal	
Concepto	**Subtotal (€)**
Costes de material	267.25
Costes de personal	5850.00
Costes indirectos	400.00
Total sin IVA	6517.25
IVA (21 %)	1368.62
Total proyecto	7885.87

Cuadro 8.3: Costes de material

Capítulo 9

Objetivos de Desarrollo Sostenible

El proyecto tiene relación con los ODS 8 y 13, que se explica de la siguiente manera:

- El ODS 8 consiste en *Promover el crecimiento económico sostenido, inclusivo y sostenible, el empleo pleno y productivo y el trabajo decente para todos.* Aunque no se trata el tema de forma directa al ser un proyecto de aeronavegación que no toca temas económicos, el proyecto BRAIN del Aeropuerto de Barcelona y las aproximaciones en trombón en general son procedimientos que permiten la expansión del tráfico que reciben los aeropuertos y por tanto abren la puerta a la atracción de riqueza hacia las ciudades beneficiadas, al recibir más turismo y viajes empresariales. Esto permite crear nuevos puestos de empleo que ayuden a acercar a las economías locales al pleno empleo.

- El ODS 13 consiste en *Adoptar medidas urgentes para combatir el cambio climático y sus efectos.* Las aproximaciones en trombón y demás tipos de aproximación vectorizados permiten que los aviones permanezcan menos tiempo gastando combustible mientras vuelan las esperas y reducen el tiempo de vuelo en general, por lo que representan un beneficio neto para el medio ambiente a la vez que mejoran la eficiencia económica del aeropuerto. Suponen por tanto una mejora clara con la que deberían contar todos los aeropuertos de alto tráfico.

Capítulo 10

Conclusiones

Como análisis final del presente proyecto, se puede concluir que se han alcanzado los objetivos principales del proyecto. La simulación se ha construido con éxito y representa de manera aceptable los procedimientos reales de aproximación VOR y de transición a la aproximación de la pista RWY24R del Aeropuerto de Barcelona-El Prat. La simulación conseguida tiene limitaciones, ya que la simulación de los circuitos de espera es imperfecta y no demasiado realista, pero es la que daba mejores resultados. En una posible expansión futura del proyecto mejorar la representación de los circuitos de espera.

Los objetivos secundarios del proyecto se han cumplido exitosamente. El desarrollo ha ofrecido una mejora del entendimiento de la programación basada en objetos, y específicamente de Java. La interfaz gráfica obtenida se considera que es intuitiva y fácil de entender para un usuario con menos conocimientos. Se ha explicado la capacidad y se ha calculado para la simulación, y se ha hecho énfasis en diferentes conceptos de secuenciación, especialmente en la vectorización, de manera que lectores menos familiarizados con la aeronavegación sean capaces de comprenderla. Se han programado varios protocolos de control y la entrada de los aviones es aleatoria y sigue una distribución de Poisson.

De los diferentes protocolos de control simulados, el esquema en trombón con colas norte y sur es superior en todos los parámetros analizados en casi todas las circunstancias de tráfico, sin tener ninguna desventaja. Se comprueba que el proyecto de ENAIRE para reformar las aproximaciones al Aeropuerto de Barcelona es acertado y mejora especialmente en escenarios de tráfico mucho más elevado del habitual, lo que permite aumentar el potencial de expansión del aeropuerto, ya que si el aeropuerto no tiene la capacidad de procesar más tráfico es poco realista esperar el crecimiento del mismo. Sin embargo, se nota que en situaciones de bajo tráfico el esquema tradicional es más eficiente que el trombón, ya que la espera no es un factor y es el que tiene las distancias más cortas entre IAFs e IF. Por este motivo es útil mantener los procedimientos tradicionales para circunstancias de bajo tráfico, como por la noche.

En cuanto a la simulación en sí, se ha conseguido que el usuario pueda cambiar el número de aviones, el intervalo medio de tiempo y la aceleración de la simulación respecto al tiempo real. Se podría haber dejado al usuario el tiempo de espera que deben realizar los aviones antes de poder realizar la espera. Se ha dejado en 3 minutos, más debido a los resultados obtenidos que a consideraciones teóricas. El motivo por el que este aspecto debe cambiarse directamente en el código y no en la interfaz es porque se considera un criterio de diseño, pero es un aspecto que se podría estudiar cambiar en una posible expansión.

Como conclusión, la aplicación desarrollada cumple los requisitos que se han establecido en la introducción, y sirve para realizar experimentos y extraer conclusiones sobre

los diferentes esquemas de control. La aplicación causa errores en algunas ocasiones, y necesitaría un período extenso de prueba para eliminarlos del todo, pero estos errores no son comunes y no afectan demasiado a los experimentos. Se ha conseguido comprobar la eficacia de la vectorización en las aproximaciones y se ha conseguido desarrollar un código que de manera autónoma sea capaz de realizar la secuenciación de aviones.

Bibliografía

[1] *Manual de autoridades aeronáuticas, Capítulo 9: Servicios de información aeronáutica y diseño de procedimientos terminales y en ruta*, Dirección general de aeronáutica civil, 11 de mayo de 2012

[2] *Anexo 10 al Convenio sobre Aviación Civil Internacional (6ª edición), Volumen I: Radioayudas para la navegación*, OACI, julio de 2006

[3] *2008 Federal Radionavigation Plan*, EEUU Departamento de Defensa, Departamento de Seguridad Naciona, y Departamento de Transporte.

[4] PH.D. Israel Quintanilla, *GPS: Modelos matemáticos*, Apuntes. Departamento de Ingeniería Cartográfica. Universitat Politècnica de València.

[5] *Simbología de las cartas aeroáuticas*, AIP España, 12 de marzo de 2009

[6] *Documento 8168. Operación de aeronaves, Volumen II: Construcción de procedimientos de vuelo visual y por instrumentos: Sexta Edición*, OACI, 2014.

[7] *ENAIRE, Control de tráfico aéreo (ATC)*. Accedido desde: `https://www.enaire.es/servicios/atm/servicios_de_transito_aereo_ats/control_de_trafico_aereo_atc`, en 20/07/2023

[8] *Continuous Descent Operations (CDO) Manual*, OACI, 2010.

[9] *D4.1 Environmental-friendly airspace structuring and traffic sequencing*, Michael Finke (DLR), 1 de enero de 2020.

[10] *PBN design*, Joan Vila Carbó, Universitat Politècnica de Valencia. Apuntes

[11] *Carta de transición a la aproximación final vuelo por instrumentos RNAV1 BARCELONA/Josep Tarradellas Barcelona-El Prat RWY24R*, AIP ESPAÑA 18 de mayo de 2023.

[12] *Airport Capacity and Delay*, FAA y Departamento de Transporte de EEUU, 1983.

[13] *Analytical Method for Calculating Sustainable Airport Capacity*, Paola Di Mascio, Gregorio Rappoli y Laura Moretti, Universidad de Roma, 6 de noviembre de 2020. Obtenido de: `https://www.semanticscholar.org/paper/Analytical-Method-for-Calculating-Sustainable-Mascio-Rappoli/dd962ec414b51b8c5e2754c8927ffdcf9080cbd2`

[14] *Plan Director del aeropuerto de Josep Tarradellas Barcelona-El Prat*, Aena, 22 de octubre de 1999

[15] *Aeropuerto de Barcelona (web)*, obtenido de: `https://www.aeropuertobarcelona-elprat.com/` a fecha de: 10 de julio de 2023

[16] *ENAIRE mejora los aterrizajes al Aeropuerto de Barcelona-El Prat con un rediseño de su espacio aéreo*, ENAIRE, obtenido de: `https://www.enaire.es/es_ES/2018_04_23/ndp_2018_04_23_proyectobrain` a fecha de: 23 de mayo de 2023

[17] *Blog ENAIRE: Nuevo espacio aéreo eficiente y predecible en Barcelona*, ENAIRE, obtenido de: `https://www.enaire.es/es_ES/2018_08_09/blog_brain_barcelona` a fecha de: 23 de mayo de 2023

[18] *Comparison of 3-D Coordinate Systems*, MathWorks, obtenido de: `https://es.mathworks.com/help/map/choose-a-3-d-coordinate-system.html` a fecha de: 20 de julio de 2023

[19] *Herramienta gráfica para control de tráfico aéreo. Gestión de la estructura del espacio aéreo*, Antonio García-Vilanova Pérez, Universitat Politècnica de València, julio de 2018

[20] *Java tutorial*, J.A. Vila Carbó y A. Rodas Jordá, Universitat Politècnica de València, 2016

[21] *Carta de aproximación por instrumentos VOR BARCELONA/Josep Tarradellas Barcelona-El Prat RWY24R*, AIP ESPAÑA 15 de junio de 2023.

[22] *Parámetros de capacidad aeroportuaria. Temporada de verano 2023*, AECFA, septiembre de 2022. Obtenido de: `https://www.slotcoordination.es/es/slot/normativa`