Trabajo Fin de Grado
Grado en Ingeniería Aeroespacial

Mejora computacional de herramientas de planificación de trayectorias de avión bajo meteorología adversa

Autor: Narciso Valverde González

Tutor: Antonio Franco Espín

Dpto. de Ingeniería Aeroespacial y Mecánica de Fluidos
Escuela Técnica Superior de Ingeniería
Universidad de Sevilla

Sevilla, 2023

Trabajo Fin de Grado
Grado en Ingeniería Aeroespacial

Mejora computacional de herramientas de planificación de trayectorias de avión bajo meteorología adversa

Autor:
Narciso Valverde González

Tutor:
Antonio Franco Espín
Profesor Contratado Doctor

Dpto. de Ingeniería Aeroespacial y Mecánica de Fluidos
Escuela Técnica Superior de Ingeniería
Universidad de Sevilla

Sevilla, 2023

Trabajo Fin de Grado: Mejora computacional de herramientas de planificación de trayectorias de avión bajo meteorología adversa

Autor: Narciso Valverde González
Tutor: Antonio Franco Espín

El tribunal nombrado para juzgar el trabajo arriba indicado, compuesto por los siguientes profesores:

Presidente:

Vocal/es:

Secretario:

acuerdan otorgarle la calificación de:

El Secretario del Tribunal

Fecha:

Agradecimientos

Este trabajo supone la finalización de cuatro años de proceso de crecimiento tanto académico como personal que ha supuesto el estudio del *Grado en Ingeniería Aeroespacial*. Por ello, me gustaría dedicar mi proyecto a todas aquellas personas sin las que esto no habría sido posible.

A toda mi familia, por valorarme mucho más de lo que yo hago. A mis abuelos, por preocuparse siempre por mí, a mi padre y mi hermano por ser apoyos incondicionales y, en especial, a mi madre, quien me guio por el camino de las ciencias y la ingeniería, por ofrecerme su ayuda en todo lo que le es posible y el gran esfuerzo que dedica día tras día para sacarnos a todos adelante.

A mis amigos de toda la vida de los grupos de *El corral* y *Caraqueños* quienes, a pesar del poco tiempo que les dedico, me regalan unas horas de desconexión y disfrute las cuales me dan la vida y me permiten seguir mis metas.

A los amigos que he ido haciendo a lo largo de la carrera, sobre todo, a los pertenecientes al grupo *La Zahurda*, por el apoyo, la ayuda y los ánimos que nos hemos brindado mutuamente a lo largo de estos cuatro años.

A Antonio Franco, por la tutela realizada en este Trabajo Fin de Grado, allanando el camino en todo momento. Su conocimiento, amabilidad y paciencia han permitido generar un clima de trabajo ameno e incrementar mi motivación por el proyecto.

A todos los profesores de la escuela que han propiciado que, a lo largo de los cuatro años de carrera, mi pasión acerca de todo lo que rodea al mundo de la aeronáutica se haya incrementado además de mantener mi afán por seguir adquiriendo conocimientos en este ámbito.

Narciso Valverde González
Grado en Ingeniería Aeroespacial

Sevilla, 2023

Resumen

En este trabajo se plantea la mejora computacional de las herramientas de planificación de trayectorias de avión en presencia de meteorología adversa.

El algoritmo que más tiempo consume en las trayectorias de evitación de tormentas es el de generación del grafo de visibilidad. La implementación actual de la que dispone el *Grupo de Ingeniería Aeroespacial* propone una solución que resulta muy costosa temporalmente cuando el número de nodos del grafo comienza a aumentar. El primer objetivo del proyecto será implementar este algoritmo para, tras ello, programar una rutina avanzada de creación del grafo de visibilidad basada en el método de *Lee* con el propósito de conseguir una reducción relevante en el tiempo de ejecución del algoritmo con respecto a las implementaciones más sencillas de las que se dispone actualmente.

Abstract

This project proposes the computational improvement of aircraft trajectory planning tools in the presence of adverse weather conditions.

The algorithm that consumes the most time in storm avoidance trajectories is the visibility graph generation algorithm. The current implementation available to the *Aerospace Engineering Group* proposes a solution that is temporarily very expensive when the number of nodes in the graph begins to increase. To begin with, the project will implement this algorithm and, in order to complete its main objective, it will program an advanced routine for creating the visibility graph based on *Lee*'s method with the purpose of achieving a relevant reduction in the execution time of the algorithm with respect to the simplest implementations currently available.

Notación

O	Punto de origen del vuelo
D	Punto de destino del vuelo
V	Matriz de vértices de los obstáculos poligonales disjuntos del grafo
$G_V(V)$	Grafo de visibilidad de V
i	Índice del vértice de origen de un segmento de recta $\overline{ij}$
i^+	Índice del vértice posterior al nodo i
i^-	Índice del vértice anterior al nodo i
j	Índice del vértice de destino de un segmento de recta $\overline{ij}$/conjunto de vértices de destino de los segmentos de recta $\overline{ij_{j0}}$
$j0$	Índice auxiliar que recorre j en caso de que este sea un conjunto de vértices
j^+	Índice del vértice posterior al nodo j
j^-	Índice del vértice anterior al nodo j
$\overline{ij}$	Segmento de recta que une i y j
w	Índice de un vértice genérico del grafo
k	Índice de la arista obstáculo contenida entre los vértices j y j^+
k^+	Índice de la arista obstáculo posterior a la arista k
k^-	Índice de la arista obstáculo anterior a la arista k
e_{k0}	Índice de una arista obstáculo genérica
(x,y)	Coordenadas de un punto en el mapa en la proyección cartográfica de Mercator
λ	Longitud
λ_0	Longitud media
ϕ	Latitud
$\vec{e}_x$, $\vec{e}_y$, $\vec{e}_z$	Vectores unitarios en las direcciones de los ejes x, y y z respectivamente
$\vec{r}$, $\vec{s}$	Vectores de la base no ortogonal formada por las aristas obstáculo adyacentes al vértice i
$\vec{q}$	Vector resultado del producto vectorial de $\vec{r}$ y $\vec{s}$
M	Matriz de la base no ortogonal formada por los vectores $\vec{r}$ y $\vec{s}$
$\vec{v}$	Vector director de la recta que une los nodos i y j

$\vec{v}_0$	Vector $\vec{v}$ normalizado
a, b	Escalares tales que $\vec{v} = a\vec{r} + b\vec{s}$
$\vec{t}$	Vector dirigido del vértice j al j^-
$\vec{t}_0$	Vector $\vec{t}$ normalizado
$\vec{u}$	Vector dirigido del vértice j al j^+
$\vec{u}_0$	Vector $\vec{u}$ normalizado
n	Número de aristas obstáculo del grafo
N	Número de nodos del grafo (orden del grafo)
N_{in}	Número de puntos dado en V
WPs	Matriz de waypoints (puntos de referencia) del grafo
NaN	*Not a Number* en MATLAB
W	Vector que almacena las posiciones de los NaN en la matriz V
$npol$	Número de polígonos del grafo
V_0	Matriz V sin NaN
$index$	Vector de índices de los nodos del grafo
(x_V, y_V)	Coordenadas de un punto de la matriz V en el mapa en la proyección cartográfica de Mercator
$V_{Mercator}$	Matriz de vértices del grafo tras aplicar la transformación cartográfica de Mercator
(x_{WPs}, y_{WPs})	Coordenadas de un punto de la matriz WPs en el mapa en la proyección cartográfica de Mercator
$WPs_{Mercator}$	Matriz de waypoints del grafo tras aplicar la transformación cartográfica de Mercator
E	Matriz que permite hallar los vértices extremos de una determinada arista obstáculo del grafo
$Einv$	Matriz que permite hallar las aristas obstáculo incidentes a un determinado vértice del grafo
$Conex$	Matriz de adyacencia "lógica" del grafo
p	Indicador del polígono en que se encuentra el vértice i
N_{vec_p}	Número de vértices del polígono p
$VerticesData$	Matriz con los datos necesarios para realizar una ordenación de vértices en sentido antihorario
σ	Semirrecta que comienza en el vértice i y se extiende a lo largo del sentido positivo del eje x
$\overline{\sigma}$	Segmento de recta que comienza en el vértice i y se extiende a lo largo del sentido positivo del eje x una longitud de 2π
xk	Vector que almacena las coordenadas en el eje x de los puntos de intersección entre dos polilíneas
yk	Vector que almacena las coordenadas en el eje y de los puntos de intersección entre dos polilíneas
K	Matriz que almacena los índices de las aristas obstáculo intersecadas por el segmento $\overline{\sigma}$
$k0$	Índice auxiliar que recorre las intersecciones detectadas entre el segmento $\overline{\sigma}$ y las aristas de los polígonos obstáculo
$midpoint$	Coordenadas del punto medio de la recta que une los vértices i y j

in	Variable bandera que indica si un punto está en el interior de un polígono
on	Variable bandera que indica si un punto se encuentra sobre una arista de un polígono
vis	Variable bandera que indica si los vértices i y j son mutuamente visibles
T	Conjunto ordenado que almacena aristas obstáculo
n_T	Número de aristas dado en T
ρ	Conjunto que almacena las distancias al vértice i de las aristas almacenadas en T
$Tinv$	Conjunto que almacena la posición que ocupan los índices de las aristas almacenadas en T
$TreeData$	Matriz con los datos necesarios para realizar una ordenación de las aristas almacenadas en T
$isvertice$	Variable bandera que indica si el punto de intersección de $\overline{\sigma}$ con los polígonos obstáculo que se trata es un vértice en la inicialización de T
$isverticerep$	Variable bandera que indica si el punto de intersección de $\overline{\sigma}$ con los polígonos obstáculo que se trata es un vértice repetido en la inicialización de T
x_{comp}	Conjunto que almacena $t_{0,x}$ y $u_{0,x}$ en los casos en que sea necesario
pos_k	Posición que ocupa la arista k en el conjunto T
$newedges$	Conjunto en el que se almacenan las nuevas aristas obstáculo a introducir en T
$pos_{newedges}$	Conjunto que almacena la posición que ocupan en T las nuevas aristas obstáculo introducidas
P	Punto de intersección de la recta que pasa por los vértices i y j con e_1
A, B	Nodos extremos de e_1
β, γ	Escalares tales que $\vec{iP} = \beta\vec{v}$ y $\vec{AP} = \gamma\vec{AB}$
d_{lox}	Distancia loxodrómica entre dos puntos
χ_{lox}	Ángulo de rumbo loxodrómico entre dos puntos
α_{lox}	Distancia angular loxodrómica entre dos puntos
R_T	Radio de la Tierra
h	Altura de vuelo
Adj	Matriz de adyacencia con pesos del grafo
i_{vec}, j_{vec}	Vectores que almacenan los índices de los pares de vértices que son mutuamente visibles
n_c	Variable contador asociada a los vectores i_{vec} y j_{vec}
G	Grafo creado a partir de la matriz Adj
$path$	Índice de los nodos a seguir para recorrer el camino más corto entre los puntos O y D
$distance$	Distancia en km recorrida a lo largo del camino más corto entre los puntos O y D
$load$	Comando de MATLAB para cargar datos almacenados en un archivo
$struc2cell$	Comando de MATLAB para convertir una estructura en tipo *cell*

find	Comando de MATLAB para encontrar los índices de los elementos no nulos en una matriz o vector
unique	Comando de MATLAB para obtener valores únicos en las filas o columnas de una matriz o vector
rmmissing	Comando de MATLAB para eliminar filas o columnas de una matriz que contengan *NaN*
length	Comando de MATLAB para obtener la longitud de un vector o la mayor dimensión de una matriz
mean	Comando de MATLAB para calcular la media de los elementos de una matriz o vector
false	Comando de MATLAB para crear una matriz cuadrada llena de ceros lógicos
polyxpoly	Comando de MATLAB para detectar las intersecciones entre dos polilíneas
inpolygon	Comando de MATLAB para para determinar si un conjunto de puntos se encuentra dentro de un polígono
abs	Comando de MATLAB para calcular el valor absoluto de un número o los elementos de una matriz
graph	Comando de MATLAB para la creación de grafos
shortestpath	Comando de MATLAB para determinar el camino más corto entre dos nodos en un grafo
geoplot	Comando de MATLAB para trazar gráficos y visualizar datos en coordenadas geográficas en un mapa
reducem	Comando de MATLAB que reduce el número de puntos de una región según una cierta tolerancia

Índice de Figuras

1.1 Radar meteorológico de un avión de investigación de la NASA (imagen extraída de [1]) 2
1.2 Zona de convergencia intertropical (imagen extraída de [2]) 3
1.3 Restos del vuelo 242 de Southern Airways (imagen extraída de [3]) 4
1.4 Visualización en cabina que ofrece un radar meteorológico (imagen extraída de [4]) 5

2.1 Ejemplos de casos para los que el algoritmo concluirá que los vértices i y j no son visibles 12
2.2 Diagrama de flujo de la primera versión del algoritmo *Ingenuo* 15
2.3 Recta que une el vértice i con vértice j en una situación genérica 16
2.4 Base no ortogonal formada por los vectores característicos de las aristas adyacentes al vértice i 17
2.5 Tipos de vértice en un polígono 17
2.6 Vectores de la base no ortogonal y vector director de la recta que une i con j 18
2.7 Descomposición del vector $\vec{v}$ en función de los vectores $\vec{r}$ y $\vec{s}$ 18
2.8 Casos en que la recta que une i y j no pasa por el interior de un polígono 19
2.9 Diagrama de flujo de la segunda versión del algoritmo *Ingenuo* 24

3.1 Trazado del segmento $\overline{ij}$ 26
3.2 Árbol de búsqueda binario de las aristas intersecadas 26
3.3 Trazado de la semirrecta σ 28
3.4 Diagrama de flujo del programa principal del algoritmo de *Lee* 34
3.5 Casos en que solo existe un vértice por encima del nodo i 36
3.6 Diagrama de flujo de la función VISIBLEVERTICESVA(i,V) 38
3.7 Caso en que la intersección del segmento $\overline{\sigma}$ con los obstáculos poligonales es un vértice y está repetido 41
3.8 Casos en que uno de los puntos de intersección de $\overline{\sigma}$ con los obstáculos poligonales en la inicialización de T es un vértice 42
3.9 Vectores $\vec{u}$ y $\vec{t}$ y distancia al vértice i 43
3.10 Caso en que los vértices i, j_{j0-1} y j_{j0} se encuentran alineados 45
3.11 Diagrama de flujo de la inicialización de T en la función VISIBLEVERTICESVA(i,V) 46
3.12 Casos de actualización de T 51
3.13 Diagrama de flujo de la comprobación de visibilidad de los vértices y actualización de T en la función VISIBLEVERTICESVA(i,V) 54
3.14 Diagrama de flujo de la actualización de T en la función VISIBLEVERTICESVA(i,V) 55
3.15 Intersección de la recta que pasa por i y j con e_1 57
3.16 Diagrama de flujo de la función VISIBLEVA(i,j_{j0},e_1) 59

4.1 Región de meteorología adversa en consideración 61

4.2 Envolvente convexa de la región de meteorología adversa en consideración 62
4.3 Diagrama de flujo del algoritmo de *Ramer-Douglas-Peucker* 63
4.4 Grafo de visibilidad del vuelo Bruselas-Sarajevo 64
4.5 Trayectoria más corta del vuelo Bruselas-Sarajevo 64
4.6 Trayectoria más corta del vuelo Madrid-Kiev 65
4.7 Trayectoria más corta del vuelo Turín-Varsovia 65
4.8 Trayectoria más corta del vuelo Ljubljana-Berna 66
4.9 Trayectoria más corta del vuelo entre $O = (43.1°N, 0.730°W)$ y $D = (44.7°N, 6.75°E)$ 66
4.10 Trayectoria más corta del vuelo entre $O = (44.5°N, 4.20°E)$ y $D = (50.0°N, 22.5°E)$ 67
4.11 Representación del ajuste de los puntos obtenidos en la resolución de problemas 68

Índice de Códigos

2.1 Cálculos preliminares de la primera versión del algoritmo *Ingenuo* 11
2.2 Cómputo del grafo de visibilidad de la primera versión del algoritmo *Ingenuo* 13
2.3 Obtención del camino más corto en la primera versión del algoritmo *Ingenuo* 14
2.4 Cálculos preliminares de la segunda versión del algoritmo *Ingenuo* 19
2.5 Formación de la base no ortogonal centrada en un nodo i en la segunda versión del algoritmo *Ingenuo* 21
2.6 Comprobación de la visibilidad de los vértices en la segunda versión del algoritmo *Ingenuo* 22

3.1 Cálculos preliminares del programa principal del algoritmo de *Lee* 30
3.2 Cálculo de las matrices E y *Einv* en el programa principal del algoritmo de *Lee* 32
3.3 Cómputo del grafo de visibilidad en el programa principal del algoritmo de *Lee* 33
3.4 Operaciones previas a la ordenación de vértices en sentido antihorario de la función VISIBLEVERTICESVA(i,V) 34
3.5 Tratamiento del vértice en caso de que solo quede uno tras la aplicación de simetría en la función VISIBLEVERTICESVA(i,V) 36
3.6 Ordenación de vértices en sentido antihorario de la función VISIBLEVERTICESVA(i,V) 37
3.7 Intersección del segmento $\overline{\sigma}$ y establecimiento de conjuntos en la inicialización del conjunto ordenado T en la función VISIBLEVERTICESVA(i,V) 39
3.8 Comprobación de las intersecciones en la inicialización del conjunto ordenado T en la función VISIBLEVERTICESVA(i,V) 41
3.9 Almacenamiento de aristas en la inicialización del conjunto ordenado T en la función VISIBLEVERTICESVA(i,V) 43
3.10 Ordenación del conjunto ordenado T en la función VISIBLEVERTICESVA(i,V) 45
3.11 Comprobación de visibilidad de los vértices en la función VISIBLEVERTICESVA(i,V) 47
3.12 Búsqueda de las aristas almacenadas en T en la función VISIBLEVERTICESVA(i,V) 47
3.13 Caso en que sea necesario eliminar dos aristas en la actualización de T en la función VISIBLEVERTICESVA(i,V) 48
3.14 Casos en que sea necesario eliminar una arista e introducir otra en la actualización de T en la función VISIBLEVERTICESVA(i,V) 49
3.15 Caso en que es necesario añadir dos aristas en la actualización de T en la función VISIBLEVERTICESVA(i,V) 52
3.16 Comprobación de que el segmento $\overline{ij}$ no pasa por el interior de un polígono del que i es un vértice en la función VISIBLEVA(i,j_{j0},e_1) 56
3.17 Comprobación de la intersección de $\overline{ij}$ con e_1 en la función VISIBLEVA(i,j_{j0},e_1) 58

A.1 Función para el cálculo de distancias loxodrómicas 74

B.1 Primera versión del algoritmo *HeapSort* 76
B.2 Primera versión de la función auxiliar *Heapify* del algoritmo *HeapSort* 77
B.3 Segunda versión del algoritmo *HeapSort* 79
B.4 Segunda versión de la de la función auxiliar *Heapify* del algoritmo *HeapSort* 80

C.1 Algoritmo de *Búsqueda Binaria e Inserción* 81

D.1 Primera versión del algoritmo *Ingenuo* 83
D.2 Segunda versión del algoritmo *Ingenuo* 85
D.3 Programa principal del algoritmo de *Lee* 87
D.4 Función VISIBLEVERTICESVA(i,V) 89
D.5 Función VISIBLEVERTICESOD(i,V) 94
D.6 Función VISIBLEVA(i,j_{j0},e_1) 98
D.7 Función VISIBLEOD(i,j_{j0},e_1) 98

Índice Abreviado

Resumen III
Abstract V
Notación VII
Índice de Figuras XI
Índice de Códigos XIII
Índice Abreviado XV

1 Introducción 1
1.1 Motivación 1
1.2 Objetivo del proyecto 6
1.3 Estructura del documento 7

2 Algoritmo *Ingenuo* 9
2.1 Introducción al método *Ingenuo* 9
2.2 Desarrollo e implementación del algoritmo *Ingenuo* 10
2.3 Mejora del costo computacional del algoritmo *Ingenuo* 16

3 Algoritmo de *Lee* 25
3.1 Introducción al método de *Lee* 25
3.2 Desarrollo e implementación del algoritmo 30

4 Resultados 61
4.1 Región de meteorología adversa en consideración y generación de problemas 61
4.2 Problemas resueltos 63
4.3 Análisis de resultados 67

5 Conclusiones y líneas futuras 71

Apéndice A Cálculo de distancias loxodrómicas 73

Apéndice B Algoritmos de ordenación 75
B.1 Primera versión del método *HeapSort* 75
B.2 Segunda versión del método HeapSort 79

Apéndice C Algoritmo de *Búsqueda Binaria e Inserción* 81

Apéndice D Códigos de MATLAB **83**
D.1 Primera versión del algoritmo *Ingenuo* 83
D.2 Segunda versión del algoritmo *Ingenuo* 85
D.3 Algoritmo de *Lee* 87

Bibliografía 101

Índice

Resumen III
Abstract V
Notación VII
Índice de Figuras XI
Índice de Códigos XIII
Índice Abreviado XV

1 Introducción 1
1.1 Motivación 1
1.2 Objetivo del proyecto 6
1.3 Estructura del documento 7

2 Algoritmo *Ingenuo* 9
2.1 Introducción al método *Ingenuo* 9
2.2 Desarrollo e implementación del algoritmo *Ingenuo* 10
2.2.1 Cálculos preliminares 10
2.2.2 Cómputo del grafo de visibilidad 11
2.2.3 Obtención del camino más corto 13
2.3 Mejora del costo computacional del algoritmo *Ingenuo* 16
2.3.1 Desarrollo de un criterio basado en las aristas adyacentes al nodo i para determinar si una recta que pasa por este nodo cruza el polígono del que i es vértice 16
2.3.2 Implementación del algoritmo *Ingenuo* con el criterio desarrollado 19

3 Algoritmo de *Lee* 25
3.1 Introducción al método de *Lee* 25
3.2 Desarrollo e implementación del algoritmo 30
3.2.1 Programa principal 30
Cálculos preliminares 30
Cómputo del grafo de visibilidad 33
Obtención del camino más corto 33
3.2.2 Funciones VISIBLEVERTICESVA(i,V) y VISIBLEVERTICESOD(i,V) 33
Ordenación de los vértices en sentido antihorario a partir del sentido positivo del eje x 33
Inicialización del conjunto ordenado T 38
Comprobación de la visibilidad de los vértices y actualización de T 45
3.2.3 Funciones VISIBLEVA(i, j_{j0}, e_1) y VISIBLEOD(i, j_{j0}, e_1) 56

4 Resultados **61**
4.1 Región de meteorología adversa en consideración y generación de problemas 61
4.2 Problemas resueltos 63
4.2.1 Problema 1. Vuelo Bruselas-Sarajevo 64
4.2.2 Problema 2. Vuelo Madrid-Kiev 65
4.2.3 Problema 3. Vuelo Turín-Varsovia 65
4.2.4 Problema 4. Vuelo Ljubljana-Berna 66
4.2.5 Problema 5. Vuelo entre $O = (43.1°N, 0.730°W)$ y $D = (44.7°N, 6.75°E)$ 66
4.2.6 Problema 6. Vuelo entre $O = (44.5°N, 4.20°E)$ y $D = (50.0°N, 22.5°E)$ 67
4.3 Análisis de resultados 67

5 Conclusiones y líneas futuras **71**

Apéndice A Cálculo de distancias loxodrómicas **73**

Apéndice B Algoritmos de ordenación **75**
B.1 Primera versión del método *HeapSort* 75
B.2 Segunda versión del método HeapSort 79

Apéndice C Algoritmo de *Búsqueda Binaria e Inserción* **81**

Apéndice D Códigos de MATLAB **83**
D.1 Primera versión del algoritmo *Ingenuo* 83
D.2 Segunda versión del algoritmo *Ingenuo* 85
D.3 Algoritmo de *Lee* 87
D.3.1 Programa principal 87
D.3.2 Función VISIBLEVERTICESVA(i, V) 89
D.3.3 Función VISIBLEVERTICESOD(i, V) 94
D.3.4 Función VISIBLEVA(i, j_{j0}, e_1) 98
D.3.5 Función VISIBLEOD(i, j_{j0}, e_1) 98

Bibliografía 101

1 Introducción

La presencia de regiones de meteorología adversa durante un vuelo compromete su comodidad y el alcance de sus objetivos. En este trabajo, se plantea la implementación de algoritmos basados en la generación del grafo de visibilidad como mecanismos de evitación de estas tormentas.

En este capítulo se exponen algunos de los peligros que representan estas zonas y la solución que se propone para evitarlas. Por otra parte, se presentará el objetivo del proyecto, así como la estructura que seguirá este documento.

1.1 Motivación

A pesar de que una aeronave no despega si la tripulación no tiene la certeza de poder garantizar la seguridad de pasajeros y mercancías, diversas circunstancias hacen que esta deba enfrentarse con frecuencia a condiciones atmosféricas adversas en la ejecución de sus planes de vuelo. Los pilotos completan un largo y duro entrenamiento para volar en situaciones extremas con difíciles maniobras y, además, adquieren conocimientos acerca de la meteorología, pero la situación climatológica puede sorprenderlos en el transcurso del vuelo y cambiar en un intervalo corto de tiempo.

Esto puede provocar problemas en los procesos de aterrizaje y despegue o alteración de los horarios de los vuelos que desembocan en retrasos molestos para los pasajeros. Asimismo, se pueden producir cancelaciones de los trayectos dependiendo del tipo de avión de que se trate y, en el peor de los casos, accidentes de distinta índole.

Durante el aterrizaje, una tormenta puede hacer que una aeronave quede atrapada en corrientes y ser movida de un lado a otro por los vientos cruzados, causando que, si este no pudiese llevarse a cabo, el vuelo se desvíe a otro aeropuerto. En el peor de los casos, es posible que la tripulación pierda el control de la aeronave y se produzca un siniestro con consecuencias de diversa importancia. Estos problemas también pueden producirse en el transcurso del despegue, aunque en estos supuestos es posible realizar una cancelación del trayecto.

A lo largo del vuelo, una tormenta puede producir muchos efectos incómodos. Los aviones disponen de radares meteorológicos que se sitúan en el morro de la aeronave y pueden recoger señales de las nubes que aparecen delante de la cabina. De esta manera, se intenta realizar el vuelo a una determinada distancia del núcleo de la tormenta (se tiene un margen de seguridad de 20 millas náuticas según [8]). Sin embargo, estos radares detectan gotas de agua y son ineficaces ante otras

inclemencias del tiempo.

Figura 1.1 Radar meteorológico de un avión de investigación de la NASA (imagen extraída de [1]).

Los principales efectos que una zona climática hostil puede producir y afectar la ruta de una aeronave se resumen a continuación (citando [9]):

- **Turbulencias**

  Tienen lugar debido al desplazamiento de masas de aire y pueden causar sacudidas de las aeronaves. Generalmente, estas están diseñadas para soportar condiciones extremas, no obstante, dependiendo de la gravedad de las turbulencias, pueden producir daños estructurales trágicos.

  El mayor problema que representan las turbulencias es la provocación de situaciones muy incómodas para los pasajeros, además de la posibilidad de la aparición de lesiones.

  Por otra parte, pueden provocar dificultades para el correcto funcionamiento del piloto automático. Esto es debido a factores como variaciones inesperadas de la altitud o la inestabilidad del rumbo.

- **Vientos de cara**

  Producen retrasos en la llegada al aeropuerto y, aunque generalmente el tiempo adicional del trayecto suele ser tenido en cuenta calculándose antes del despegue de la aeronave, puede que estos vientos surjan de manera impredecible durante su trayectoria.

- **Rayos**

  Las aeronaves actúan como Jaulas de Faraday en caso de la caída de un rayo, es decir, la corriente eléctrica proveniente nunca pasa a su interior, sino que se desvía siendo conducida por su exterior. De esta forma, el ala de los aviones tiene "mechas estáticas", las cuales disipan la electricidad.

  Aun así, puntualmente, un rayo puede penetrar en el fuselaje a través de una fisura o producir el llamado "fuego de San Telmo", que se trata de electricidad estática que parpadea en el parabrisas, o bien otros problemas como el que ocurrió en la pérdida de un Boeing 747 de la Fuerza Iraní en 1976 cerca de Madrid, cuando un rayo que generó vapor en un depósito de combustible provocó una explosión.

- **Granizo y agua**

  Pese a que los aviones están diseñados y certificados para soportar ciertos niveles de impacto de granizo y operar en condiciones de lluvia, las condiciones extremas pueden representar riesgos significativos para la aeronave y su seguridad.

  En caso de caída de granizo severo, se podrían producir daños tanto en la estructura como en los distintos sensores y antenas.

  En el supuesto de lluvia intensa, los motores podrían verse afectados, debido a que podrían obstruirse las tomas de aire o causar problemas en los procesos de combustión.

Los efectos anteriores se agravan en las zonas de nuestro planeta que se encuentran en las estaciones otoño e invierno, debido a que el clima se vuelve más incómodo y las tormentas eléctricas son más frecuentes en ellas.

Por añadidura, en nuestra atmósfera, se halla la denominada "zona de convergencia intertropical" (ITCZ), región del globo terrestre donde convergen los vientos alisios del hemisferio norte con los del hemisferio sur. En ella, las corrientes ascendentes generadas por las aguas cálidas y el movimiento de rotación de la esfera terrestre crean masas compactas de nubes tempestuosas. De esta manera, las masas de aire con grandes nubes de tormenta pueden encontrarse a alturas que oscilan entre la proximidad a la superficie hasta más de los 12 *km* de altura. Esta ITCZ se localiza ligeramente por encima o por debajo del Ecuador geográfico según la época del año como se muestra en la Figura 1.2. Constituye la zona de mayor actividad tormentosa de la Tierra y debe ser cruzada, con asiduidad, por las aeronaves.

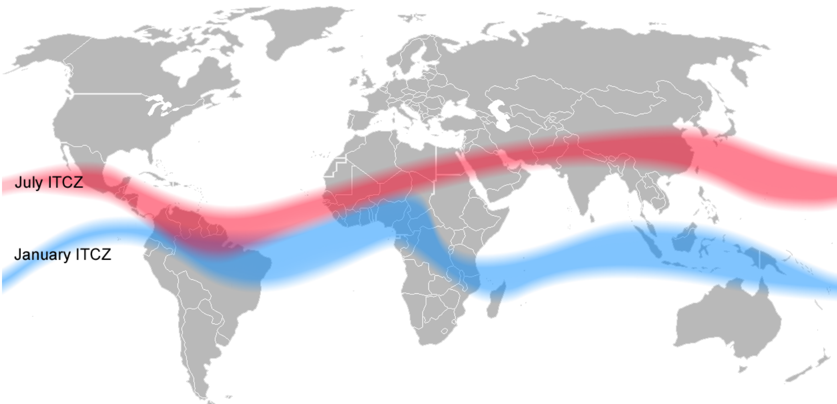


Figura 1.2 Zona de convergencia intertropical (imagen extraída de [2]).

Un ejemplo de accidente causado por las circunstancias anteriores es el vuelo 272 de Southern Airways del 4 de abril de 1977, sufrido por un McDonnell Douglas DC-9. Citando [3], en opinión de Francis H. McAdams, miembro de la NTSB (National Transportation Safety Board), organismo que llevó a cabo la investigación del accidente, la causa razonable de este accidente fue la decisión del capitán de entrar en una zona climática hostil en lugar de esquivarla.

Figura 1.3 Restos del vuelo 242 de Southern Airways (imagen extraída de [3]).

A raíz de lo expuesto resulta evidente que una aeronave debe evitar, en la medida de lo posible, el trayecto en el interior de una tormenta. Para determinar cómo debe un piloto eludir la entrada en una zona climática hostil, es necesario realizar un análisis.

Volar por debajo de una tormenta podría no ser solución al problema, pues un vuelo prolongado a baja altitud supone un coste operacional excesivo, además de un impacto importante a nivel de gestión de tráfico aéreo. Por otra parte, en el supuesto de volar por debajo de los 10000 ft, aparecen la posibilidad del impacto de aves o la proximidad a núcleos urbanos y el problema ambiental derivado del ruido producido por el avión.

Volar por encima no es siempre una buena alternativa, sobre todo si la altura de la tormenta supera demasiado la altitud de crucero. Las aeronaves podrían presentar limitaciones en cuanto a su rendimiento y capacidades de vuelo. Además, debido a la disminución de la presión del aire y cantidad de oxígeno, se requerirían sistemas de presurización y suministro de oxígeno adicionales.

Por ende, la mejor solución suele ser bordear la tormenta. Así, con el uso de radares meteorológicos que detectan las masas nubosas y los núcleos de precipitación en conjunción con las previsiones climatológicas, será posible diseñar rutas alternativas al paso a través de una zona climática hostil.

En las aeronaves de línea aérea, normalmente se vuela con el piloto automático conectado para aliviar la carga de trabajo y prestar atención también a otros aspectos de la operación. Cuando se

detecta meteorología adversa en la ruta debido a tormentas, los pilotos solicitan al control aéreo la autorización para el desvío. Así, el avión sale de su ruta original y es conducido hacia zonas libres de mal tiempo empleando la información que el radar de a bordo muestra en su pantalla (Figura 1.4).

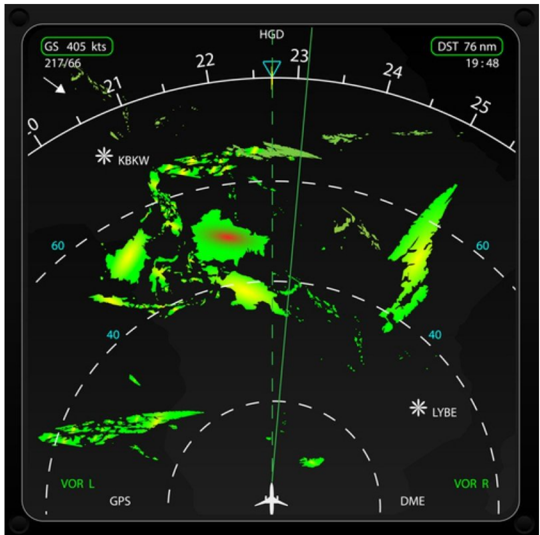


Figura 1.4 Visualización en cabina que ofrece un radar meteorológico (imagen extraída de [4]).

Consecuentemente, resulta necesario idear una aplicación que, en este punto, calcule con celeridad dichas trayectorias para guiar la aeronave por el camino más corto. De esta forma, no se dejará al factor suerte la resolución de todas las vicisitudes que puedan ocasionarse por la presencia de tormentas y otras alteraciones de la normalidad climatológica.

En este contexto, el *Grupo de Ingeniería Aeroespacial* ha estado coordinando y trabajando, durante los últimos años, en proyectos financiados por la *Comisión Europea*, centrados en el análisis y la gestión del impacto de la meteorología adversa en el tráfico aéreo. En el curso de dicha investigación, se ha detectado la necesidad de mejorar la eficiencia computacional de los algoritmos de evitación de tormentas.

Puesto que las tormentas como las que se muestran en la Figura 1.4 son modeladas como polígonos, el principal objetivo de estos algoritmos es hallar el camino más corto entre un punto de origen del vuelo O y un punto de destino D evitando una serie de obstáculos poligonales cuyos vértices vienen dados en una matriz denominada V. El procedimiento seguido para conseguir este propósito es hallar, en primer lugar, el grafo de visibilidad del conjunto $\{V \cup [O,D]\}$ cuya información será almacenada en una estructura denominada *Conex* para, tras ello, asignar un peso a cada conexión de dos vértices mutuamente visibles, (i,j), y utilizar el algoritmo de *Dijkstra* el cual permite obtener el camino más corto desde el nodo de origen O al de destino D.

A continuación se adjunta un pseudocódigo sencillo para facilitar una mayor comprensión de la estructura que siguen estos algoritmos de evitación de tormentas ([13]):

Pseudocódigo 1.1 SHORTESTPATH(V,O,D)

Entrada: Una serie de obstáculos poligonales disjuntos definidos por un conjunto ordenado de vértices, V y dos puntos que no se encuentren en el interior de los polígonos, O y D.

Salida: Camino más corto entre O y D.

1: $Conex \leftarrow$ VISIBILITYGRAPH$(V \cup [O,D])$
2: Asignar a cada conexión (i,j) un peso.
3: Emplear el algoritmo de Dijkstra para computar el camino más corto entre O y D en *Conex*.

Por consiguiente, la principal dificultad de estos algoritmos radica en la construcción del grafo de visibilidad. La geometría computacional lo define de la siguiente manera ([10]): dado un conjunto V de obstáculos poligonales disjuntos, se define el grafo de visibilidad $G_V(V)$ como aquel grafo cuyos nodos son los vértices de V en el que una conexión entre dos vértices i y j existe si son mutuamente visibles, es decir, si el segmento $\overline{ij}$ no atraviesa el interior de ningún obstáculo en V.

1.2 Objetivo del proyecto

Tal y como se ha expuesto, en el ámbito de la aeronáutica la meteorología es un factor a tener en cuenta en cada momento. Las circunstancias que rodean la aeronave deben ser observadas continuamente ya que enfrentarse a condiciones adversas supone un peligro inminente que se debe evitar a toda costa.

Según se ha establecido en la sección anterior, la necesidad primordial en la implementación de los algoritmos de evitación de tormentas radica en la generación del grafo de visibilidad. Por consiguiente, el primer objetivo de este trabajo será desarrollar un algoritmo para la generación del grafo de visibilidad a través del método *Ingenuo*, que constituye la implementación actual de la que dispone el *Grupo de Ingeniería Aeroespacial*.

La rutina resultante propone una solución que es muy costosa en cuanto al factor tiempo cuando el número de nodos del grafo comienza a aumentar, dando lugar a largos periodos de reacción de la tripulación ante alguna alerta sobre una situación peligrosa.

El principal objetivo de este proyecto será conseguir una respuesta más rápida pues puede ocurrir que, en determinadas tesituras, las condiciones ambientales cambien impredeciblemente en breves periodos de tiempo. En estos casos, la rapidez en las decisiones es clave y puede suponer la supervivencia de los pasajes. Para ello, se busca reducir el tiempo de ejecución del cálculo del grafo de visibilidad en los algoritmos evitando largos intervalos de espera y poner en un peligro innecesario toda la misión.

Con este propósito, se desarrollará un algoritmo para el cómputo del grafo de visibilidad basado en el método de *Lee*, cuyo costo computacional teórico es significativamente menor que el presentado por el método *Ingenuo*.

Por último, se resolverán una serie de problemas con el fin de comprobar, principalmente, si la implementación del método de *Lee* para la generación del grafo de visibilidad da lugar a una mejora temporal significativa con respecto a aquella solución basada en el método *Ingenuo*.

Considerando que la seguridad en los aviones no es responsabilidad solo de los pilotos, técnicos de mantenimiento y controladores aéreos, sino de la interrelación entre las diferentes áreas que

sostienen este transporte, el objetivo es potenciar en cuanto a factor tiempo y trayectoria la labor de las tripulaciones y contribuir así al hecho de que el transporte más seguro, año tras año, sea el suministrado por las aeronaves.

Para el desarrollo de las diferentes herramientas informáticas planteadas se empleará MATLAB debido a que, además de ser una plataforma utilizada por ingenieros y científicos para analizar datos, desarrollar algoritmos y crear modelos, es el lenguaje de programación con el cual el autor se encuentra más familiarizado.

1.3 Estructura del documento

En esta sección se describe la estructura que presentará este trabajo, que se fraccionará en cinco capítulos.

En este primer capítulo se han presentado tanto la motivación como el objetivo que perseguirá este proyecto.

En el Capítulo 2, se desarrollará el algoritmo que constituye la implementación actual de la que dispone el *Grupo de Ingeniería Aeroespacial* para la obtención de la trayectoria más corta a lo largo de una región de meteorología adversa, cuya generación del grafo de visibilidad se basa en el método *Ingenuo*. Tras ello, se desarrollará e implementará un criterio con el fin de reducir ligeramente el costo computacional del algoritmo.

En el Capítulo 3, se desarrollará un algoritmo para la obtención de la trayectoria más corta a lo largo de una región de meteorología adversa, en el que la generación del grafo de visibilidad se basará en el método de *Lee*, cuyo costo computacional se esperará sea significativamente menor que el asociado al algoritmo desarrollado en el Capítulo 2.

En el Capítulo 4, se mostrará la región tormentosa que se considerará en este trabajo, así como el proceso seguido para la generación de problemas con el fin de testear los algoritmos desarrollados en los Capítulos 2 y 3. Tras ello, se realizará un análisis de los resultados obtenidos.

Finalmente, en el Capítulo 5, se extraerán una serie de conclusiones y se definirán una serie de líneas futuras a realizar en otros trabajos.

Por otra parte, en los Apéndices A, B y C, se mostrarán una serie de funciones que se han empleado pero cuya implementación no es objeto de este trabajo y, en el Apéndice D, se adjuntarán todos los códigos de MATLAB desarrollados para la implementación de los algoritmos planteados en los Capítulos 2 y 3.

2 Algoritmo *Ingenuo*

Una vez entendida la complejidad del problema al que se pretende dar solución, se van a elaborar una serie de algoritmos con tal propósito.

En este capítulo, en primer lugar, se va a formular e implementar un algoritmo para la obtención de la trayectoria más corta cuya generación del grafo de visibilidad se base en el método *Ingenuo*. Seguidamente, se desarrollará y se aplicará un criterio para reducir ligeramente el tiempo de ejecución del mismo sin perder la esencia del método.

2.1 Introducción al método *Ingenuo*

Con el fin de la generación del grafo de visibilidad, para la implementación del método *Ingenuo* se va a desarrollar un algoritmo exhaustivo que va recorriendo todas las parejas de vértices (i, j) y comprobando si el segmento $\overline{ij}$ corta a las aristas de los polígonos que conforman el grafo, es decir, a las aristas obstáculo. Este algoritmo viene descrito en el siguiente pseudocódigo ([10]):

Pseudocódigo 2.1 VISIBILITYGRAPH(V)

Entrada: La variable V, formada por una serie de obstáculos poligonales disjuntos definidos por un conjunto ordenado de vértices y dos puntos que no se encuentren en el interior de los polígonos.

Salida: Grafo de visibilidad dado en *Conex*.

1: Inicializar un grafo vacío *Conex*.
2: **para** cada vértice $i \in V$???? **hacer**
3:     **para** cada vértice $j \in [V - i]$ **hacer**
4:         **si** el segmento $\overline{ij}$ no interseca ninguna arista obstáculo e_{k0} **entonces**
5:             Añadir $\overline{ij}$ a *Conex*.
6: **devolver** *Conex*

Nótese que V es, en este caso, la concatenación de la variable V presentada en el Pseudocódigo 1.1 y los nodos de origen O y destino D.

Puesto que se designará como n al número de aristas, el número de nodos del grafo será $N = n + 2$ (el número de aristas a los que se añaden los puntos de origen y destino). Si se observa el Pseudocódigo 2.1, la cuarta línea de código representaría una complejidad $O(n = N - 2)$ cuya operación dentro del bucle, debido a la tercera línea de código, se repite $N - 1$ veces y, debido a la segunda línea, lo anterior se repite N veces, por lo que el costo computacional total del algoritmo sería de

$O[N(N-1)(N-2)] \sim O(N^3)$.

En la siguiente sección, se detalla el desarrollo del algoritmo planteado, así como su implementación en MATLAB.

2.2 Desarrollo e implementación del algoritmo *Ingenuo*

En el caso de la implementación del método *Ingenuo*, se van a combinar los algoritmos descritos en los Pseudocódigos 1.1 y 2.1 en un mismo programa debido a la sencillez del resultado.

Como entradas al algoritmo, se tendrán el punto de origen del vuelo, O, el punto de destino, D, y una matriz de vértices de la región tormentosa, V. En esta última estructura se encuentran las coordenadas de los nodos de los distintos polígonos obstáculo del grafo ordenados en sentido horario, de forma que el primer vértice de cada figura se encuentra repetido para cerrar el polígono y, además, los vértices de distintos polígonos se encuentran separados por *NaN* (*Not a Number* en MATLAB). Si se denomina N_{in} al número de puntos dados en V, esta matriz será de dimensiones $N_{in} \times 2$.

La primera coordenada de los puntos O y D será la latitud geográfica y, la segunda, la longitud geográfica. Por su parte, en la primera columna de la matriz V aparecerán las coordenadas de la latitud geográfica de los vértices y en su segunda columna, las coordenadas de la longitud geográfica. Dichos puntos vendrán dadas en grados decimales.

2.2.1 Cálculos preliminares

En primer lugar, se introducirán desde el principio los puntos de origen, O, y de destino, D, al final de la matriz V, separados también por *NaN*.

Acto seguido, interesa crear una variable donde únicamente aparezcan los vértices del grafo sin puntos repetidos y sin *NaN* entre nodos de distintos polígonos. Este parámetro será una matriz que se denominará *WPs* (Waypoints) y será de dimensiones $N \times 2$. Para hallarla, primeramente, se aplicará el comando *unique*, pasando también los argumentos de *'rows'* para indicar que se debe realizar la búsqueda de valores únicos considerando las filas completas de una matriz, y *'stable'* para indicar que el orden relativo de los valores únicos se debe preservar en el resultado. Una vez calculada esta matriz, se obtendrá el número total de nodos del grafo, N.

Tras ello, puesto que se va a crear un grafo plano, es necesario convertir la información meteorológica dada en la Tierra, una región esférica, a un mundo plano. Este objetivo se va a realizar a través de la transformación cartográfica de Mercator, dada por las siguientes ecuaciones ([12]):

$$x = \lambda - \lambda_0 \tag{2.1}$$

$$y = \ln\left[\left(\frac{\pi}{4} + \frac{\phi}{2}\right)\right] \tag{2.2}$$

Donde (x,y) son las coordenadas de un punto en el mapa en la proyección de Mercator, λ es la longitud en radianes, ϕ es la latitud en radianes y λ_0 es una longitud media que vendrá dada por la siguiente fórmula:

$$\lambda_0 = \frac{1}{N}\sum_{i=1}^{N}\lambda_i \tag{2.3}$$

Esta transformación se aplicará tanto a las coordenadas dadas en la matriz V, como a las dadas en la matriz WPs, obteniendo como resultado las matrices $V_{Mercator}$ y $WPs_{Mercator}$. En este caso, en la primera columna de las matrices se introducirán las coordenadas en el eje x de los distintos puntos y en la segunda columna, las coordenadas en el eje y.

De igual forma, es necesario inicializar la variable que almacene información asociada a la conectividad de los distintos nodos del grafo. Con tal propósito, se empleará una matriz tipo *false* y será la anteriormente mencionada *Conex*. Tendrá dimensiones $N \times N$.

```
1  clear all; close all; clc;
2
3  %% Insertar puntos de origen y destino (Madrid y Kiev en este caso)
4  O = [40.471926, -3.562640]; D = [50.345001, 30.894699];
5
6  %% Insertar polígonos obstáculo
7
8  storms = load('TFG_Narciso.mat');
9  storms = struct2cell(storms);
10 V = [storms{1} storms{2}];
11
12 %% CÁLCULOS PRELIMINARES
13
14 V = [V; NaN(1,2); O; NaN(1,2); D]*pi/180;
15
16 % Matriz de WPs y orden del grafo
17 WPs = rmmissing(unique(V, 'rows', 'stable')); % Primero se suprimen vértices repetidos
18 % con el comando 'unique' y tras ello se suprimen los NaN con el comando 'rmmissing'
19 N = length(WPs);
20
21 % Implementación de la transformación de Mercator
22 Lon_0 = mean(WPs(:,2));
23 x_V = (V(:,2) - Lon_0); y_V = log(tan(pi/4 + V(:,1)/2)); V_Mercator = [x_V, y_V];
24 x_WPs = (WPs(:,2) - Lon_0); y_WPs = log(tan(pi/4 + WPs(:,1)/2)); WPs_Mercator = ...
25     [x_WPs, y_WPs];
26
27 % Inicialización de la matriz de adyacencia lógica
28 Conex = false(N,N);
29 %
```

Código 2.1 Cálculos preliminares de la primera versión del algoritmo *Ingenuo*.

2.2.2 Cómputo del grafo de visibilidad

Se recorrerán todos los vértices del grafo y, fijado un vértice, se comprobará la visibilidad con respecto al resto. Se aprovechará el carácter simétrico del problema, por lo que el vértice de origen de la recta, i, viajará a través del intervalo $[1, N-1]$ y, el vértice de destino, j, a lo largo del intervalo $[i+1, N]$.

Fijados los vértices (i, j), se comprobará la intersección del segmento que los une, $\overline{ij}$, con todas las aristas de los obstáculos poligonales. Para ello se hará uso del comando *polyxpoly*, pasando

también el argumento *'unique'* con objeto de evitar intersecciones duplicadas (en el caso de que un punto de intersección fuese un vértice repetido, de esta forma el comando no detectaría dos puntos de intersección exactamente iguales).

Tras ello, para decidir acerca de la visibilidad de un vértice, se mirará en primer lugar el número de puntos de intersección. Este, es siempre superior o igual a 2 ya que los propios vértices i y j son detectados como puntos de intersección. Así, si el número de puntos de intersección no es igual a dos, equivaldrá a que la recta que une i y j ha intersecado algún punto que no es ni el vértice i ni el j, por lo que dichos nodos no serán mutuamente visibles.

Hay casos en los que, en realidad, aún habiendo más de dos puntos de intersección, los vértices sí serían visibles, como se muestran en la Figura 2.1. No obstante, en estos casos se puede imponer que no existe visibilidad directa entre los vértices i y j, pero sí desde i hasta el primer vértice intersección ($j-1$ o w en la Figura 2.1). Así, tras una aplicación recursiva de esta misma idea, aunque j no sea visible desde i, podrá llegarse desde i hasta j recorriendo el mismo camino que si lo fuese. La principal ventaja de este enfoque radica en la disminución en las conexiones del grafo resultante mediante la eliminación de conexiones redundantes.

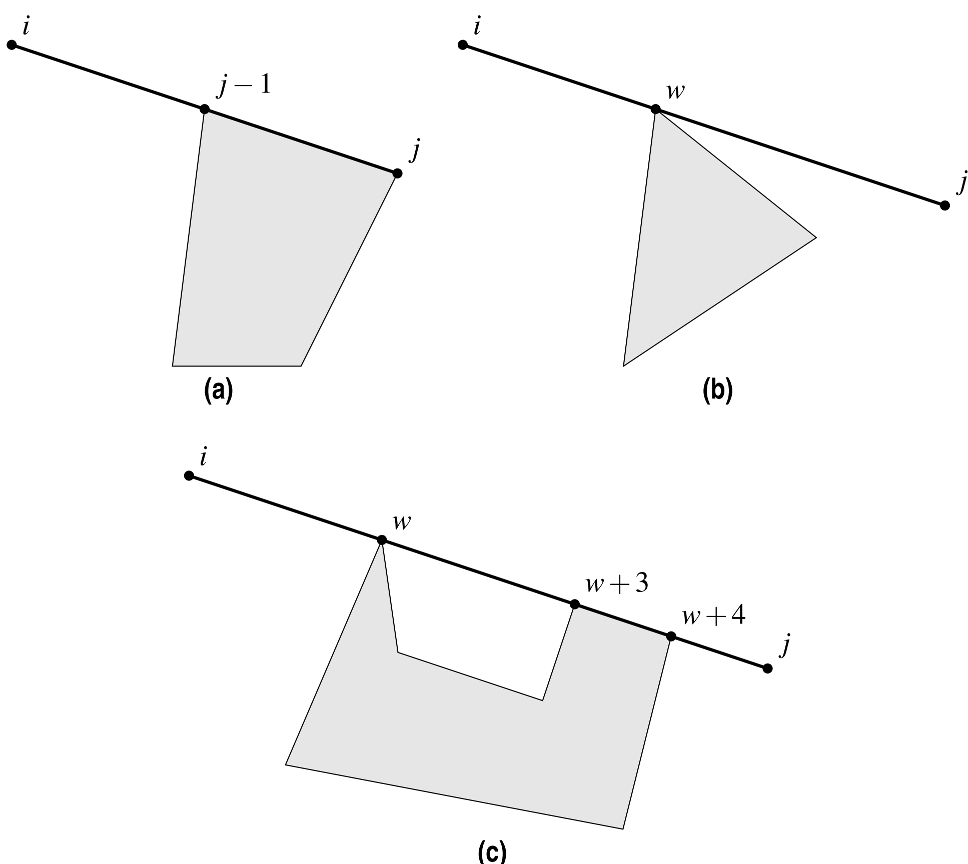


Figura 2.1 Ejemplos de casos para los que el algoritmo concluirá que los vértices i y j no son visibles.

Lo único que queda por comprobar para conocer si los vértices i y j son visibles es el paso de la recta que los une por el interior de un polígono. Teniendo en cuenta lo ejecutado anteriormente, esto solo puede ocurrir si dichos puntos se encuentran en el mismo polígono. Para ello, se calcula el

punto medio del segmento $\overline{ij}$ y se hace uso del comando *inpolygon*, que permite determinar tanto si el punto mencionado no se encuentra en el interior de un polígono como si está sobre una de las aristas obstáculo, en cuyos casos i y j serán mutuamente visibles lo cual se indicará en la matriz *Conex*.

```
30 %% CÓMPUTO DEL GRAFO DE VISIBILIDAD
31 for i = 1 : N - 1
32     for j = i + 1 : N
33         % Intersección entre la recta que une i con j y el conjunto de polígonos
34         % obstáculo
35         [xk, yk] = polyxpoly([WPs_Mercator(i,1), WPs_Mercator(j,1)], ...
36             [WPs_Mercator(i,2), WPs_Mercator(j,2)], V_Mercator(:,1), ...
37             V_Mercator(:,2),‘unique’);
38
39         if length(xk) == 2 % Único caso en que se considera que i y j puedan ser
40         % visibles
41             % Cálculo del punto medio
42             midpoint = [(xk(1) + xk(2))/2, (yk(1) + yk(2))/2];
43
44             % Se comprueba tanto si el punto medio está dentro del polígono como si
45             % está en una arista
46             [in, on] = inpolygon(midpoint(1), midpoint(2), V_Mercator(:,1), ...
47                 V_Mercator(:,2));
48             if in == 0 || (in == 1 && on == 1)
49                 Conex(i,j) = 1; Conex(j,i) = 1;
50             end
51         end
52     end
53 end
54 %
```

Código 2.2 Cómputo del grafo de visibilidad de la primera versión del algoritmo *Ingenuo*.

2.2.3 Obtención del camino más corto

Antes del desarrollo de las líneas de código para la obtención del camino más corto que finalizarían la rutina, se recuerda que estas quedan fuera de la complejidad y el tiempo de computación del algoritmo para la generación del grafo de visibilidad.

Una vez hallada la matriz *Conex*, se creará una matriz de adyacencia con pesos que se denominará *Adj*. Con esa finalidad, se almacenarán tanto los índices (i, j) de los puntos mutuamente visibles (elementos tales que $Conex_{i,j} = 1$) como la distancia loxodrómica d_{lox} entre i y j, calculada por medio de una función auxiliar. Esta viene detallada en el Apéndice A.

Se debe tener en cuenta que la matriz *Adj* ha de ser simétrica y, dado que se emplea un conjunto de líneas de código en el que no se aprovecha el carácter simétrico del problema y considerando que el cálculo de las distancias loxodrómicas lleva acarreados ciertos errores de redondeo, se ejecutarán una serie de comandos para reducir el número de cifras significativas característico de las distancias calculadas a 9. Seguidamente, se crea la matriz de adyacencia empleando el comando *sparse*.

Llegado a este punto es necesario hacer un inciso: puesto que se pretende hallar el camino más corto entre O y D, se debería emplear una matriz de adyacencia cuyos pesos fuesen, en lugar de las distancias loxodrómicas, las ortodrómicas, pues estas representan las distancias más cortas entre dos

puntos en el globo terráqueo. No obstante, puestos que los puntos mutuamente visibles en el grafo se encuentran relativamente cerca, los dos tipos de distancias mencionados serán muy similares entre sí.

Posteriormente, se emplea el comando *graph* para construir el grafo a partir de la matriz *Adj* y, a continuación, se hace uso del comando *shortestpath*, el cual implementa el método de *Dijkstra*, para hallar tanto los vértices a seguir como la distancia recorrida a lo largo del camino más corto.

En adición a lo anterior, se proporcionan unos comandos para dibujar tanto el grafo de visibilidad (en este caso, se propone el empleo de un doble bucle *for* para aprovechar la simetría del problema debido al alto costo temporal del comando *geoplot*) como la trayectoria más corta de forma independiente.

```
55 %% OBTENCIÓN DEL CAMINO MÁS CORTO ENTRE O Y D
56
57 % Creación de la matriz de adyacencia con pesos
58 [i_vec, j_vec] = find(Conex);
59 for n_c = 1:length(i_vec)
60     d_lox(n_c) = LoxodromicDistance(WPs(i_vec(n_c),1), WPs(i_vec(n_c),2), ...
61         WPs(j_vec(n_c),1), WPs(j_vec(n_c),2), 0);
62 end
63
64 significant_numbers = 9; % Número de cifras significativas que se considera
65 factor = 10.^(significant_numbers - ceil(log10(abs(d_lox))));
66 d_lox = round(d_lox .* factor) ./ factor;
67
68 Adj = sparse(i_vec,j_vec,d_lox);
69
70 % Cálculo del camino más corto
71 G = graph(Adj);
72 [path, distance] = shortestpath(G, N - 1, N);
73
74 %% OPERACIONES AUXILIARES AL ALGORITMO
75
76 %%% Para trazar el grafo de visibilidad, descomentar las siguientes líneas de código
77 % figure(1)
78 % for i = 1 : N
79 %     for j = i : N
80 %         if Conex(i,j) == 1
81 %             geoplot([WPs(i,1), WPs(j,1)]*180/pi, [WPs(i,2), WPs(j,2)]*180/pi);
82 %             hold on
83 %         end
84 %     end
85 % end
86 %
87 %%% Para trazar el camino más corto, descomentar las siguientes líneas de código
88 % figure(2)
89 % geoplot(V(:,1)*180/pi, V(:,2)*180/pi,'b');
90 % hold on
91 % geoplot(WPs(path,1)*180/pi, WPs(path,2)*180/pi,'r');
92 % hold on
93 % for i=1:length(path)
94 %     geoplot(WPs(path(i),1)*180/pi,WPs(path(i),2)*180/pi,'rd')
95 % end
```

Código 2.3 Obtención del camino más corto en la primera versión del algoritmo *Ingenuo*.

En la Figura 2.2, se muestra un diagrama de flujo de la rutina desarrollada con el fin de la implementación del algoritmo *Ingenuo* para la generación del grafo de visibilidad. Este tipo de diagramas introducidos a lo largo del documento presentará la siguiente convención de símbolos:

- : representan el inicio y el final del algoritmo.
- : representan los datos de entrada que recibe el algoritmo.
- : representan procesos u operaciones.
- : representan decisiones tras las que son posibles varios caminos alternativos.
- ⟶ : representan líneas de flujo que indican el orden de ejecución de las operaciones.

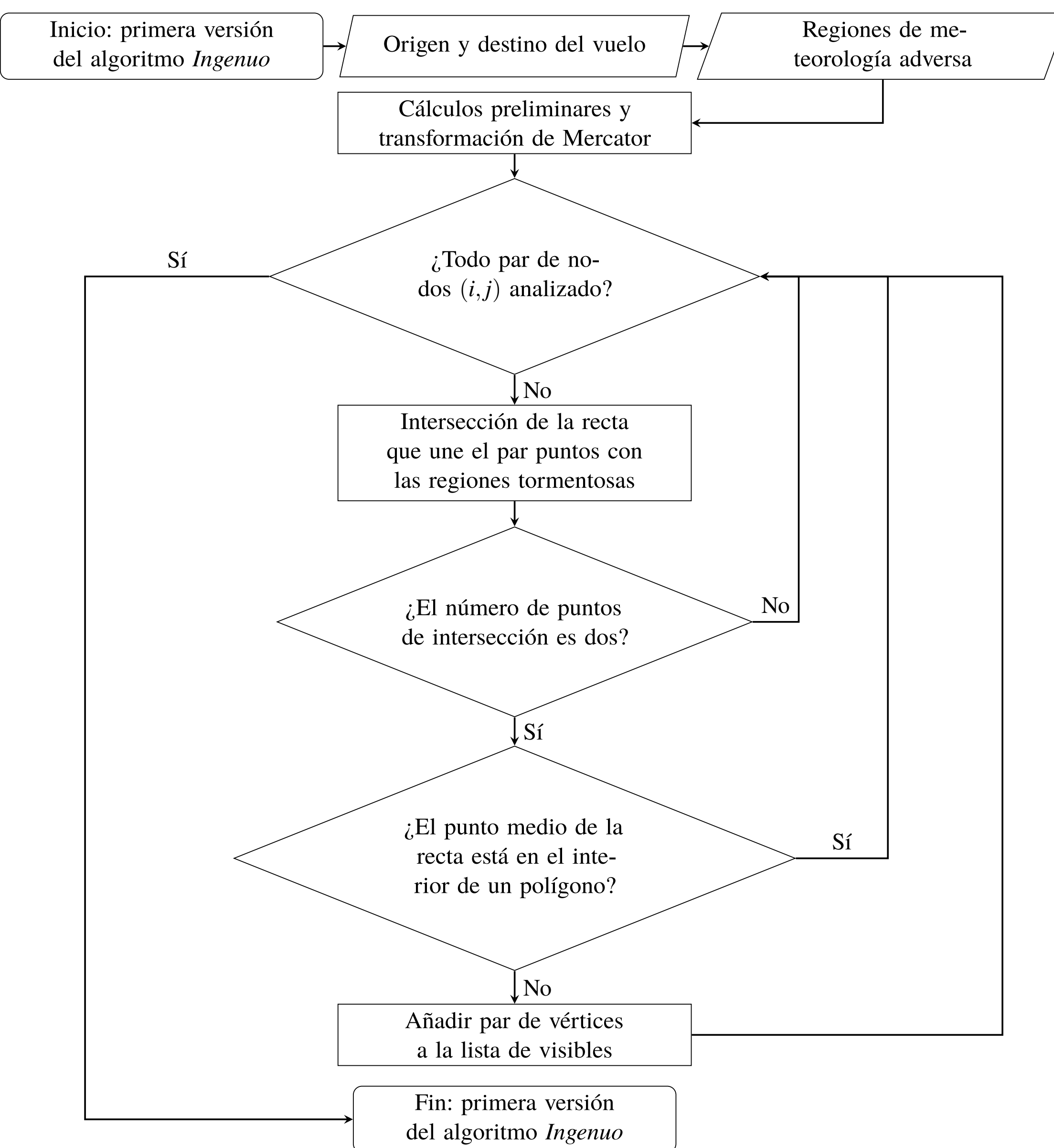


Figura 2.2 Diagrama de flujo de la primera versión del algoritmo *Ingenuo*.

2.3 Mejora del costo computacional del algoritmo *Ingenuo*

El algoritmo implementado en la sección anterior presenta un gran inconveniente a pesar de su sencillez: el costo temporal del mismo es muy elevado cuando el número de nodos del grafo comienza a aumentar.

En esta sección se va a abordar este problema sin perder la esencia del método *Ingenuo*. Por tanto, se tratará de sustituir el criterio basado en el punto medio para determinar si una arista pasa por el interior de un polígono, ya que conlleva un costo computacional relevante debido a que el comando empleado realiza comprobaciones de un punto con todo el conjunto de nodos del grafo.

Para este criterio, se empleará un planteamiento vectorial y, pese a ser usado en este algoritmo cuando el número de puntos de intersección de la recta que une los vértices i y j es dos, se desarrollará de forma que sea válido para cualquier otra situación.

2.3.1 Desarrollo de un criterio basado en las aristas adyacentes al nodo i para determinar si una recta que pasa por este nodo cruza el polígono del que i es vértice

Considérese una situación genérica como la mostrada en la Figura 2.3:

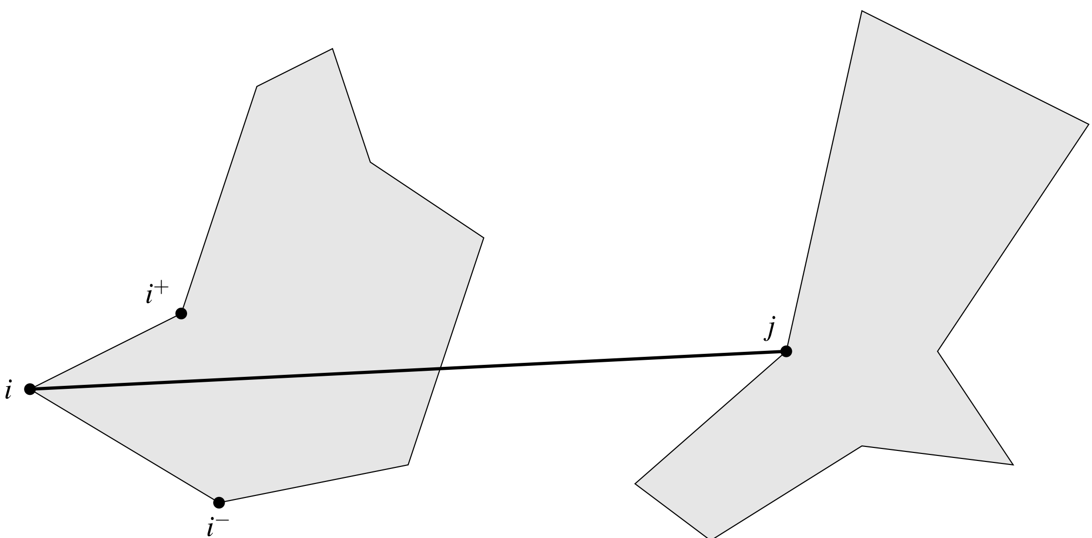


Figura 2.3 Recta que une el vértice i con vértice j en una situación genérica.

Se va a contemplar una base no ortogonal formada por el vector $\vec{r}$, dirigido del vértice i al nodo posterior, i^+, y el vector $\vec{s}$, orientado del vértice i al nodo anterior, i^-:

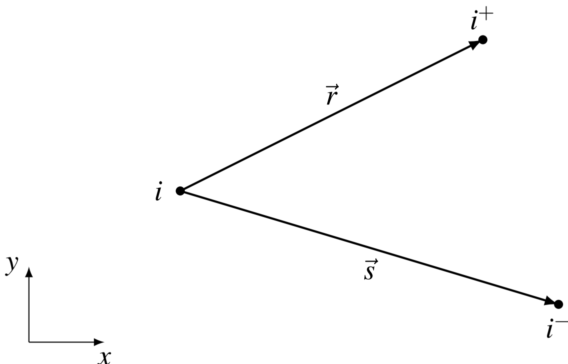


Figura 2.4 Base no ortogonal formada por los vectores característicos de las aristas adyacentes al vértice i.

Para determinar si la recta que une i con j pasa por el interior del polígono del que i es vértice, es necesario, en primer lugar, conocer si este vértice es convexo o cóncavo.

Un vértice convexo en un polígono se caracteriza porque el ángulo interior en dicho vértice es inferior a 180° (Figura 2.5a). Análogamente, un vértice cóncavo en un polígono se caracteriza porque el ángulo interior en dicho vértice es superior a 180° (Figura 2.5b).

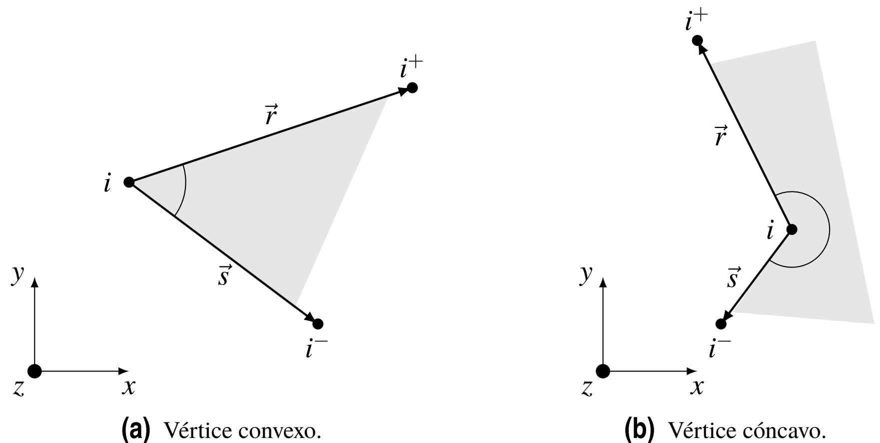


Figura 2.5 Tipos de vértice en un polígono.

Para conocer si un vértice es convexo o cóncavo, sabiendo que los nodos de los distintos polígonos están ordenados en sentido horario, se realiza el producto vectorial de los vectores $\vec{r}$ y $\vec{s}$, cuyo resultado se denominará $\vec{q}$. Puesto que estos vectores están contenidos en el plano xy, $\vec{q}$ tendrá una única componente no nula en el eje z:

$$\vec{q} = \vec{r} \times \vec{s} = \begin{vmatrix} \vec{e}_x & \vec{e}_y & \vec{e}_z \\ r_x & r_y & 0 \\ s_x & s_y & 0 \end{vmatrix} = \begin{Bmatrix} 0 \\ 0 \\ r_x s_y - r_y s_x \end{Bmatrix} = \begin{Bmatrix} 0 \\ 0 \\ q_z \end{Bmatrix} \tag{2.4}$$

Con la finalidad de la aplicación del criterio, resulta necesario conocer el orden en que vienen dados los vértices en la matriz V. Como se ha comentado en la sección 2.2, los nodos se dispondrán en sentido horario pero, en caso de que esto no fuese así, se cambiará la organización de los vértices en

MATLAB para que queden en el orden que interesa. Teniendo en cuenta lo anterior, aplicando el criterio de la mano derecha se obtiene que q_z es negativo si el vértice es convexo y positivo si es cóncavo.

Por otra parte, se hallará el vector $\vec{v}$, dirigido del vértice i al vértice j, es decir, el vector director de la recta que une los nodos.

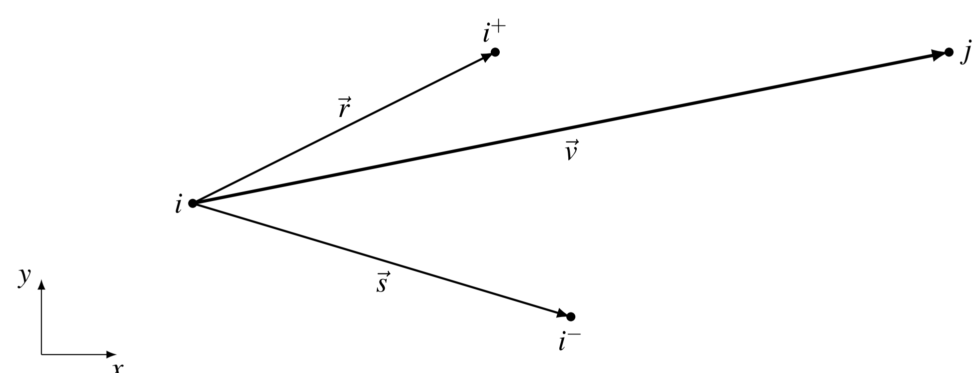


Figura 2.6 Vectores de la base no ortogonal y vector director de la recta que une i con j.

Puesto que se conocen las coordenadas de los nodos en consideración, los vectores $\vec{r}$, $\vec{s}$ y $\vec{v}$ son dato, por lo que es posible expresar el vector $\vec{v}$ de la siguiente forma:

$$\vec{v} = a\vec{r} + b\vec{s} \tag{2.5}$$

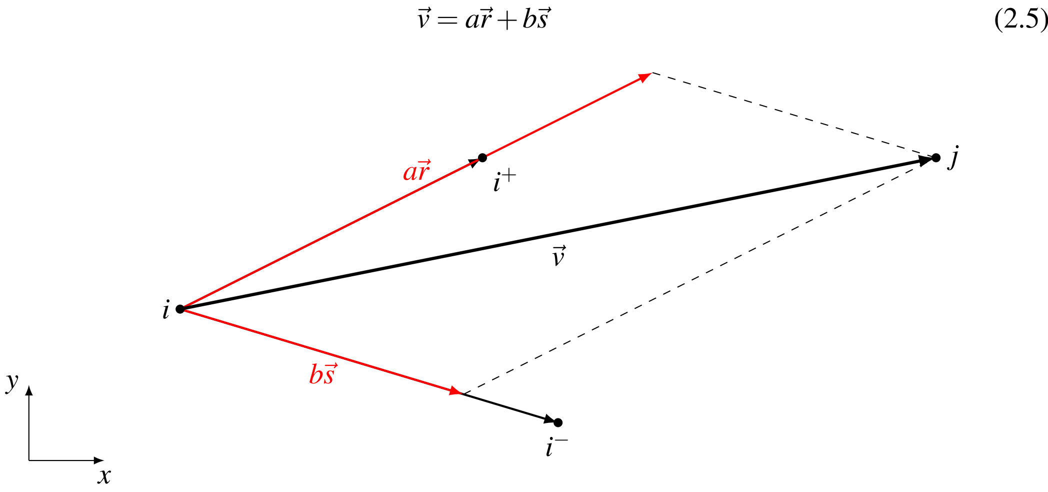


Figura 2.7 Descomposición del vector $\vec{v}$ en función de los vectores $\vec{r}$ y $\vec{s}$.

Donde a y b son dos escalares no conocidos a priori. Para hallarlos, puesto que la matriz de la base no ortogonal, M, es conocida:

$$M = \begin{bmatrix} \vec{r} & \vec{s} \end{bmatrix} = \begin{bmatrix} r_x & s_x \\ r_y & s_y \end{bmatrix} \tag{2.6}$$

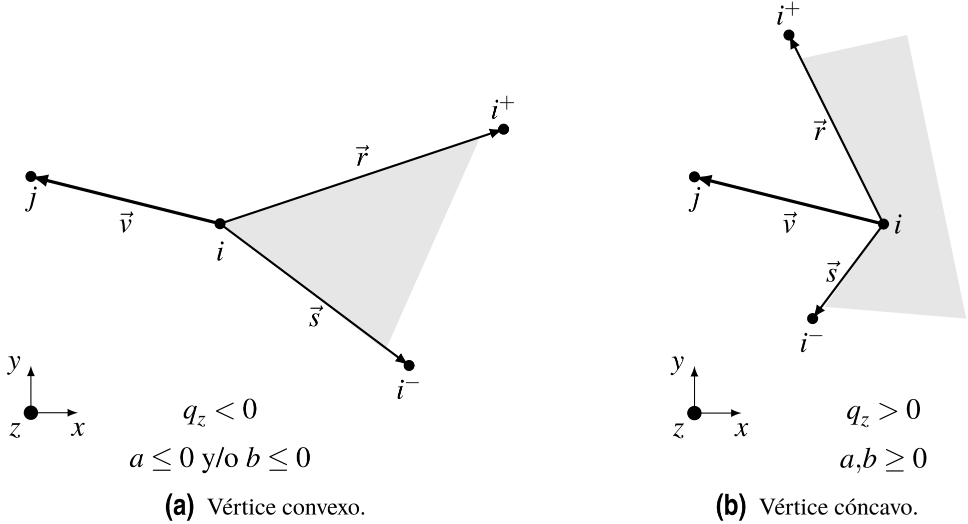
Es posible plantear un sistema de ecuaciones donde las incógnitas son a y b, que se resolverá analíticamente empleando la regla de *Cramer*:

$$a\vec{r} + b\vec{s} = \vec{v} \rightarrow \begin{bmatrix} r_x & s_x \\ r_y & s_y \end{bmatrix} \begin{Bmatrix} a \\ b \end{Bmatrix} = \begin{Bmatrix} v_x \\ v_y \end{Bmatrix} \tag{2.7}$$

$$a = \frac{\begin{vmatrix} v_x & s_x \\ v_y & s_y \end{vmatrix}}{\begin{vmatrix} r_x & s_x \\ r_y & s_y \end{vmatrix}} = \frac{v_x s_y - v_y s_x}{q_z} \qquad b = \frac{\begin{vmatrix} r_x & v_x \\ r_y & v_y \end{vmatrix}}{\begin{vmatrix} r_x & s_x \\ r_y & s_y \end{vmatrix}} = \frac{r_x v_y - r_y v_x}{q_z} \tag{2.8}$$

Una vez hallados a y b, para el criterio se tendrán en cuenta tanto los signos de estos escalares como el de q_z, obteniendo dos casos para los que la recta que une i y j no pasa por el interior de un polígono:

- El primero de ellos es para el supuesto de que el vértice sea convexo ($q_z < 0$) y ni a ni b sean mayores que 0 (Figura 2.8a).
- El segundo es para el caso de que el vértice sea cóncavo ($q_z > 0$) y a y b sean mayores o iguales a 0 (Figura 2.8b).



(a) Vértice convexo. **(b)** Vértice cóncavo.

Figura 2.8 Casos en que la recta que une i y j no pasa por el interior de un polígono.

2.3.2 Implementación del algoritmo *Ingenuo* con el criterio desarrollado

Para el desarrollo del algoritmo *Ingenuo* modificado, a los cálculos preliminares mostrados en la sección 2.2.1 y el Código 2.1, se va a añadir un vector que contendrá los índices de la posición de los *NaN* en la matriz V. Este, se computará a través de los comandos *isnan* y *find* y se denominará W.

Además de esto, para calcular la matriz de *WPs* se realizará una ligera modificación: puesto que las posiciones de los *NaN* son conocidas a través de W, no se empleará el comando *rmmissing* para hallarla. Se calculará una matriz de vértices V_0 sin *NaN* y se aplicará el comando *unique* para obtener *WPs*.

```
1 close all; clear all; clc;
2 
3 %% Insertar puntos de origen y destino (Bruselas y Sarajevo en este caso)
4 O = [50.901402, 4.484440]; D = [43.824600, 18.331499];
```

```
5
6 %% Insertar polígonos obstáculo
7
8 storms = load('TFG_Narciso.mat');
9 storms = struct2cell(storms);
10 V = [storms{1} storms{2}];
11
12 %% CÁLCULOS PRELIMINARES
13
14 V = [V; NaN(1,2); 0; NaN(1,2); D]*pi/180;
15
16 % Vector que contiene los índices de la matriz V donde se sitúan los NaN's
17 W = find(isnan(V(:,1)));
18
19 % Matriz de WPs y orden del grafo
20 V_0 = V; V_0(W,:) = []; % Primero se suprimen los NaN
21 WPs = unique(V_0,'rows','stable'); % Seguidamente se suprimen los vértices repetidos
22 N = length(WPs);
23
24 % Implementación de la transformación de Mercator
25 Lon_0 = mean(WPs(:,2));
26 x_V = (V(:,2) - Lon_0); y_V = log(tan(pi/4 + V(:,1)/2)); V_Mercator = [x_V, y_V];
27 x_WPs = (WPs(:,2) - Lon_0); y_WPs = log(tan(pi/4 + WPs(:,1)/2)); WPs_Mercator = ...
28     [x_WPs, y_WPs];
29
30 % Inicialización de la matriz de adyacencia lógica
31 Conex = false(N,N);
32 %
```

Código 2.4 Cálculos preliminares de la segunda versión del algoritmo *Ingenuo*.

Seguidamente, se ha de fijar el vértice i y formar la base no ortogonal. Las mayores dificultades para la implementación del criterio que se acaba de desarrollar se encuentran cuando el vértice i es, o bien el primer vértice del polígono al que pertenece, o bien el último. En un caso genérico, $i^+ = i+1$ e $i^- = i-1$, por lo que los vectores $\vec{r}$ y $\vec{s}$ se calcularán buscando las coordenadas de los nodos $i-1$, $i+1$ e i en la matriz $WPs_{Mercator}$ de la siguiente forma:

$$\vec{r} = WPs_{Mercator,(i+1)} - WPs_{Mercator,i} \tag{2.9}$$

$$\vec{s} = WPs_{Mercator,(i-1)} - WPs_{Mercator,i} \tag{2.10}$$

No obstante, en los dos casos especiales mencionados anteriormente es necesario realizar una corrección relacionada con el número de vértices del polígono en que se encuentra el nodo i. Con este objetivo, antes de fijar el vértice i se iniciará un contador, p, igual a 0 y que se incrementará en una unidad cuando se detecte que i es el primer vértice de un polígono, es decir, p indicará el polígono en que se encuentra el vértice i.

Teniendo en cuenta lo anterior, es posible calcular el número de vértices del polígono p, que responde a la siguiente expresión:

$$N_{vec_p} = \begin{cases} W_p - 2 & \text{si} \quad p = 1 \\ W_p - W_{p-1} - 2 & \text{si} \quad p > 1 \end{cases} \tag{2.11}$$

Donde W_p es la posición del *NaN* entre el polígono en que se encuentra el vértice i y el siguiente y W_{p-1} la del *NaN* entre el polígono en que se encuentra el vértice i y el anterior. Se restan dos unidades para no tener el cuenta ni el vértice repetido ni el *NaN* en el conteo de vértices del polígono (recuérdese que las posiciones de los *NaN* son obtenidas a partir de la matriz V).

Prosiguiendo, es posible identificar cuándo se da un caso degenerado y calcular los vectores de la base no ortogonal:

- Si i es el primer vértice de un polígono, se cumplirá que $i = W_p - 2p + 1$, donde W_p es, en este único caso, la posición del *NaN* entre el polígono en que se encuentra el vértice i y el anterior (se ha de tener presente que todavía no se ha incrementado p, esto se realiza tras detectar el primer vértice de un polígono como se muestra en el Código 2.5), para eliminar los vértices repetidos y *NaN* que se tienen en cuenta en el vector W se resta dos veces el número de polígonos por el que se ha transcurrido y se incrementa en una unidad para localizar el primer vértice del polígono en cuestión. El vector $\vec{r}$ se calculará de acuerdo con 2.9 y $\vec{s}$ a través de la siguiente la siguiente expresión:

$$\vec{s} = WPs_{Mercator,(i-1+N_{vec_p})} - WPs_{Mercator,i} \tag{2.12}$$

- Si i es el último vértice de un polígono, se cumplirá que $i = W_p - 2p$, donde W_p es la posición del *NaN* entre el polígono actual y el siguiente (se ha de tener presente que tras pasar por el primer vértice del polígono sí se incrementó p) y para eliminar los vértices repetidos y *NaN* que se tienen en cuenta en el vector W se resta dos veces el número de polígonos por el que se ha transcurrido. El vector $\vec{s}$ se calculará de acuerdo con 2.10 y $\vec{r}$ a través de la siguiente expresión:

$$\vec{r} = WPs_{Mercator,(i+1-N_{vec_p})} - WPs_{Mercator,i} \tag{2.13}$$

Para el desarrollo del algoritmo, una vez fijado el vértice i se deberá comprobar en primer lugar que este nodo no sea el punto de origen O (la condición implementada en MATLAB también comprobaría que este nodo no sea el punto de destino D, pero esto en realidad no es necesario pues $i \in [1, N-1]$), ya que el desarrollo anterior se realizó para vértices que pertenecían a algún polígono a evitar. Tras ello, se hallarán los vectores $\vec{r}$ y $\vec{s}$ y la componente no nula del producto vectorial de dichos vectores, q_z.

```
33 %% CÓMPUTO DEL GRAFO DE VISIBILIDAD
34 
35 p = 0; % Inicialización del contador asociado al número de polígonos
36 for i = 1 : N - 1
37     if i <= N - 2 % Para no formar la base no ortogonal con el punto de origen O
38 
39         % Formación de la base no ortogonal de vectores
40         if i == 1 || i == W(p) - 2*p + 1 % Caso en que i es el primer vértice del
41         % primer polígono o del resto de polígonos (téngase en cuenta que cuando p = 0,
42         % i = 1. Cuando se compruebe si i == 1, no se entrará a mirar si
43         % i == W(p) - 2*p + 1, por lo que no se no se evaluará W(p) cuando p = 0)
44             p = p + 1;
45             r = WPs_Mercator(i+1,:)' - WPs_Mercator(i,:)';
46 
47             if p == 1 % Caso en que i sea el primer vértice del primer polígono
48                 s = WPs_Mercator(i - 1 + W(p) - 2,:)' - WPs_Mercator(i,:)';
49             else % Caso en que i sea el primer vértice del resto de polígonos
50                 s = WPs_Mercator(i - 1 + W(p) - W(p-1) - 2,:)' - WPs_Mercator(i,:)';
```

```
51              end
52
53          elseif i == W(p) - 2*p % Caso en que i es el último vértice de algún polígono
54              if p == 1 % Caso en que i sea el último vértice del primer polígono
55                  r = WPs_Mercator(1, :)' - WPs_Mercator(i,:)';
56              else % Caso en que i sea el último vértice del resto de polígonos
57                  r = WPs_Mercator(i + 1 - (W(p) - W(p-1) - 2), :)' - WPs_Mercator(i,:)';
58              end
59              s = WPs_Mercator(i - 1,:)' - WPs_Mercator(i,:)';
60
61          else % Caso genérico
62              r = WPs_Mercator(i+1,:)' - WPs_Mercator(i,:)';
63              s = WPs_Mercator(i-1,:)' - WPs_Mercator(i,:)';
64          end
65
66          % Producto vectorial de r y s
67          q_z = r(1)*s(2) - r(2)*s(1);
68      end
69      %
```

Código 2.5 Formación de la base no ortogonal centrada en un nodo i en la segunda versión del algoritmo *Ingenuo*.

Seguidamente, se fija el vértice j y se crea una variable bandera denominada *vis*, que será igual a 1 (verdadero) si i y j son visibles y 0 (falso) en caso contrario. Tras ello, se vuelve a emplear el comando *polyxpoly* para comprobar si el número de puntos de intersección es dos. En caso de que esto se cumpla, se testea en primer lugar si el vértice i es el punto de origen O, en cuyo supuesto se tendría que i y j son visibles (es imposible que la recta que une O con otro vértice pase por el interior de un polígono si el número de puntos de intersección es dos). En caso contrario, se calcula el vector director de la recta que une i y j, $\vec{v}$, y se obtienenen los escalares a y b, que, junto al signo de q_z, permiten determinar si la recta que une i y j pasa por el interior de un polígono, cuyo caso contrario se almacenará en la matriz *Conex*. Para la aplicación de este criterio, se tendrán en cuenta 11 cifras significativas para los escalares a y b.

```
70      for j = i + 1 : N
71          vis = 0; % vis = 0 si los vértices no son mutuamente visibles y vis = 1 si se
72          % demuestra que son mutuamente visibles
73
74          [xk, yk] = polyxpoly([WPs_Mercator(i,1) WPs_Mercator(j,1)], ...
75              [WPs_Mercator(i,2) WPs_Mercator(j,2)], V_Mercator(:,1), ...
76              V_Mercator(:,2),'unique');
77
78          if length(xk) == 2
79              if i == N - 1
80                  vis = 1;
81              else
82
83                  % Cálculo del vector v
84                  v = WPs_Mercator(j,:)' - WPs_Mercator(i,:)';
85
86                  % Resolución del sistema de ecuaciones
87                  a = (v(1)*s(2) - v(2)*s(1))/q_z;
88                  b = (r(1)*v(2) - r(2)*v(1))/q_z;
89
90                  % Aplicación del criterio con una cierta tolerancia
91                  if sign(q_z) == -1 && ~(a > 1e-12 && b > 1e-12)
```

```
92                      vis = 1;
93                  elseif sign(q_z) == 1 && (a >= -1e-12 && b >= -1e-12)
94                      vis = 1;
95                  end
96              end
97          end
98          if vis == 1
99              Conex(i,j) = 1; Conex(j,i) = 1;
100         end
101     end
102 end
```

Código 2.6 Comprobación de la visibilidad de los vértices en la segunda versión del algoritmo *Ingenuo*.

Tras la obtención de *Conex*, el proceso seguido para el cómputo de la trayectoria óptima es exactamente el mismo que el descrito en la sección 2.2.3 y el Código 2.3.

En la Figura 2.9, se muestra un diagrama de flujo de la rutina desarrollada para la implementación del algoritmo *Ingenuo* cambiando el criterio del punto medio por el criterio desarrollado en la sección 2.3.1.

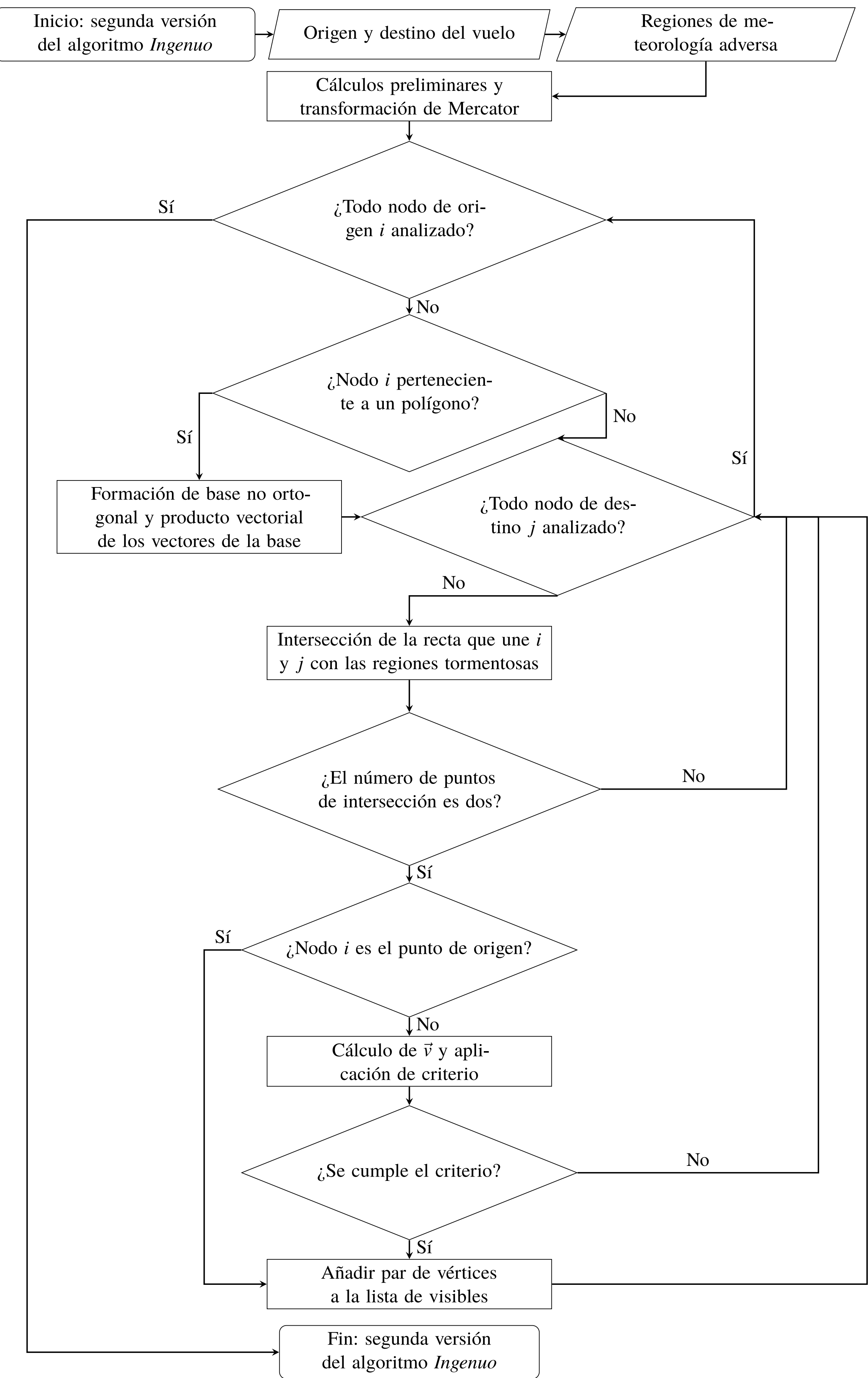


Figura 2.9 Diagrama de flujo de la segunda versión del algoritmo *Ingenuo*.

3 Algoritmo de *Lee*

Los códigos de los algoritmos desarrollados en la sección anterior presentan un importante inconveniente: pese a su sencillez, el costo computacional de los mismos cuando el número de nodos de la región comienza a aumentar es muy elevado. Consecuentemente, surge la necesidad de implementar rutinas que reduzcan este tiempo de computación.

En este capítulo se va a desarrollar un algoritmo para la generación del grafo de visibilidad basado en el método de *Lee*, quien expuso la primera solución no trivial al problema. Dicho resultado presenta, teóricamente, un costo computacional significativamente menor que el obtenido por el método *Ingenuo*.

3.1 Introducción al método de *Lee*

El algoritmo para la generación del grafo de visibilidad va a seguir un enfoque completamente distinto al desarrollado en el capítulo anterior. Es de gran importancia resaltar la influencia que ha tenido la obra de *M. de Berg, O. Cheong, M. van Kreveld* y *M. Overmars* [13] en esta sección. En primer lugar se fijará el vértice i para, tras ello, emplear otras rutinas que comprueben la visibilidad de este mismo vértice con respecto al resto de nodos del grafo. Se seguirá el procedimiento que se describe en el siguiente pseudocódigo:

Pseudocódigo 3.1 VISIBILITYGRAPH(V)

Entrada: La variable V, formada por una serie de obstáculos poligonales disjuntos definidos por un conjunto ordenado de vértices y dos puntos que no se encuentren en el interior de los polígonos.
Salida: Grafo de visibilidad dado en *Conex*.

```
1: Inicializar un grafo vacío Conex.
2: para todos los vértices i ∈ V hacer
3:     si i se encuentra en algún polígono obstáculo entonces
4:         Conex ← VISIBLEVERTICESVA(i,V)
5:     si no
6:         Conex ← VISIBLEVERTICESOD(i,V)
7: devolver Conex
```

Nótese que V es, en este caso, la concatenación de la variable V presentada en el Pseudocódigo 1.1 y los nodos de origen O y destino D.

En el método *Ingenuo* se comprobaba la intersección del segmento $\overline{ij}$ con todas las aristas obstáculo. En el caso del método de *Lee*, se plantea la posibilidad de usar la información que se obtiene cuando se testea un vértice j, de forma que se pueda comprobar la visibilidad de otros vértices más rápido. Para conseguir esto es necesario, en primer lugar, ordenar los vértices que se encuentran alrededor del nodo i en un sentido cíclico mientras que, a su vez, se va manteniendo información que ayudará a decidir acerca de la visibilidad del siguiente vértice a ser tratado.

Considérese una situación genérica como la mostrada en la Figura 3.1. En ella, se traza el segmento que va del nodo i al nodo j, $\overline{ij}$. El vértice j será visible si $\overline{ij}$ no interseca ninguna arista obstáculo antes de llegar a j. Para comprobar esto, en el método de *Lee* original se propone la construcción de un árbol de búsqueda binario en el que se almacenen las aristas intersecadas por $\overline{ij}$ denominado T, tal y como se muestra en la Figura 3.2b. Así, es posible comprobar si j se encuentra tras una arista obstáculo, vista desde i.

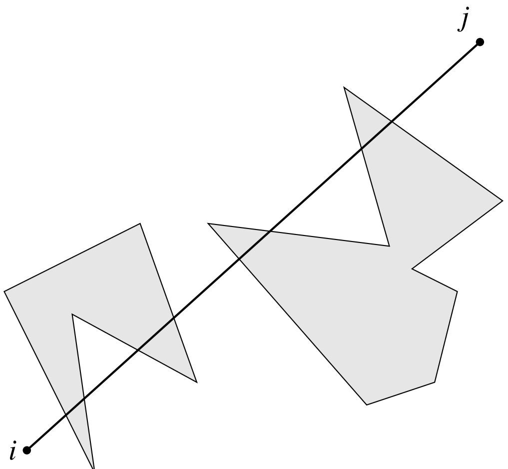
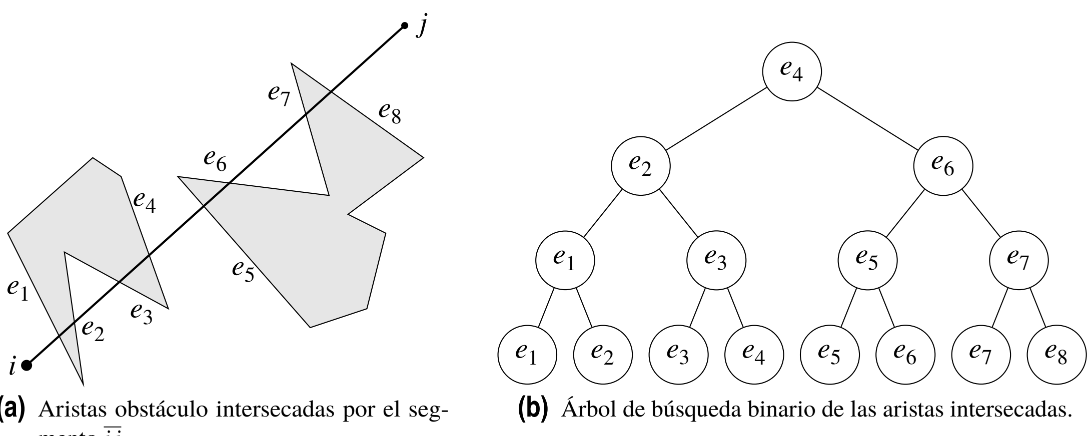


Figura 3.1 Trazado del segmento $\overline{ij}$.



(a) Aristas obstáculo intersecadas por el segmento $\overline{ij}$.

(b) Árbol de búsqueda binario de las aristas intersecadas.

Figura 3.2 Árbol de búsqueda binario de las aristas intersecadas.

A la vez que se tratan los vértices en un orden cíclico alrededor de i se van guardando las aristas obstáculo intersecadas por $\overline{ij}$ en T. Sus hojas las almacenan en el siguiente orden:

1. En los nodos inferiores, la hoja que se encuentra más a la izquierda almacena la primera arista obstáculo intersecada por $\overline{ij}$, la siguiente el segmento que es intersecado tras el primero y así

sucesivamente.

2. Los nodos interiores almacenan la arista obstáculo e_{k0} que se encuentra más a la derecha del subárbol izquierdo, de forma que todas las aristas que se encuentran en el subárbol derecho son intersecadas por $\overline{ij}$ tras e_{k0} y todas las que se encuentran en el subárbol izquierdo son intersecadas anteriormente a e_{k0} (excepto la propia e_{k0}).

No obstante en este trabajo, en lugar de emplear un árbol de búsqueda binario, se dispondrá el almacenamiento de las aristas en un conjunto ordenado, de forma que los segmentos del árbol mostrado en la Figura 3.2b quedarían en T de acuerdo con 3.1. Así, para encontrar una determinada arista, se empleará un algoritmo de *Búsqueda binaria*.

$$T = \begin{Bmatrix} e_1 \\ e_2 \\ e_3 \\ e_4 \\ e_5 \\ e_6 \\ e_7 \\ e_8 \end{Bmatrix} \tag{3.1}$$

Los algoritmos VisibleVerticesVA(i,V) y VisibleVerticesOD(i,V) desarrollan lo descrito anteriormente. La función VisibleVerticesVA(i,V) no tendrá en cuenta los nodos aislados, es decir, se empleará para comprobar la visibilidad de los nodos pertenecientes a polígonos obstáculo alrededor del vértice i cuando este pertenece a algún polígono obstáculo; a través de esta función se halla lo que se denominará "autografo" de visibilidad. La función VisibleVerticesOD(i,V) se empleará para comprobar la visibilidad de todos los nodos alrededor del vértice i cuando este es, o bien el nodo de origen O o bien el de destino D; se empleará para introducir estos puntos en el "autografo" de visibilidad.

Puesto que es necesario realizar una inicialización del conjunto T, se va a trazar una semirrecta denominada σ desde el vértice i en el sentido positivo del eje x como se muestra en la Figura 3.3, de forma que las aristas intersecadas por σ se almacenarán en T.

Tras lo anterior, se irán visitando los vértices alrededor del nodo i en orden cíclico. Si dos o más vértices se encuentran en la misma dirección vistos desde el nodo i, lógicamente se tomarán primero los vértices cuya distancia a i sea menor pues el planteamiento de la visibilidad se ha de realizar sobre estos vértices de acuerdo con lo expuesto en la sección 2.2.2 y la Figura 2.1.

Seguidamente, se ha de determinar la visibilidad de cada vértice. En caso de que no haya arista alguna en T, el vértice j_{j0} será visible desde i y, en caso contrario, se emplearán las rutinas denominadas VisibleVA(i,j_{j0},e_1) y VisibleOD(i,j_{j0},e_1) que decidirán acerca de la visibilidad del nodo j_{j0} respecto al vértice i. Finalmente, se actualizará el conjunto ordenado T tratando las aristas incidentes a j_{j0}.

El algoritmo VisibleVerticesVA(i,V) viene descrito en el Pseudocódigo 3.2.

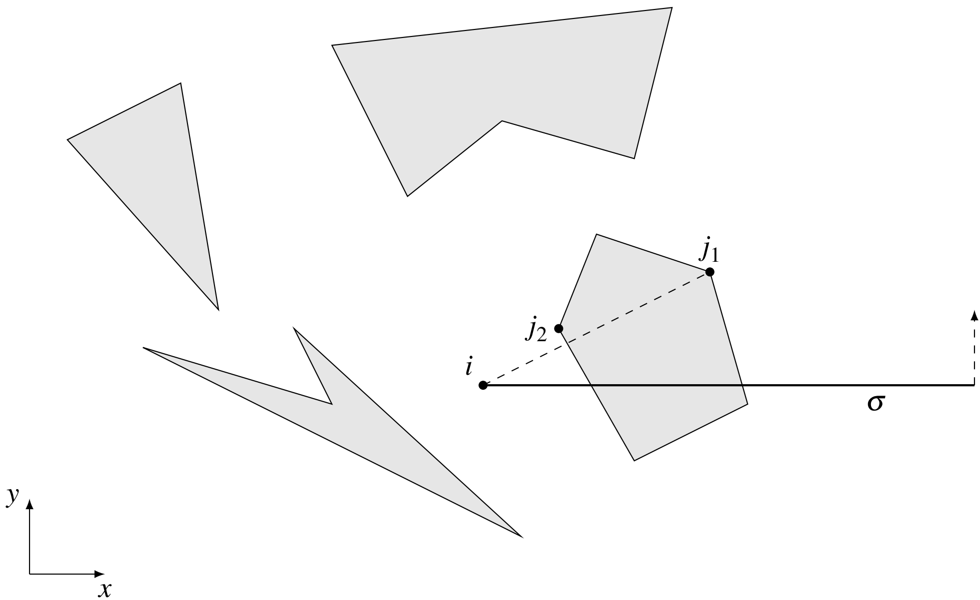


Figura 3.3 Trazado de la semirrecta σ.

Pseudocódigo 3.2 VisibleVerticesVA(i,V)

Entrada: Una serie de obstáculos poligonales disjuntos definidos por un conjunto ordenado de vértices, V, y un punto i no aislado que no se encuentra en el interior de ningún obstáculo.

Salida: El conjunto de todos los nodos visibles desde i.

1: Ordenar los nodos del grafo en sentido cíclico alrededor del vértice i a partir del sentido positivo del eje x. En caso de empates, los nodos más cercanos a i se dispondrán en primer lugar. Sea j_1, j_2,...,j_{n-1} la lista ordenada.
2: Sea σ la semirrecta trazada desde i en el sentido positivo del eje x. Encontrar las aristas obstáculo intersecadas por σ y almacenarlas en un conjunto ordenado T en el orden en que son intersecadas por σ.
3: **para** $j0 \leftarrow 1$ **a** $n-1$ **hacer**
4:     **si** T está vacío **entonces**
5:         Añadir $\overline{i j_{j0}}$ a Conex
6:     **si no**
7:         **si** VisibleVA(i, j_{j0}, e_1) **entonces**
8:             Añadir $\overline{i j_{j0}}$ a *Conex*.
9:     Insertar en T las aristas incidentes al vértice j_{j0} que se encuentran en el lado antihorario de la semirrecta σ.
10:     Eliminar de T las aristas incidentes al vértice j_{j0} que se encuentran en el lado horario de la semirrecta σ.
11: **devolver** *Conex*

Nótese que V es, en este caso, la misma variable V presentada en el Pseudocódigo 1.1.

El algoritmo VisibleVerticesOD(i,V) se rige por el mismo pseudocódigo que el que se muestra para la función VisibleVerticesVA(i,V) con las siguientes salvedades:

- En adición a los obstáculos poligonales, recibe como entrada los puntos de origen O y destino D (es decir, en esta función, V será la concatenación de la variable V presentada en el Pseudocódigo 1.1 y los nodos de origen O y destino D).
- El punto i sería, en este caso, un nodo aislado.
- En lugar de emplear la función VISIBLEVA(i,j_{j0},e_1) para decidir acerca de la visibilidad de vértices desde el nodo i, se empleará una función denominada VISIBLEOD(i,j_{j0},e_1).

Sumado a ello, las funciones VISIBLEVERTICESVA(i,V) y VISIBLEVERTICESOD(i,V) presentarán otras diferencias en lo referente a su implementación en MATLAB que se detallarán más adelante.

La rutina VISIBLEVA(i,j_{j0},e_1) decide si un cierto vértice j_{j0} es visible desde el nodo i cuando este pertenece a alguno de los polígonos obstáculo. En primer lugar se ha de comprobar si el segmento $\overline{ij_{j0}}$ atraviesa el interior del polígono del que i es un vértice, en cuyo caso los nodos no serían mutuamente visibles. En caso contrario, se realiza la intersección de la primera arista almacenada en T, es decir, e_1 con el segmento $\overline{ij_{j0}}$. Si no hay intersección o si la intersección es justamente el vértice j_{j0}, este nodo e i serán mutuamente visibles. Lo explicado se recoge en el Pseudocódigo 3.3.

Pseudocódigo 3.3 VISIBLEVA(i,j_{j0},e_1)

Entrada: El vértice j_{j0} sobre el que se plantea la visibilidad.
Salida: **Verdadero** o **Falso**.

1: **si** $\overline{ij_{j0}}$ atraviesa el interior del polígono del que i es un vértice **entonces**
2:     **devolver falso**
3: **si no**
4:     **si** $\overline{ij_{j0}}$ interseca e_1 y la intersección no es el propio j_{j0} **entonces**
5:         **devolver falso**
6:     **si no**
7:         **devolver verdadero**

Por su parte, la rutina VISIBLEOD(i,j_{j0},e_1) decide si un cierto vértice j_{j0} es visible desde el nodo i cuando este es el punto de origen O o el de destino D. La diferencia que presenta con respecto a la función VISIBLEVA(i,j_{j0},e_1) radica en que no es necesario comprobar si $\overline{ij_{j0}}$ atraviesa el interior del polígono del que i es un vértice, pues en este caso los nodos i están aislados.

En cuanto a la complejidad característica de las funciones VISIBLEVERTICESVA(i,V) y VISIBLEVERTICESOD(i,V), la primera línea del pseudocódigo 3.2 presentaría una complejidad de $O[(n-1)\log_2(n-1)] \sim O(N\log_2 N)$ para la función VISIBLEVERTICESVA(i,V) y una de $O[(N-1)\log_2(N-1)] \sim O(N\log_2 N)$ para la función VISIBLEVERTICESOD(i,V), por lo que la ordenación de vértices presentaría la misma complejidad para ambas funciones. La segunda línea del pseudocódigo 3.2 presenta una de $O(n) \sim O(N)$ para las dos funciones, por lo que no se vería incrementada la complejidad total del algoritmo, y tanto la búsqueda en T como su actualización representa, en el peor de los casos, un costo computacional del $O(\log_2 n) \sim O(\log_2 N)$ que, puesto que se realiza para todo nodo alrededor del vértice i, lleva a una complejidad de $O[(n-1)\log_2 N] \sim O(N\log_2 N)$ para la función VISIBLEVERTICESVA(i,V) y una de $O[(N-1)\log_2 N] \sim O(N\log_2 N)$ para la función VISIBLEVERTICESOD(i,V). Por ende, la complejidad de la rutinas VISIBLEVERTICESVA(i,V) y VISIBLEVERTICESOD(i,V) es de $O(N\log_2 N)$.

Puesto que la función VISIBLEVERTICESVA(i,V) se ejecuta para cada vértice perteneciente a polígonos obstáculos, es decir, n veces y VISIBLEVERTICESOD(i,V) se ejecuta para los nodos

de inicio y final, es decir, 2 veces, el algoritmo de *Lee* presentaría una complejidad total de $O(nN\log_2 N + 2N\log_2 N) \sim O(N^2\log_2 N)$.

En la siguiente sección, se detalla el desarrollo del algoritmo planteado así como su implementación en MATLAB.

3.2 Desarrollo e implementación del algoritmo

El algoritmo completo será resultado de la disposición de una serie de códigos: un programa principal que abarcará los algoritmos descritos en los pseudocódigos 1.1 y 3.1 y, por otro lado, las respectivas funciones VISIBLEVERTICESVA(i,V), VISIBLEVERTICESOD(i,V), VISIBLEVA(i,j_{j0},e_1) y VISIBLEOD(i,j_{j0},e_1). A continuación, se desarrollan los algoritmos planteados.

3.2.1 Programa principal

Al igual que en el desarrollo del método *Ingenuo*, como entradas al algoritmo, se tendrán el punto de origen del vuelo, O, el punto de destino, D, y una matriz de vértices de la región tormentosa, V.

Cálculos preliminares

Se obtiene el vector W, se creará la matriz WPs, se obtendrá el número de nodos N y se implementará la transformación de Mercator de acuerdo a lo descrito en las secciones 2.2.1 y 2.3.2.

En adición a lo anterior, se calculará el número de puntos dado en la matriz V, N_{in} y se computará el número de aristas obstáculo del grafo, n. Además, para la ordenación de los vértices en sentido antihorario descrita en la sección anterior se creará un vector denominado *index*, el cual almacenará los índices de los puntos del grafo que se ordenarán una vez fijado el nodo i.

```
1  clear all; close all; clc;
2
3  %% Insertar puntos de origen y destino (Puntos artificiales en este caso)
4  O = [43, -0.86]; D = [44.7, 6.75];
5
6  %% Insertar polígonos obstáculo
7
8  storms = load('TFG_Narciso.mat');
9  storms = struct2cell(storms);
10 V = [storms{1} storms{2}];
11
12 %% CÁLCULOS PRELIMINARES
13
14 V = [V; NaN(1,2); O ; NaN(1,2); D]*pi/180; N_in = length(V);
15
16 % Vector que contiene los índices de la matriz V donde se sitúan los NaN's y obtención
17 % del número de polígonos obstáculo del grafo
18 W = find(isnan(V(:,1)));
19 npol = length(W) - 1;
20
21 % Matriz de WPs, orden del grafo y número de aristas obstáculo del grafo
22 V_0 = V; V_0(W,:) = []; % Primero se suprimen los NaN
23 WPs = unique(V_0, 'rows', 'stable'); % Seguidamente se suprimen los vértices repetidos
24 N = length(WPs); n = N - 2;
```

```
25
26 % Implementación de la transformación de Mercator
27 Lon_0 = mean(WPs(:,2));
28 x_V = (V(:,2) - Lon_0); y_V = log(tan(pi/4 + V(:,1)/2)); V_Mercator = [x_V, y_V];
29 x_WPs = (WPs(:,2) - Lon_0); y_WPs = log(tan(pi/4 + WPs(:,1)/2)); WPs_Mercator = ...
30     [x_WPs, y_WPs];
31
32 % Vector de índices de los puntos del grafo
33 index = 1:N;
34 %
```

Código 3.1 Cálculos preliminares del programa principal del algoritmo de *Lee*.

Por otra parte, para el desarrollo del algoritmo es interesante crear una variable que, conociendo el índice de una determinada arista obstáculo, informe acerca de cuáles son los vértices extremos de dicha arista. Esta variable se denominará E y será una matriz de dimensiones $N_{in} \times 2$. En los casos en los que la pareja de vértices consecutivos incluya un *NaN*, se considerarán dos *NaN* en esa fila de E. El empleo de estos índices resulta de interés en la inicialización del conjunto ordenado T cada vez que se llame a la función $\textsc{VisibleVertices}(i,V)$, ya que, al obtener mediante el comando *polyxpoly* los índices de los segmentos de polilínea intersecados, estos se corresponden con las filas de la matriz E según el planteamiento empleado.

Análogamente a lo anterior, es conveniente crear una variable que, conociendo el índice de un determinado vértice, informe acerca de cuáles son las aristas obstáculo incidentes a ese nodo. Esta variable se denominará *Einv* y será una matriz de dimensiones $N \times 2$. De igual modo, se almacenarán los mismos índices de arista que los tenidos en cuenta para la matriz E.

Puesto que la arista k será la contenida entre los vértices j y j^+ (vértice posterior a j), las matrices E y *Einv* presentarán la siguiente estructura:

$$E = \begin{matrix} \text{Fila} \\ \vdots \\ k^- \\ k \\ \vdots \end{matrix} \begin{bmatrix} \vdots & \vdots \\ j^- & j \\ j & j^+ \\ \vdots & \vdots \end{bmatrix} \qquad Einv = \begin{matrix} \text{Fila} \\ \vdots \\ j \\ j^+ \\ \vdots \end{matrix} \begin{bmatrix} \vdots & \vdots \\ k & k^- \\ k^+ & k \\ \vdots & \vdots \end{bmatrix} \tag{3.2}$$

Para el cálculo de estas matrices en MATLAB, se hallará el número de polígonos del grafo *npol* y se irá visitando cada polígono e introduciendo la información pertinente en dichas matrices a partir del vector W. Teniendo en cuenta que p es el índice del polígono del cual se quiere introducir información, el primer vértice de dicho polígono será $W_{p-1} - 2(p-1) + 1$ (a la posición del *NaN* entre el polígono actual y el anterior se resta el doble del número de polígonos por el que se ha pasado sin contar el actual y se suma una unidad para obtener el primer vértice del polígono) y el último será $W_p - 2p$ (a la posición del *NaN* entre el polígono actual y el siguiente se resta el doble número de polígonos por el que se ha pasado).

Por otra parte, la primera arista de un determinado polígono será $W_{p-1} + 1$ (a la posición del *NaN* entre el polígono actual y el anterior se suma una unidad) y la última, $W_p - 2$ (a la posición del *NaN* entre el polígono actual y el siguiente se restan dos unidades). Además, se recuerda que N_{vec_p} es el número de vértices del polígono p.

Teniendo en cuenta lo anterior, los elementos de la matriz E presentarán las siguientes expresiones:

- Si $p = 1$:

$$\begin{aligned} E_{k,1} &= \; k \quad \text{si} \quad k \le N_{vec_1} \\ E_{k,2} &= \begin{cases} k+1 & \text{si} \quad k < N_{vec_1} \\ 1 & \text{si} \quad k = N_{vec_1} \end{cases} \\ E_{k,1} = E_{k,2} &= \; NaN \quad \text{si} \quad N_{vec_1} < k \le N_{vec_1} + 2 \end{aligned} \tag{3.3}$$

- Si $1 < p \le npol$:

$$\begin{aligned} E_{k,1} &= \; k - 2(p-1) \quad \text{si} \quad W_{p-1} + 1 \le k \le W_p - 2 \\ E_{k,2} &= \begin{cases} k - 2(p-1) + 1 & \text{si} \quad W_{p-1} + 1 \le k < W_p - 2 \\ W_{p-1} - 2(p-1) + 1 & \text{si} \quad k = W_p - 2 \end{cases} \\ E_{k,1} = E_{k,2} &= \; NaN \quad \text{si} \quad W_p - 2 < k \le W_p \end{aligned} \tag{3.4}$$

Por otra parte, los elementos de la matriz *Einv* tendrán las siguientes expresiones:

- Si $p = 1$:

$$\begin{aligned} Einv_{j,1} &= \; j \quad \text{si} \quad j \le W_p - 2 \\ Einv_{j,2} &= \begin{cases} N_{vec_1} & \text{si} \quad j = 1 \\ j - 1 & \text{si} \quad 1 < j \le W_p - 2 \end{cases} \end{aligned} \tag{3.5}$$

- Si $1 < p \le npol$:

$$\begin{aligned} Einv_{j,1} &= \; j + 2(p-1) \quad \text{si} \quad W_{p-1} - 2(p-1) + 1 \le j \le W_p - 2p \\ Einv_{j,2} &= \begin{cases} j + 2(p-1) + N_{vec_p} - 1 & \text{si} \quad j = W_{p-1} - 2(p-1) + 1 \\ j + 2(p-1) & \text{si} \quad W_{p-1} - 2(p-1) + 1 < j \le W_p - 2p \end{cases} \end{aligned} \tag{3.6}$$

Por último, se inicializará una matriz de adyacencia lógica *Conex* como se realizó para los métodos *Ingenuo*.

```
35 % Cálculo de las matrices E y Einv
36 E = NaN(length(V),2); Einv = zeros(N,2);
37 for p = 1:npol
38     if p == 1
39         E(1: W(p) - 2,:) = [1 : W(p) - 2; 2 : W(p) - 2, 1]';
40         Einv(1: W(p) - 2,:) = [1 : W(p) - 2; W(p) - 2, 1 : W(p) - 3]';
41     else
42         E(W(p-1) + 1: W(p) - 2,:) = [W(p-1) + 1 - 2*(p-1) : W(p) - 2*p; ...
43             W(p-1) + 2 - 2*(p-1) : W(p) - 2*p, W(p-1) + 1 - 2*(p-1)]';
44         Einv(W(p-1) - 2*(p-1) + 1 : W(p) - 2*p,:) = [W(p-1) + 1 : W(p) - 2; ...
45             W(p) - 2, W(p-1) + 1 : W(p) - 3]';
46     end
47 end
48 
49 % Inicialización de la matriz de adyacencia lógica
50 Conex = false(N,N);
51 %
```

Código 3.2 Cálculo de las matrices E y *Einv* en el programa principal del algoritmo de *Lee*.

Cómputo del grafo de visibilidad

Tras las operaciones preliminares, es hora de implementar la generación del grafo de visibilidad. Para ello, se empleará un bucle *for* en el que se irán fijando los nodos i; cuando i pertenezca a polígonos obstáculo se llamará a la función VISIBLEVERTICESVA(i,V), encargada de la generación del "autografo" de visibilidad y, en caso contrario, se llamará a la función VISIBLEVERTICESOD(i,V), encargada de la introducción de los puntos de origen O y destino D en el "autografo" de visibilidad.

```
52 %% CÓMPUTO DEL GRAFO DE VISIBILIDAD
53
54 for i = 1:N
55     if i <= n
56         Conex = VisibleVerticesVA(i, V_Mercator(1:N_in - 4,:), WPs_Mercator(1:n,:), ...
57             index(1:n), E, Einv, Conex, N_in);
58     else
59         Conex = VisibleVerticesOD(i, V_Mercator, WPs_Mercator, N, index, E, Einv, ...
60             Conex, N_in);
61     end
62 end
63 %
```

Código 3.3 Cómputo del grafo de visibilidad en el programa principal del algoritmo de *Lee*.

Obtención del camino más corto

El proceso seguido para la obtención de la trayectoria más corta es exactamente el mismo que el descrito en la sección 2.2.3 y el Código 2.3.

En la Figura 3.4, se muestra un diagrama de flujo del programa principal para la implementación del grafo de visibilidad a través del algoritmo de *Lee*.

3.2.2 Funciones VISIBLEVERTICESVA(i,V) y VISIBLEVERTICESOD(i,V)

En primer lugar, se va a desarrollar la función VISIBLEVERTICESVA(i,V), empleada para la generación del "autografo" de visibilidad. La función para la introducción de los puntos de origen O y destino D al "autografo", VISIBLEVERTICESOD(i,V), se programará a partir de la anterior.

La implementación de la función se dividirá en tres partes:

1. Ordenación de los vértices en sentido antihorario a partir del sentido positivo del eje x.

2. Inicialización del conjunto ordenado T.

3. Comprobación de la visibilidad de los vértices y actualización de T.

Ordenación de los vértices en sentido antihorario a partir del sentido positivo del eje x

La primera operación que debe realizar la función es obtener los índices de los puntos que se almacenarán en j (nótese que, en esta ocasión, j no es un único nodo, sino que será un vector de nodos) y se ordenarán alrededor del vértice i.

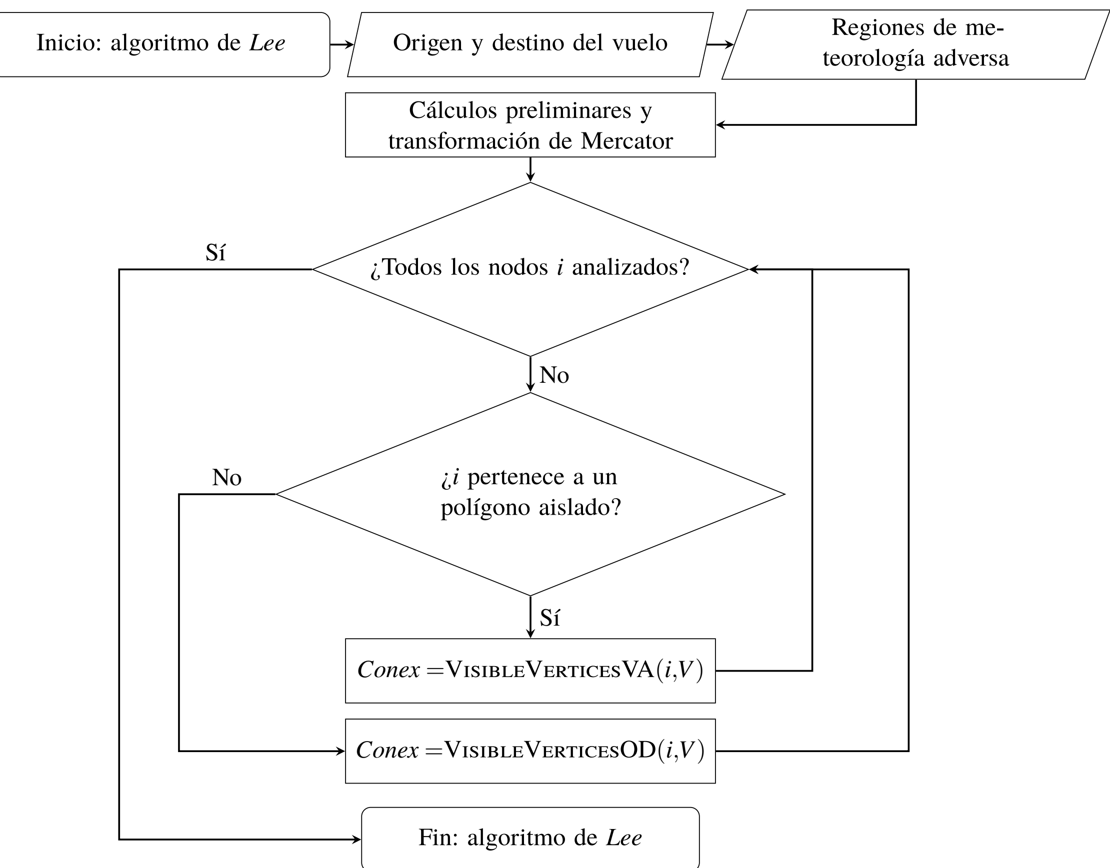


Figura 3.4 Diagrama de flujo del programa principal del algoritmo de *Lee*.

Para evitar calcular ángulos y minimizar operaciones, la ordenación de vértices en sentido antihorario a partir del sentido positivo del eje x se realizará a través de las coordenadas de los distintos nodos. Para ello, se introducirán tanto las coordenadas de los nodos almacenadas en la matriz $WPs_{Mercator}$ como los índices de dichos vértices, que se encuentran en el vector *index*, en una variable matricial denominada *VerticesData*, que presentará la siguiente estructura:

$$VerticesData = \begin{bmatrix} x_1 & y_1 & j_1 \\ x_2 & y_2 & j_2 \\ \vdots & \vdots & \vdots \\ x_n & y_n & j_n \end{bmatrix} \tag{3.7}$$

En la matriz anterior se van a eliminar, en primer lugar, la fila correspondiente al índice del nodo i y, seguidamente, puesto que se va a aprovechar el carácter simétrico del problema, solo se considerarán los nodos que se encuentren en el semiplano $y - i_y \geq 0$, por lo que se suprimirán las filas correspondientes a los índices de los vértices que no cumplan esta condición.

```
1 function Conex = VisibleVerticesVA(i, V_Mercator, WPs_Mercator, N, index, E, Einv, ...
2     Conex, N_in)
3
4 %% ORDENACIÓN DE VÉRTICES EN SENTIDO ANTIHORARIO ALREDEDOR DEL NODO i
5
6 VerticesData = [WPs_Mercator, index'];
```

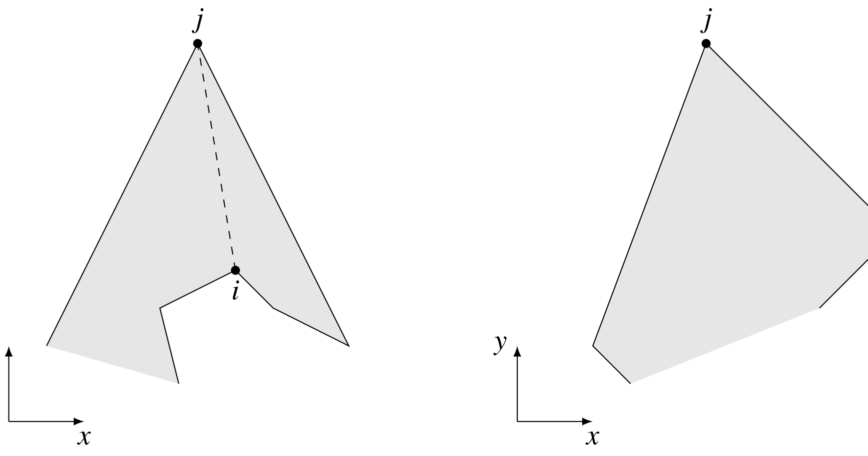
```
7 VerticesData(i,:) = [];
8 VerticesData((VerticesData(:,2) - WPs_Mercator(i,2)) < 0,:) = [];
9 %
```

Código 3.4 Operaciones previas a la ordenación de vértices en sentido antihorario de la función VisibleVerticesVA(i,V).

Tras ello, es hora de realizar la ordenación de los vértices restantes en sentido antihorario pero, en primer lugar, cabe preguntarse cuándo es necesario realizar esta ordenación. Se pueden dar tres casos:

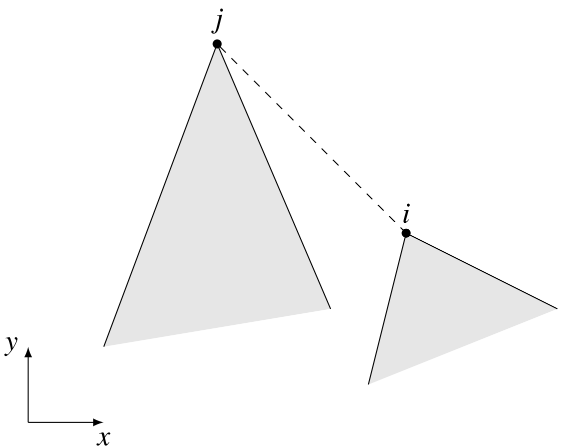
- Si el número de vértices restante es nulo, es decir, si la matriz *VerticesData* está vacía, no hay vértices a tratar y el algoritmo finalizaría.
- Si el número de vértices restante es mayor que la unidad, es necesario recorrer el algoritmo completo.
- Si el número de vértices restante es exactamente uno, es decir, si el número de filas de la matriz *VerticesData* es igual a la unidad, no es necesario ordenar los vértices. Tampoco es necesario emplear el algoritmo general para determinar la visibilidad de ese nodo, ya que se puede aprovechar el hecho de que no pueden existir aristas que corten el segmento $\overline{ij}$. Con todo, para determinar la visibilidad de los nodos i y j hay que atender al caso de que se trate (ver Figura 3.5):
  - Caso en que i y j son vértices no consecutivos de un mismo polígono (Figura 3.5a). En este caso, i y j serán visibles si el segmento $\overline{ij}$ no atraviesa el interior del polígono.
  - Caso en que i y j son vértices consecutivos de un mismo polígono (Figura 3.5b). En este caso, i y j serán siempre visibles.
  - Caso en que i y j son vértices de distintos polígonos (Figura 3.5c). En este caso, i y j serán siempre visibles.

  En conclusión, para este caso i y j serán visibles si el segmento $\overline{ij}$ no atraviesa el interior de un polígono. Con el propósito de determinar esto, se aplicará el criterio basado en las aristas adyacentes al vértice i desarrollado en la sección 2.3.1. Para la búsqueda de los vértices i^+ e i^- se emplearán las matrices E y $Einv$, de forma que el procedimiento será buscar en primer lugar en la matriz $Einv$ el índice de las aristas que van del vértice i a los nodos i^+ e i^-, para luego buscar en la matriz E los índices de los vértices entre los que se encuentran dichas aristas y computar las coordenadas de estos nodos a través de la matriz $WPs_{Mercator}$. De esta forma, se tendrán en cuenta tanto los casos genéricos como los casos degenerados.



(a) Caso en que i y j son vértices no consecutivos de un mismo polígono.

(b) Caso en que i y j son vértices consecutivos de un mismo polígono.



(c) Caso en que i y j son vértices de distintos polígonos.

Figura 3.5 Casos en que solo existe un vértice por encima del nodo i.

```
10 % Caso en el que solo queda un vértice restante, para el que se comprobará directamente
11 % la visibilidad
12 if length(VerticesData(:,3)) == 1
13
14     j = VerticesData(3);
15
16     % Cálculo del vector v
17     v = (WPs_Mercator(j, :) - WPs_Mercator(i, :))';
18
19     % Formación de la base no ortogonal de vectores
20     r = (WPs_Mercator(E(Einv(i,1), 2),:) - WPs_Mercator(i,:))';
21     s = (WPs_Mercator(E(Einv(i,2), 1),:) - WPs_Mercator(i,:))';
22
23     % Producto vectorial de r y s
24     q_z = r(1)*s(2) - r(2)*s(1);
25
26     % Resolución del sistema de ecuaciones
27     a = (v(1)*s(2) - v(2)*s(1))/q_z;
28     b = (r(1)*v(2) - r(2)*v(1))/q_z;
29
```

```
30     % Aplicación del criterio con una cierta tolerancia
31     if (sign(q_z) == -1 && ~(a>1e-12 && b>1e-12)] || (sign(q_z) == 1 && ...
32             (a>=-1e-12 && b>=-1e-12))
33         Conex(i,j) = 1; Conex(j,i) = 1;
34     end
35 %
```

Código 3.5 Tratamiento del vértice en caso de que solo quede uno tras la aplicación de simetría en la función VisibleVerticesVA(i,V).

Siguiendo con el algoritmo, para realizar la ordenación de vértices en sentido antihorario a partir del sentido positivo del eje x, es necesario emplear un algoritmo de ordenación. En el caso de este trabajo, se hará uso del método *HeapSort* el cual garantiza que, en el peor de los casos, la complejidad de la ordenación será de $O[(n-1)\log_2(n-1)] \sim O(N\log_2 N)$. La implementación de este método así como las condiciones de permutación de los distintos nodos se encuentran en el Apéndice B.1.

Tras la ordenación, se extraerá el conjunto j donde estarán los índices de los vértices que se han ordenado alrededor del nodo i en sentido antihorario a partir del sentido positivo del eje x.

```
36 % Caso en que queda más de un vértice, para el que se recorrerá el algoritmo completo
37 elseif length(VerticesData(:,3)) > 1
38
39     %% ORDENACIÓN DE VÉRTICES EN SENTIDO ANTIHORARIO
40     VerticesData= HeapSort_v1(VerticesData, WPs_Mercator(i,:));
41     j = VerticesData(:,3);
42     %
```

Código 3.6 Ordenación de vértices en sentido antihorario de la función VisibleVerticesVA(i,V).

Debido a la extensión de la función VisibleVertices(i,V), se van a realizar varios diagramas de flujo. En la Figura 3.6 se muestra un diagrama de flujo de lo que sería la función completa. Más adelante se desarrollarán diagramas de flujo tanto para la inicialización de T como para la comprobación de visibilidad de los vértices y actualización de T.

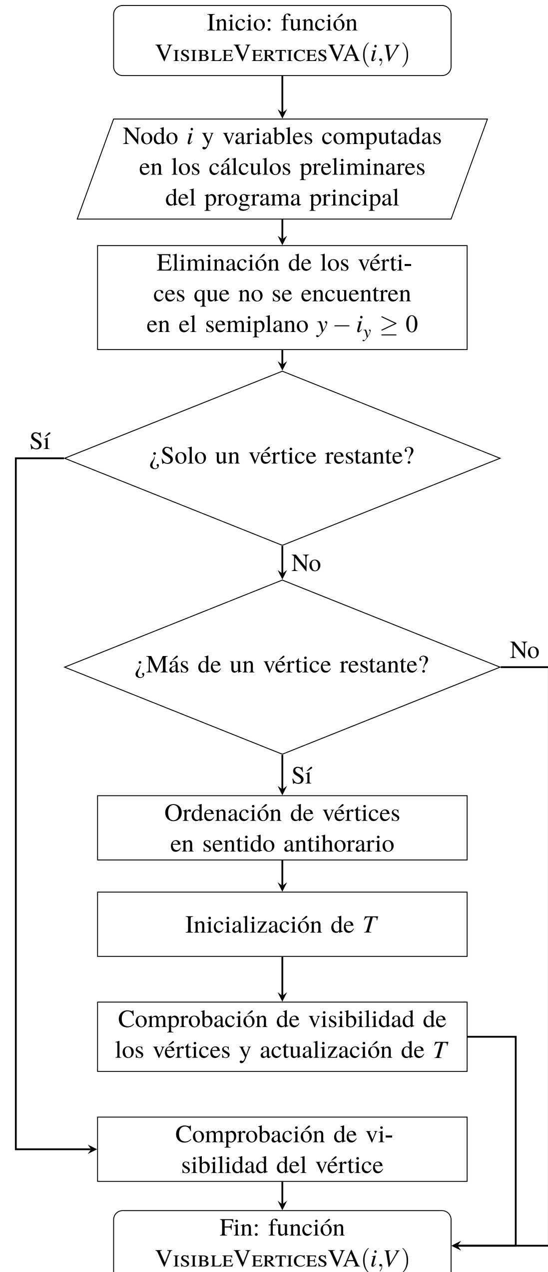


Figura 3.6 Diagrama de flujo de la función VisibleVerticesVA(i,V).

Inicialización del conjunto ordenado T

En esta sección del algoritmo, se realizarán las siguientes tareas individuales:

1. Determinar las intersecciones de la semirrecta σ con las aristas de los polígonos obstáculo.
2. Analizar si las intersecciones son vértices.
3. Decidir qué aristas se deben introducir en T
4. Ordenar T

En primer lugar, se traza la semirrecta σ y se realiza su intersección con todas las aristas de los polígonos obstáculo. Ante la imposibilidad de llevar una semirrecta al infinito en MATLAB

se trazará un segmento denominado $\overline{\sigma}$ desde las coordenadas del nodo i, (i_x,i_y) hasta el punto de coordenadas $(i_x+2\pi,i_y)$ de forma que se abarquen todas las intersecciones posibles. Teniendo presente que la coordenada x representa un incremento de longitud de acuerdo con 2.1, 2π constituye una cota superior, de manera que no haya ninguna intersección que no sea detectada porque el segmento $\overline{\sigma}$ sea demasiado corto.

Como se mencionó anteriormente, se empleará el comando *polyxpoly*. En este caso, se le pedirán tres salidas: los vectores *xk* e *yk*, que almacenan las coordenadas de los puntos de intersección, y la matriz K, en cuya primera columna se encuentran los índices de los segmentos intersecados de la primera polilínea (puesto que $\overline{\sigma}$ se introduce como un único segmento suficientemente grande y, por tanto, no está formada por una concatenación de segmentos diferentes, estos valores serán siempre iguales a la unidad), y en cuya segunda columna se encuentran los índices de los segmentos intersecados de la segunda polilínea, es decir, los índices de las aristas obstáculo intersecadas.

Téngase en cuenta que no es posible tomar $K_{:,2}=T$ puesto que, en primer lugar, en K no vendrán los índices de las aristas ordenados debidamente y, en caso de que una de las intersecciones sea un vértice, solo detectará una arista intersecada, pero en este supuesto es necesario decidir tanto si se debe introducir esta arista como tratar acerca de la otra arista incidente a dicho vértice. Este problema se desarrollará más adelante.

Seguidamente, se inicializa una serie de conjuntos para almacenar información acerca de las aristas de intersección: el ya mencionado conjunto ordenado T, un conjunto que contendrá las distancias de los puntos de intersección al vértice i denominado ρ, y un último conjunto *Tinv* que tendrá dimensiones $N_{in}\times 1$ y almacenará la posición en T que ocupan los índices de las aristas (en las filas correspondientes a las aristas que no se encuentran almacenadas en T aparecerá un *NaN*).

$$T = \begin{matrix}\text{Fila}\\ \vdots\\ pos_k\\ \vdots\end{matrix}\begin{Bmatrix}\vdots\\ k\\ \vdots\end{Bmatrix}; \quad \rho = \begin{matrix}\text{Fila}\\ \vdots\\ pos_k\\ \vdots\end{matrix}\begin{Bmatrix}\vdots\\ \rho_k\\ \vdots\end{Bmatrix}; \quad Tinv = \begin{matrix}\text{Fila}\\ \vdots\\ k\\ \vdots\end{matrix}\begin{Bmatrix}\vdots\\ pos_k\\ \vdots\end{Bmatrix} \tag{3.8}$$

Por otra parte, también se crea un conjunto denominado x_{comp}, que será de utilidad cuando se deba decidir acerca del orden de las aristas obstáculo para el caso en que el punto de intersección sea un vértice y se deban introducir las dos aristas incidentes a ese nodo como se verá más adelante.

```
43     %% INICIALIZACIÓN DEL CONJUNTO ORDENADO T
44
45     % Intersección del segmento sigmagorro con todos los polígonos
46     [xk, yk, K] = polyxpoly([WPs_Mercator(i,1), WPs_Mercator(i,1) + 2*pi], ...
47         [WPs_Mercator(i,2), WPs_Mercator(i,2)], V_Mercator(:,1), V_Mercator(:,2));
48
49     % Inicialización de estructuras para introducir el conjunto ordenado T, distancia
50     % de las aristas en T al punto i en rho, posición de las aristas de T en Tinv y
51     % la componente x del vector característico de las aristas en x_comp
52     T = zeros(0,1); rho = zeros(0,1); Tinv = NaN(N_in,1); x_comp = zeros(0,1);
53     %
```

Código 3.7 Intersección del segmento $\overline{\sigma}$ y establecimiento de conjuntos en la inicialización del conjunto ordenado T en la función VISIBLEVERTICESVA(i,V).

Tras lo anterior, se definirá un índice $k0$ que recorrerá cada una de las intersecciones por separado, para las que se realizarán una serie de comprobaciones:

1. En primer lugar, se comprueba si el punto de intersección correspondiente a la arista intersecada que se considere no es el propio vértice i, pues las aristas incidentes a este nodo no representan un obstáculo, por lo que no serán almacenadas en T. Se comprueba únicamente la coordenada en el eje x del punto de intersección, pues las coordenadas en el eje y para todos los puntos de intersección son iguales.
2. A continuación, se extrae la arista obstáculo que se ha intersecado de la segunda columna de la matriz K, es decir, la arista k (puesto que el comando *polyxpoly* detecta las aristas que van del vértice j al j^+, nodo posterior al vértice j, aunque la intersección sea el propio vértice j) y se comprueba tanto si la intersección mostrada es un vértice como si es un vértice repetido (en caso de dar con un vértice cuyas coordenadas aparecen duplicadas en la matriz V, estas también aparecerán duplicadas en $[xk,yk]$, en este caso, al recorrer esta última matriz, en la primera pasada se trataría como un vértice y es en la segunda cuando es necesario detectar que está repetido):
   a) Para comprobar si la intersección es un vértice, puesto que el índice de la arista k se corresponde con el índice de la posición en que se encuentran las coordenadas del nodo j en la matriz $V_{Mercator}$, se comprueba si estas coordenadas son iguales a las del punto de intersección, en cuyo caso se concreta que la intersección es un vértice.
   b) Como se ha mencionado anteriormente, en el caso de que la intersección sea un vértice repetido, aparecen en los vectores xk e yk las coordenadas de dicho vértice dos veces: en la primera pasada, se detecta que la intersección es un vértice y en la segunda se detecta si dicho vértice está repetido. Para comprobar esto último, en la segunda columna de la matriz K aparecerá una arista de intersección, que, en este único caso, es la que va del vértice repetido j a j^- (nodo anterior al vértice j) y, puesto que en este supuesto ese índice de arista se corresponde con el índice de la posición en que se encuentran las coordenadas del nodo j^- en la matriz $V_{Mercator}$, se busca en dicha matriz las coordenadas en la posición de dicha arista pero incrementada en una unidad y se comprueba si estas coordenadas son iguales a las del punto de intersección, en cuyo caso se concreta que la intersección es un vértice repetido (Figura 3.7).

En el caso de la intersección de vértices, solo se considerarán las situaciones en que no se da un vértice repetido, pues para ese caso se detectará antes que la intersección es un vértice y se realizarán las operaciones pertinentes.

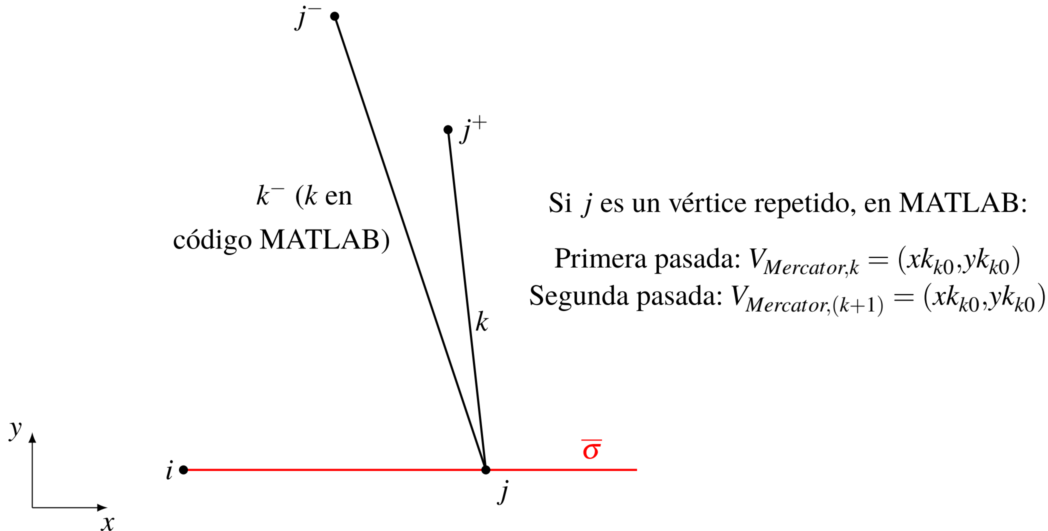


Figura 3.7 Caso en que la intersección del segmento $\overline{\sigma}$ con los obstáculos poligonales es un vértice y está repetido.

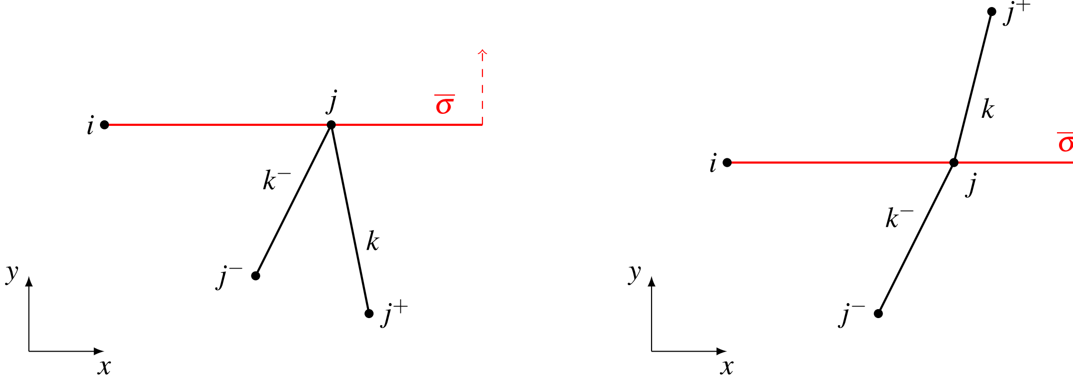
```
54     % Bucle para tratar las aristas que se deban insertar en T
55     for k0 = 1 : length(xk)
56
57         % Primera comprobación: que el punto de intersección no sea el punto i del que
58         % parte el segmento sigmagorro
59         if xk(k0) ~= WPs_Mercator(i,1)
60
61             % Arista de intersección
62             k = K(k0,2);
63
64             % Segunda comprobación: si la intersección es justamente un vértice o si es
65             % un vértice repetido
66             isvertice = 0; % Variable que se igualará a 1 si se detecta el caso de
67             % vértice
68             isverticerep = 0; % Variable que se igualará a 1 si se detecta el caso de
69             % vértice repetido
70
71             if V_Mercator(k,1) == xk(k0) && V_Mercator(k,2) == yk(k0)
72                 isvertice = 1;
73             elseif V_Mercator(k+1,1) == xk(k0) && V_Mercator(k+1,2) == yk(k0)
74                 isverticerep = 1;
75             end
76             %
```

Código 3.8 Comprobación de las intersecciones en la inicialización del conjunto ordenado T en la función VisibleVerticesVA(i,V).

Siguiendo a las comprobaciones realizadas, se debe decidir qué aristas se han de almacenar en T:

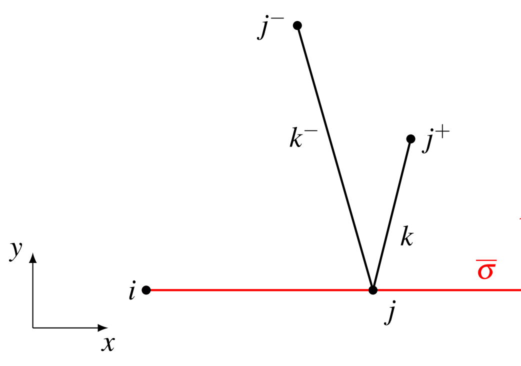
- En caso de que la intersección no sea ni un vértice ni un vértice repetido, se almacenará directamente el índice de la arista obstáculo intersecada en T y su distancia al vértice i en ρ. En el conjunto x_{comp} se introducirá un NaN (pues nunca será necesario mirar este conjunto para ordenar esa arista).

- Cuando la intersección sea un vértice, se pueden dar tres casos:
  - Si las dos aristas obstáculo incidentes al vértice j se encuentran en el lado horario del segmento $\overline{\sigma}$, no se deberá insertar en T ninguna de ellas, pues estas no van a representar un obstáculo (Figura 3.8a).
  - Si una de las aristas obstáculo incidentes al vértice j se encuentra en el lado horario del segmento $\overline{\sigma}$ y la otra se encuentra en el lado antihorario, se deberá insertar en T únicamente esta última pues es la que va a representar un obstáculo (Figura 3.8b).
  - Si las dos aristas obstáculo incidentes al vértice j se encuentran en el lado antihorario del segmento $\overline{\sigma}$, se deberán insertar ambas en T pues estas van a representar un obstáculo (Figura 3.8c).



(a) Aristas obstáculo en el lado horario de $\overline{\sigma}$.

(b) Una arista obstáculo en el lado horario de $\overline{\sigma}$ y otra en el lado antihorario.



(c) Aristas obstáculo en el lado antihorario de $\overline{\sigma}$.

Figura 3.8 Casos en que uno de los puntos de intersección de $\overline{\sigma}$ con los obstáculos poligonales en la inicialización de T es un vértice.

Para determinar si se debe insertar o no una arista en T, se seguirán los siguientes procedimientos:

- En el caso de la arista k^- (índice de la arista anterior a k), se trazará el vector $\vec{t}$, dirigido del vértice j al j^-, se normalizará (aunque para esta comprobación no sea necesario) siendo el vector resultado $\vec{t_0}$ y se mirará la componente en el eje y de dicho vector, $t_{0,y}$. En caso de que esta sea positiva, la arista k^- se añadirá a T. Además, se introducirá la distancia del punto de intersección al vértice i en el conjunto ρ y, por si se da el caso de

que se deba introducir también la arista k, para desempatar en cuanto a la ordenación de las aristas en T se introducirá la componente en el eje x de $\vec{t_0}$, $t_{0,x}$ en el conjunto x_{comp}.

Para proceder en MATLAB, será necesario conocer el índice de la arista obstáculo k^-. Puesto que se conoce el índice de la arista k, se computará el de k^- del siguiente modo: en primer lugar, se buscará en la matriz E el índice j del nodo extremo de la arista k para luego, buscar en la matriz $Einv$ el índice de la arista k^- incidente a dicho nodo j. De esta forma, se tendrán en cuenta los casos degenerados.

Tras ello, puesto que el índice k se corresponde con el índice de la posición en que se encuentran las coordenadas del nodo j en la matriz $V_{Mercator}$ y el índice k^- con el de la posición de las coordenadas del nodo j^-, se hallará el vector $\vec{t}$ a partir de la matriz $V_{Mercator}$ en esta ocasión.

– En el caso de la arista k, se trazará el vector $\vec{u}$, dirigido del vértice j al j^+, y se realizarán operaciones análogas a las ejecutadas en el caso anterior.

  Nótese que, en este caso, para proceder en MATLAB al cálculo del vector $\vec{u}$ con la matriz $V_{Mercator}$, no es necesario buscar el índice de la arista k^+ pues, aunque este se corresponda con el caso degenerado de que sea la primera arista de un polígono, en la matriz $V_{Mercator}$ los primeros vértices de cada polígono vienen repetidos, por lo que se puede computar $\vec{u}$ directamente.

  * Si es necesario introducir tanto k^- como k en T, ocupará una posición menor la que presente menor componente en el eje x de su vector característico normalizado.

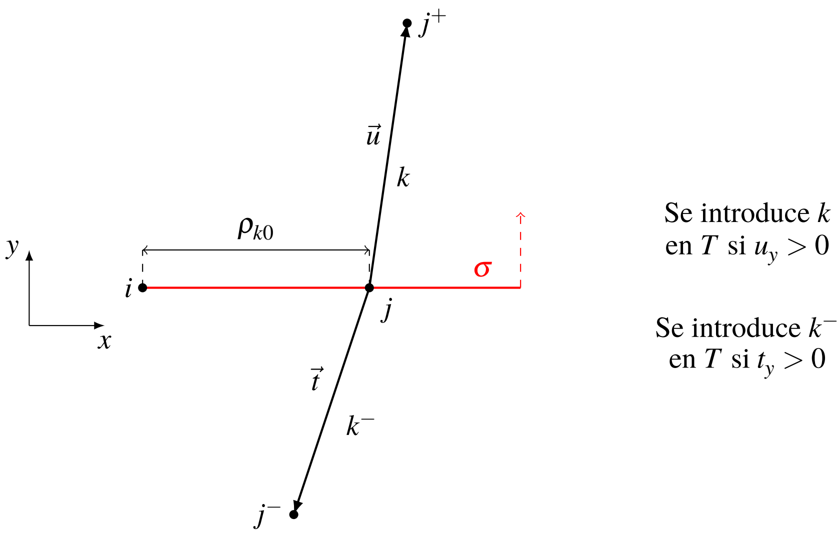


Figura 3.9 Vectores $\vec{u}$ y $\vec{t}$ y distancia al vértice i.

```
77            if isvertice == 1
78                % Caso en que la intersección es un vértice
79                kminus = Einv(E(k,1),2);
80
81                % Cálculo del vector t
82                t = V_Mercator(kminus,:) - V_Mercator(k,:); t_0 = t/norm(t);
83
84                % Para incluir la arista o no en T, se observa la componente en el eje
85                % y del vector t
```

```
86                 if t_0(2) > 0
87
88                     % Introducción de la arista correspondiente en T
89                     T = [T; kminus];
90
91                     % Cálculo de la distancia de la arista al punto i
92                     rho_k0 = xk(k0) - WPs_Mercator(i,1);
93                     rho = [rho; rho_k0];
94
95                     % Almacenamiento de la componente x del vector t normalizado por si
96                     % se da el caso de que haya que desempatar en cuanto a la
97                     % ordenación en T
98                     x_comp = [x_comp; t_0(1)];
99                 end
100
101                % Las operaciones con el vector u son análogas a las realizadas con el
102                % vector t
103                u = V_Mercator(k + 1,:) - V_Mercator(k,:); u_0 = u/norm(u);
104                if u_0(2) > 0
105                    T = [T; k];
106                    rho_k0 = xk(k0) - WPs_Mercator(i,1);
107                    rho = [rho; rho_k0];
108                    x_comp = [x_comp; u_0(1)];
109                end
110
111            elseif isverticerep ~= 1
112                % La intersección ya no puede ser un vértice
113                T = [T; k];
114                rho_k0 = xk(k0) - WPs_Mercator(i,1);
115                rho = [rho; rho_k0];
116                x_comp = [x_comp; NaN]; % Puesto que en este caso no será necesario
117                % desempatar en cuanto a la ordenación en T, se introduce un NaN
118            end
119        end
120    end
121    %
```

Código 3.9 Almacenamiento de aristas en la inicialización del conjunto ordenado T en la función VISIBLEVERTICESVA(i,V).

Posteriormente a la inserción de las aristas en T, es necesario realizar la ordenación de las mismas, donde irán primero las que presenten menor distancia al nodo i y, en caso de empates, se mirará en el conjunto x_{comp} las componentes en el eje x de los vectores característicos de las aristas normalizados.

Para la ordenación, se creará una matriz *TreeData* donde se introduzcan los parámetros nombrados con el fin de volver a emplear el método *HeapSort* pero, en este caso, puesto que las condiciones de permutación son distintas a las consideradas para la ordenación de vértices en sentido antihorario a partir del sentido positivo del eje x, se empleará una segunda versión de la función donde se implementen los nuevos supuestos de permutación. Esta se muestra en el Apéndice B.2.

$$TreeData = \begin{bmatrix} e_1 & \rho_1 & x_{comp,1} \\ e_2 & \rho_2 & x_{comp,2} \\ \vdots & \vdots & \vdots \\ e_{n_T} & \rho_{n_T} & x_{comp,n_T} \end{bmatrix} \tag{3.9}$$

Téngase en cuenta que, si el número total de aristas en T es n_T, la ordenación en este caso supone un costo computacional de $O(n_T \log_2 n_T)$ no considerado anteriormente pero, ni en el peor de los escenarios, esto supone un aumento de la complejidad del algoritmo completo.

Por último, tras realizar la ordenación, se está en disposición de rellenar el conjunto *Tinv* con la posición de las aristas que aparecen en T, finalizando esta etapa del algoritmo.

```
122     TreeData = [T, x_comp, rho];
123
124     % Ordenación a través del método HeapSort
125     TreeData = HeapSort_v2(TreeData);
126     T = TreeData(:,1); rho = TreeData(:,3);
127
128     % Relleno del conjunto Tinv
129     for y = 1:length(T)
130         Tinv(T(y)) = y;
131     end
132     %
```

Código 3.10 Ordenación del conjunto ordenado T en la función VisibleVerticesVA(i,V).

En la Figura 3.11, se muestra un diagrama de flujo de las operaciones a realizar para la inicialización del conjunto T en la función VisibleVerticesVA(i,V).

Comprobación de la visibilidad de los vértices y actualización de T

Se comprobará la visibilidad de los vértices en el supuesto de que $j0 = 1$ o en caso de que $j0 > 1$ y el segmento $\overline{ij_{j0}}$ no se encuentre en la misma dirección que el segmento $\overline{ij_{j0-1}}$, de forma que no se considerarán las situaciones redundantes expuestas en la sección 2.2.2 y la Figura 2.1. En este supuesto, se tiene la situación genérica presentada en la Figura 3.10:

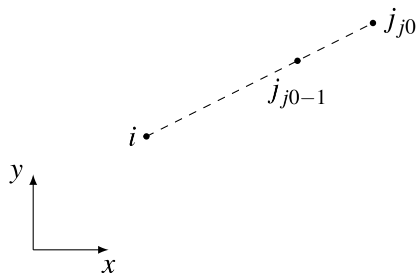


Figura 3.10 Caso en que los vértices i, j_{j0-1} y j_{j0} se encuentran alineados.

Para obtener una condición que implique que dichos vértices están alineados, se va a considerar el vector dirigido de i a j_{j0-1}, $\overrightarrow{ij_{j0-1}}$ y el vector director del segmento $\overline{ij}$, $\vec{v}$. Si estos vectores van en la misma dirección, el producto vectorial $\overrightarrow{ij_{j0-1}} \times \vec{v}$ será nulo:

$$\overrightarrow{ij_{j0-1}} \times \vec{v} = \begin{vmatrix} \vec{e}_x & \vec{e}_y & \vec{e}_z \\ ij_{j0-1,x} & ij_{j0-1,y} & 0 \\ v_x & v_y & 0 \end{vmatrix} = \begin{Bmatrix} 0 \\ 0 \\ ij_{j0-1,x}v_y - ij_{j0-1,y}v_x \end{Bmatrix} = \begin{Bmatrix} 0 \\ 0 \\ 0 \end{Bmatrix} \tag{3.10}$$

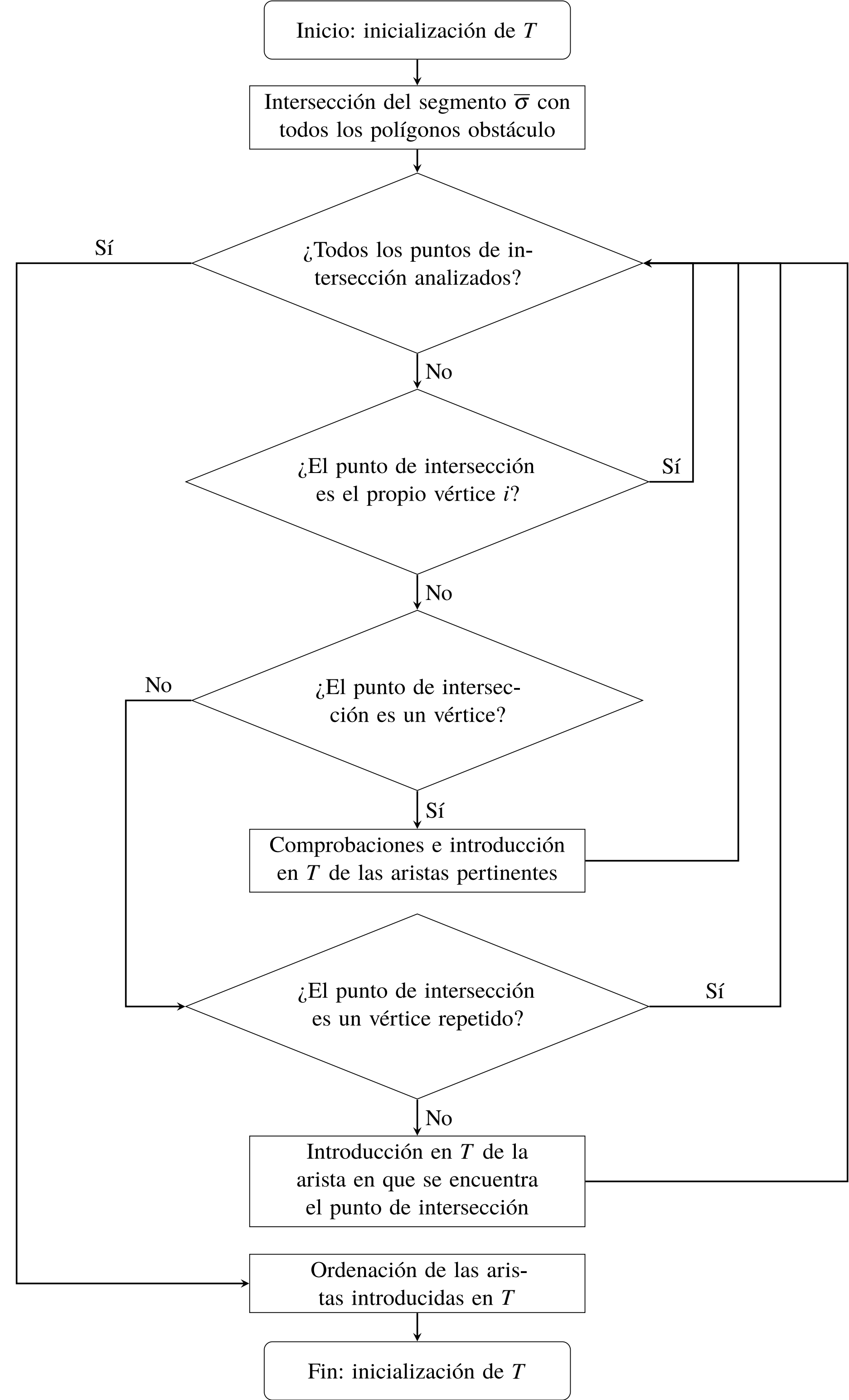


Figura 3.11 Diagrama de flujo de la inicialización de T en la función VisibleVerticesVA(i,V).

Por tanto, se obtiene que la condición para que los segmentos $\overline{ij_{j0}}$ y $\overline{ij_{j0-1}}$ no se encuentren alineados será pues $ij_{j0-1,x}v_y - ij_{j0-1,y}v_x \neq 0$.

Tras lo anterior, se realizará la comprobación de si T está vacío, en cuyo caso i y j_{j0} serán mutuamente visibles, lo cual se indicará a través de la variable bandera *vis* nuevamente. En caso contrario, la visibilidad de dichos vértices se realizará a través de la función VisibleVA(i, j_{j0}, e_1), que recibirá los argumentos necesarios para su funcionamiento. En caso de obtener vértices mutuamente visibles, se almacenará dicha información en la matriz *Conex*.

```
133     %% COMPROBACIÓN DE LA VISIBILIDAD DE LOS VÉRTICES
134 
135     for j0 = 1:length(j)
136         if j0  == 1 || (j0 > 1 && (WPs_Mercator(j(j0-1),1) - WPs_Mercator(i,1))* ...
137                 (WPs_Mercator(j(j0),2) - WPs_Mercator(i,2)) ~= ...
138                 (WPs_Mercator(j(j0),1) - WPs_Mercator(i,1))* ...
139                 (WPs_Mercator(j(j0-1),2) - WPs_Mercator(i,2))) % No se considerarán
140                 % casos redundantes
141 
142             if isempty(T)
143                 vis = 1;
144             else
145                 vis = VisibleVA(j(j0), i, WPs_Mercator, T(1), E, Einv);
146             end
147             if vis == 1
148                 Conex(i,j(j0)) = 1; Conex(j(j0),i) = 1;
149             end
150         end
151         %
```

Código 3.11 Comprobación de visibilidad de los vértices en la función VisibleVerticesVA(i, V).

Por último, se realizará la actualización de T. Se comprobará que el segmento $\overline{i j_{j0}}$ no vaya dirigido según el sentido positivo del eje x, pues en caso contrario no hay nada que actualizar porque es lo que se ha realizado en la inicialización de T a través de la intersección del segmento $\overline{\sigma}$ con todos los polígonos. La condición que no se ha de cumplir para que $\overline{i j_{j0}}$ no vaya dirigido según el sentido positivo del eje x es $j_{j0,y} = i_y$ y $j_{j0,x} > i_x$.

Por otra parte, también se testeará si el vértice j_{j0} no es el último en la lista ordenada ya que, en este supuesto, no es necesario actualizar T debido a que no se va a comprobar la visibilidad de más vértices.

Acto seguido, se buscarán en la matriz *Einv* cuáles son las aristas incidentes al vértice j_{j0}, k y k^-, así como la posición que ocupan dichas aristas en el árbol T, pos_k y pos_{k^-} (en caso de que las aristas no aparezcan en T, estos valores serán *NaN*).

```
152         if ~(WPs_Mercator(j(j0), 2) == WPs_Mercator(i,2) && WPs_Mercator(j(j0),1) ...
153                 > WPs_Mercator(i,1)) && j0 < length(j) % Para alpha=0 ya se ha
154                 % realizado la inicialización de T y para el último vértice no es
155                 % necesario actualizar % T pues no se comprobará la visibilidad
156                 % de ningún otro nodo
157 
158             %% ACTUALIZACIÓN DE T
159 
160             % Búsqueda de la existencia en T tanto de la arista k como de la arista
161             % k^{-}
162             k = Einv(j(j0),1); kminus = Einv(j(j0),2);
```

```
163            pos_k = Tinv(k); pos_kminus = Tinv(kminus);
164            %
```

Código 3.12 Búsqueda de las aristas almacenadas en T en la función VisibleVerticesVA(i,V).

A partir de ahora, se explicarán los distintos casos posibles de actualización de T, ordenados de menor a mayor dificultad:

- **Caso en que sea necesario eliminar ambas aristas en T** (Figura 3.12a)

  En este supuesto, $pos_k \neq NaN$ y $pos_{k^-} \neq NaN$. En primer lugar, se cambiará la posición que ocupan en el conjunto T, la cual viene dada en *Tinv*, por un *NaN*. Tras ello, será necesario realizar una corrección en *Tinv* de la posición de las aristas situadas tras k y k^- en T, a la que habrá que restar dos unidades (se suprimen dos aristas) y, por último, se eliminarán las aristas de T así como sus distancias al punto i de ρ. En 3.11 se muestra un ejemplo de estas operaciones:

$$Tinv = \begin{matrix} \text{Fila} \\ \vdots \\ k \\ k^- \\ \vdots \\ e_{k0} \\ \vdots \end{matrix} \left\{ \begin{matrix} \\ \vdots \\ \cancel{pos_k} = NaN \\ \cancel{pos_{k^-}} = NaN \\ \vdots \\ \cancel{pos_{e_{k0}}} = pos_{e_{k0}} - 2 \\ \vdots \end{matrix} \right\} \qquad T = \begin{matrix} \text{Fila} \\ \vdots \\ pos_{k^-} \\ pos_k \\ \vdots \\ \cancel{pos_{e_{k0}}} = pos_{e_{k0}} - 2 \\ \vdots \end{matrix} \left\{ \begin{matrix} \vdots \\ \cancel{k^-} \\ \cancel{k} \\ \vdots \\ e_{k0} \\ \vdots \end{matrix} \right\}; \tag{3.11}$$

```
165            % Caso en que sea necesario eliminar ambas aristas en T
166            if ~isnan(pos_k) && ~isnan(pos_kminus)
167
168                Tinv([k; kminus]) = NaN;
169                if pos_kminus > pos_k
170                    Tinv(T(pos_kminus + 1:length(T))) = Tinv(T(pos_kminus + ...
171                        1:length(T))) - 2;
172                else
173                    Tinv(T(pos_k + 1:length(T))) = Tinv(T(pos_k + 1:length(T))) - 2;
174                end
175                T([pos_k; pos_kminus]) = [];
176                rho([pos_k; pos_kminus]) = [];
177            %
```

Código 3.13 Caso en que sea necesario eliminar dos aristas en la actualización de T en la función VisibleVerticesVA(i,V).

- **Caso en que sea necesario eliminar una arista e introducir otra** (Figura 3.12b)
  - Si $pos_k \neq NaN$ y $pos_{k^-} = NaN$, se deberá eliminar de T la arista k y añadir la arista k^- en la misma posición en que se encuentra la arista k, eso sí, siempre que k^- no se encuentre en la misma dirección que la línea de escaneo, pues en este caso no representaría un obstáculo. Con el objeto de comprobar esto, se calcularán el vector director normalizado del segmento $\overline{ij}_{j0}$, $\vec{v_0}$, y el vector normalizado dirigido del vértice j al j^-, $\vec{t_0}$, y se comprobará si $\vec{v_0} = \vec{t_0}$ o bien si $\vec{v_0} = -\vec{t_0}$.

* Si se cumple lo anterior, se cambiará la posición que ocupa la arista k en el conjunto T dada en *Tinv* por un *NaN*. Tras ello, será necesario realizar una corrección en *Tinv* de la posición de las aristas situadas tras k en T, a la que se requerirá restarle una unidad (se suprime una arista) y, por último, se eliminará la arista k de T y su distancia al punto i de ρ. En 3.12 se muestra un ejemplo de estas operaciones:

$$Tinv = \begin{matrix} \text{Fila} \\ \vdots \\ k \\ \vdots \\ e_{k0} \\ \vdots \end{matrix} \left\{ \begin{matrix} \vdots \\ \cancel{pos_k} = NaN \\ \vdots \\ \cancel{pos_{e_{k0}}} = pos_{e_{k0}} - 1 \\ \vdots \end{matrix} \right\} \qquad T = \begin{matrix} \text{Fila} \\ \vdots \\ pos_k \\ \vdots \\ \cancel{pos_{e_{k0}}} = pos_{e_{k0}} - 1 \\ \vdots \end{matrix} \left\{ \begin{matrix} \vdots \\ \cancel{k} \\ \vdots \\ e_{k0} \\ \vdots \end{matrix} \right\}; \tag{3.12}$$

* Si lo anterior no se da, se cambiará la posición que ocupa la arista k en el conjunto T dada en *Tinv* por un *NaN* y se modificará la posición que ocupa la arista k^- en el conjunto T dada en *Tinv* por la que ocupaba la arista k, pos_k. Por último, se cambiará en T la arista k por la arista k^-. En 3.13 se muestra un ejemplo de estas operaciones:

$$Tinv = \begin{matrix} \text{Fila} \\ \vdots \\ k \\ k^- \\ \vdots \\ e_{k0} \\ \vdots \end{matrix} \left\{ \begin{matrix} \vdots \\ \cancel{pos_k} = NaN \\ \cancel{NaN} = pos_k \\ \vdots \\ pos_{e_{k0}} \\ \vdots \end{matrix} \right\} \qquad T = \begin{matrix} \text{Fila} \\ \vdots \\ pos_k \\ \vdots \\ pos_{e_{k0}} - 1 \\ \vdots \end{matrix} \left\{ \begin{matrix} \vdots \\ \cancel{k} = k^- \\ \vdots \\ e_{k0} \\ \vdots \end{matrix} \right\}; \tag{3.13}$$

– Si pos_k =*NaN* y $pos_{k^-} \neq$*NaN* se daría un caso análogo al anterior.

```
178            % Casos en que sea necesario eliminar una arista e introducir otra (si esta
179            % no va en la dirección del segmento ij_{j0})
180            elseif ~isnan(pos_k) && isnan(pos_kminus)
181
182                % Vector director del segmento ij_{j0}
183                v = (WPs_Mercator(j(j0),:) - WPs_Mercator(i,:))'; v_0 = v/norm(v);
184
185                % Vector t
186                t = V_Mercator(kminus,:) - V_Mercator(k,:); t_0 = t/norm(t);
187
188                if (abs(t_0(1) - v_0(1)) < 1e-12 && abs(t_0(2) - v_0(2)) < 1e-12) ...
189                        || (abs(t_0(1) + v_0(1)) < 1e-12 && abs(t_0(2) + v_0(2)) ...
190                        < 1e-12) % Se ha aplicado una cierta tolerancia
191                    % Caso en que solo se suprime una arista
192                    Tinv(k) = NaN; Tinv(T(pos_k + 1:length(T))) = Tinv(T(pos_k + 1:...
193                        length(T))) - 1;
194                    T(pos_k) = [];
195                    rho(pos_k) = [];
196                else
```

```
197                     % Caso en que se sustituye una arista por la otra
198                     Tinv(k) = NaN; Tinv(kminus) = pos_k;
199                     T(pos_k) = kminus;
200                 end
201
202             % Caso análogo al anterior
203             elseif isnan(pos_k) && ~isnan(pos_kminus)
204                 v = (WPs_Mercator(j(j0),:) - WPs_Mercator(i,:))'; v_0 = v/norm(v);
205                 u = V_Mercator(k+1,:) - V_Mercator(k,:); u_0 = u/norm(u);
206
207                 if (abs(u_0(1) - v_0(1)) < 1e-12 && abs(u_0(2) - v_0(2)) < 1e-12) ...
208                         || (abs(u_0(1) + v_0(1)) < 1e-12 && abs(u_0(2) + v_0(2)) ...
209                         < 1e-12)
210                     Tinv(kminus) = NaN; Tinv(T(pos_kminus + 1:length(T))) = ...
211                         Tinv(T(pos_kminus + 1:length(T))) - 1;
212                     T(pos_kminus) = [];
213                     rho(pos_kminus) = [];
214                 else
215                     Tinv(kminus) = NaN; Tinv(k) = pos_kminus;
216                     T(pos_kminus) = k;
217                 end
218             %
```

Código 3.14 Casos en que sea necesario eliminar una arista e introducir otra en la actualización de T en la función VISIBLEVERTICESVA(i,V).

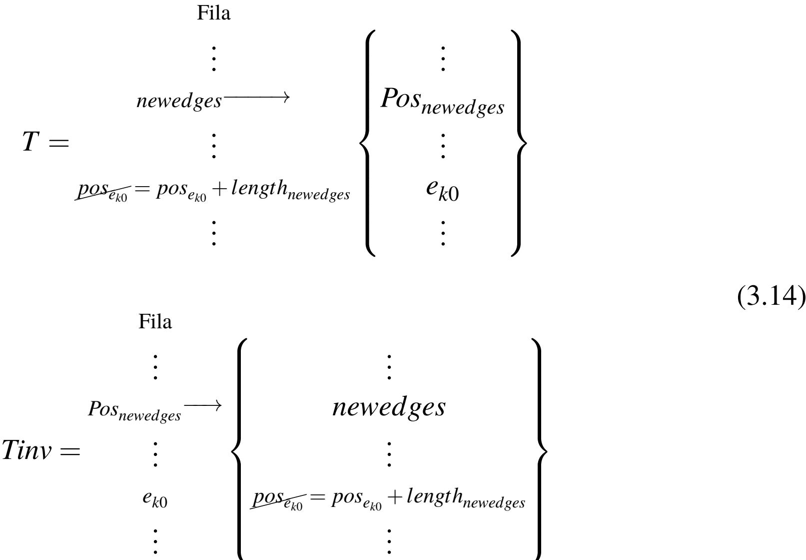
- **Caso en que sea necesario añadir dos aristas** (Figura 3.12c)

  Si $pos_k = NaN$ y $pos_{k^-} = NaN$, será necesario añadir dos aristas siempre que ninguna de ellas se encuentre en la misma dirección que la recta de escaneo. Se calcularán el vector director normalizado del segmento $\overline{ij_{j0}}$ (el dirigido de i a j_{j0}), $\vec{v}_0$, y los vectores característicos de las aristas normalizados, $\vec{u}_0$ y $\vec{t}_0$, y se comprobará que las dirección de estos dos últimos vectores no sea la misma que la de $\vec{v}_0$.

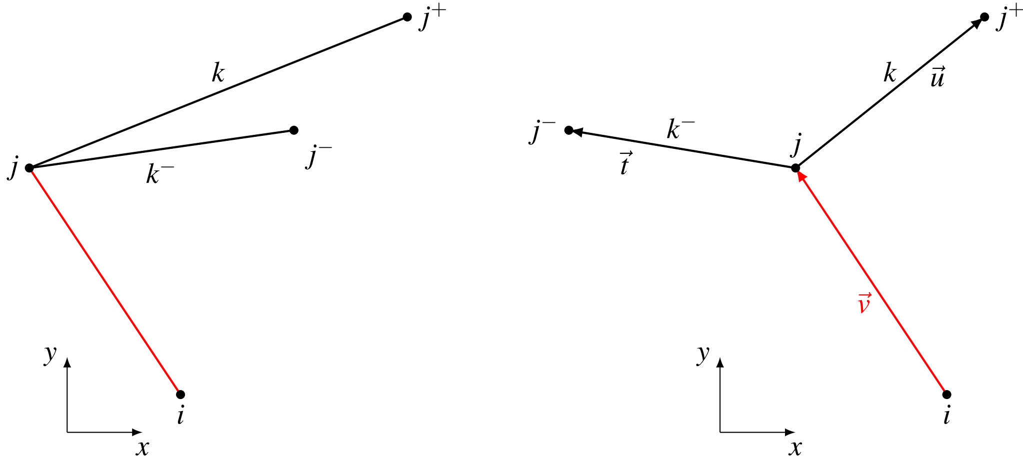
  Por otra parte, también es necesario calcular la distancia del punto j al nodo i, ρ_{k0}.

  Puesto que no se dará el caso de que no haya que añadir ninguna arista (si no, no existiría un vértice entre ellas), las aristas a agregar se introducirán en un conjunto denominado *newedges*, que contendrá o bien una arista, o bien dos. En este último caso será necesario, en primer lugar, ordenar las dos aristas a añadir. Esto se realizará a través de los productos escalares $\vec{u}_0 \cdot \vec{v}$ y $\vec{t}_0 \cdot \vec{v}$. La arista cuyo producto escalar de su vector característico con el vector director de la recta de escaneo sea menor se posicionará antes que la otra.

  En caso de que T esté vacío, se tendrá $T = newedges$ y $\rho = \rho_{k0}$. En caso contrario, se introducirá el conjunto *newedges* en T empleando un algoritmo de *Búsqueda Binaria e Inserción*, cuyo funcionamiento e implementación se muestran en el Apéndice C. De este algoritmo se obtendrán tanto los conjuntos T y ρ actualizados, así como las posiciones que ocupan las nuevas aristas en T, $Pos_{newedges}$, lo cual se indicará en *Tinv*, además de realizar en esta misma estructura la corrección de la posición de las aristas que han quedado situadas tras las insertadas en T. En 3.14 se muestra un ejemplo de estas operaciones:

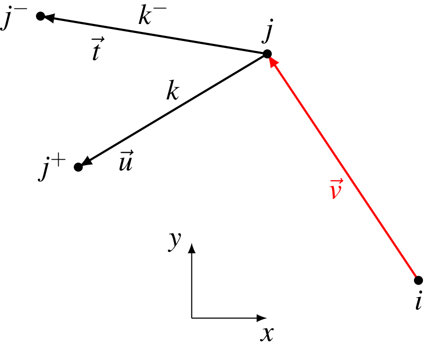


(3.14)



(a) Caso en que es necesario eliminar dos aristas.

(b) Caso en que es necesario eliminar una arista e introducir otra.



(c) Caso en que es necesario introducir dos aristas.

Figura 3.12 Casos de actualización de T.

```
219             % Caso en que hay que añadir dos aristas (si la dirección de estas no va
220             % en la del segmento ij_{j0})
221             else
222 
223                 % Vector director del segmento ij_{j0}
224                 v = (WPs_Mercator(j(j0),:) - WPs_Mercator(i,:))'; v_0 = v/norm(v);
225 
226                 % Vectores u y t
227                 u = V_Mercator(k + 1,:) - V_Mercator(k,:); u_0 = u/norm(u);
228                 t = V_Mercator(kminus,:) - V_Mercator(k,:); t_0 = t/norm(t);
229 
230                 % Distancia al nodo i
231                 rho_k0 = norm(v);
232 
233                 % Definición de las aristas a introducir en T
234                 if (abs(u_0(1) - v_0(1)) < 1e-12 && abs(u_0(2) - v_0(2)) < 1e-12) ...
235                         || (abs(u_0(1) + v_0(1)) < 1e-12 && abs(u_0(2) + v_0(2)) ...
236                         < 1e-12)
237                     newedges = kminus; % Solo se añadirá una arista
238 
239                 elseif (abs(t_0(1) - v_0(1)) < 1e-12 && abs(t_0(2) - v_0(2)) ...
240                         < 1e-14) || (abs(t_0(1) + v_0(1)) < 1e-14 && abs(t_0(2) + ...
241                         v_0(2)) < 1e-12)
242                     newedges = k; % Solo se añadirá una arista
243 
244                 else
245 
246                     %Se añadirán dos aristas
247                     rho_k0 = [rho_k0; rho_k0];
248 
249                     % Las dos aristas se ordenan a través del producto escalar
250                     udotv = dot(u_0,v);
251                     tdotv = dot(t_0,v);
252 
253                     if udotv > tdotv
254                         newedges = [kminus; k];
255                     else
256                         newedges = [k; kminus];
257                     end
258 
259                 end
260 
261                 % Introducción en T y actualización de rho y Tinv
262                 if isempty(T)
263                     T = newedges;
264                     rho = rho_k0;
265                     Tinv(T(1:length(newedges))) = 1:length(newedges);
266                 else
267                     [pos_newedges, rho, T] = BinarySearchAndInsertion(rho, rho_k0, ...
268                         T, newedges, V_Mercator, WPs_Mercator(i,:), ...
269                         WPs_Mercator(j(j0),:));
270                     Tinv(T(pos_newedges)) = pos_newedges;
271                     Tinv(T(pos_newedges(length(pos_newedges)) + 1:length(T))) = ...
272                         Tinv(T(pos_newedges(length(pos_newedges)) + 1:length(T))) ...
273                         + length(pos_newedges);
274                 end
275             end
276         end
277     end
278 end
279 end
```

Código 3.15 Caso en que es necesario añadir dos aristas en la actualización de T en la función VisibleVerticesVA(i,V).

Esto último finaliza el desarrollo del algoritmo VisibleVerticesVA(i,V) para la generación del "autografo" de visibilidad.

En las Figuras 3.13 y 3.14 se muestran los diagramas de flujo asociados a la comprobación de visibilidad de los vértices y actualización de T en la función VisibleVerticesVA(i,V).

El código desarrollado para la función VisibleVerticesOD(i,V) resulta muy similar al de la función VisibleVerticesVA(i,V), con cuatro diferencias significativas:

- Cuando se realiza la ordenación de los vértices en sentido horario, en este caso no es posible realizar un aprovechamiento del carácter simétrico del problema. Debido a lo anterior, no se tendrá el caso de que solo haya un vértice a analizar (o no haya ninguno), por lo que será necesario recorrer el algoritmo completo siempre.
- Para la comprobación de la visibilidad de los vértices, en detrimento de VisibleVA(i,j_{j0},e_1) se empleará una función programada específicamente para los casos en que el nodo i es el punto de origen O o de destino D denominada VisibleOD(i,j_{j0},e_1).
- A la hora del trazado del segmento $\overline{\sigma}$ para la inicialización de T, se ha de comprobar que uno de los puntos de intersección no sea el nodo de origen O ni de destino D, pues, si es así, no habrá que añadir ninguna arista obstáculo asociada a este nodo.
- Del mismo modo, cuando se deba actualizar T, es necesario comprobar que el punto j_{j0} no sea el nodo de origen O ni de destino D puesto que no existen aristas obstáculo incidentes a estos nodos. En caso de que j_{j0} sí sea un nodo aislado, no se actualizará T.

Puesto que la función VisibleVerticesOD(i,V) resulta muy semejante a la ya desarrollada, el código implementado en MATLAB para esta rutina se adjunta en el Apéndice D.3.3.

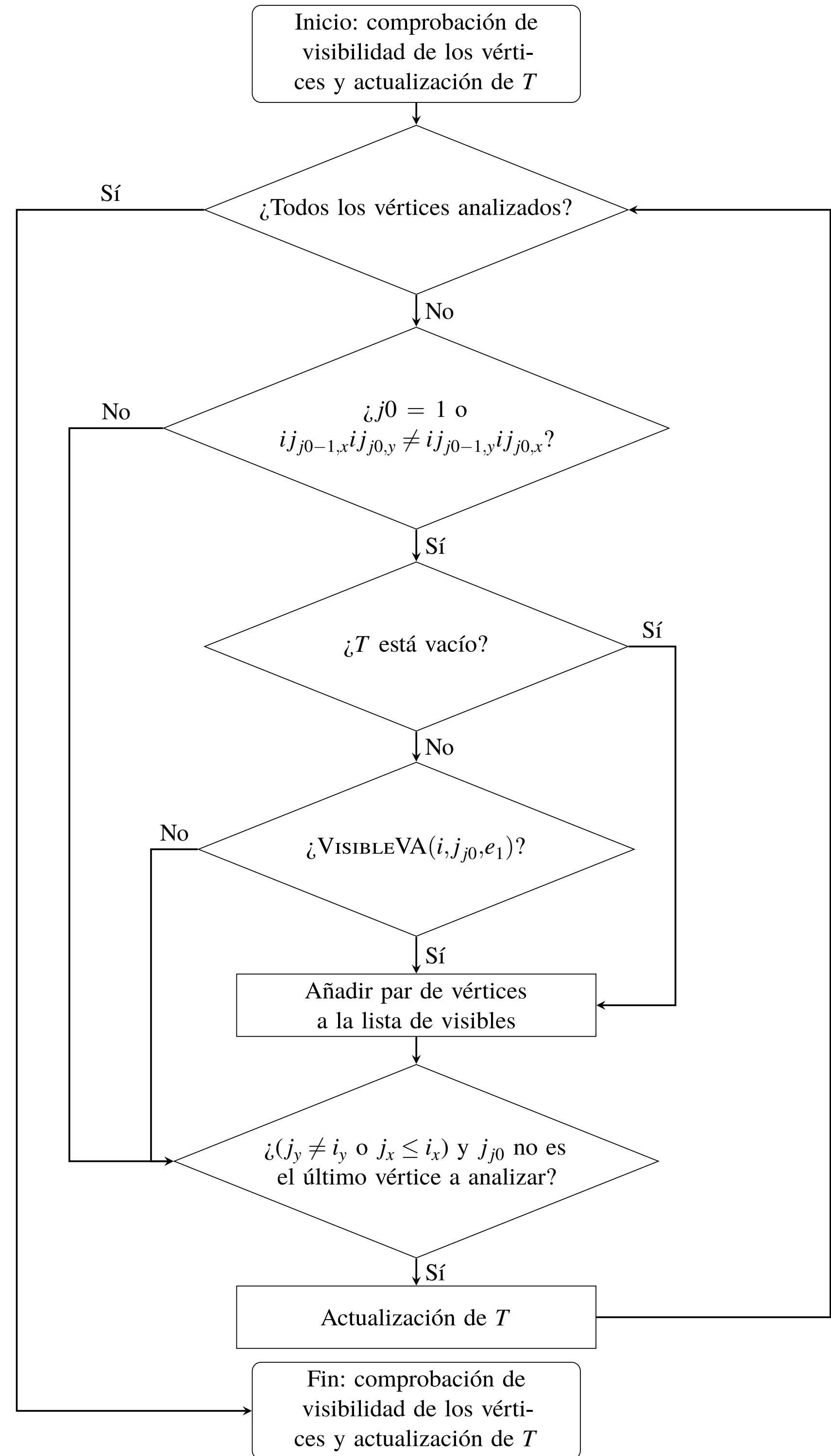


Figura 3.13 Diagrama de flujo de la comprobación de visibilidad de los vértices y actualización de T en la función VisibleVerticesVA(i, V).

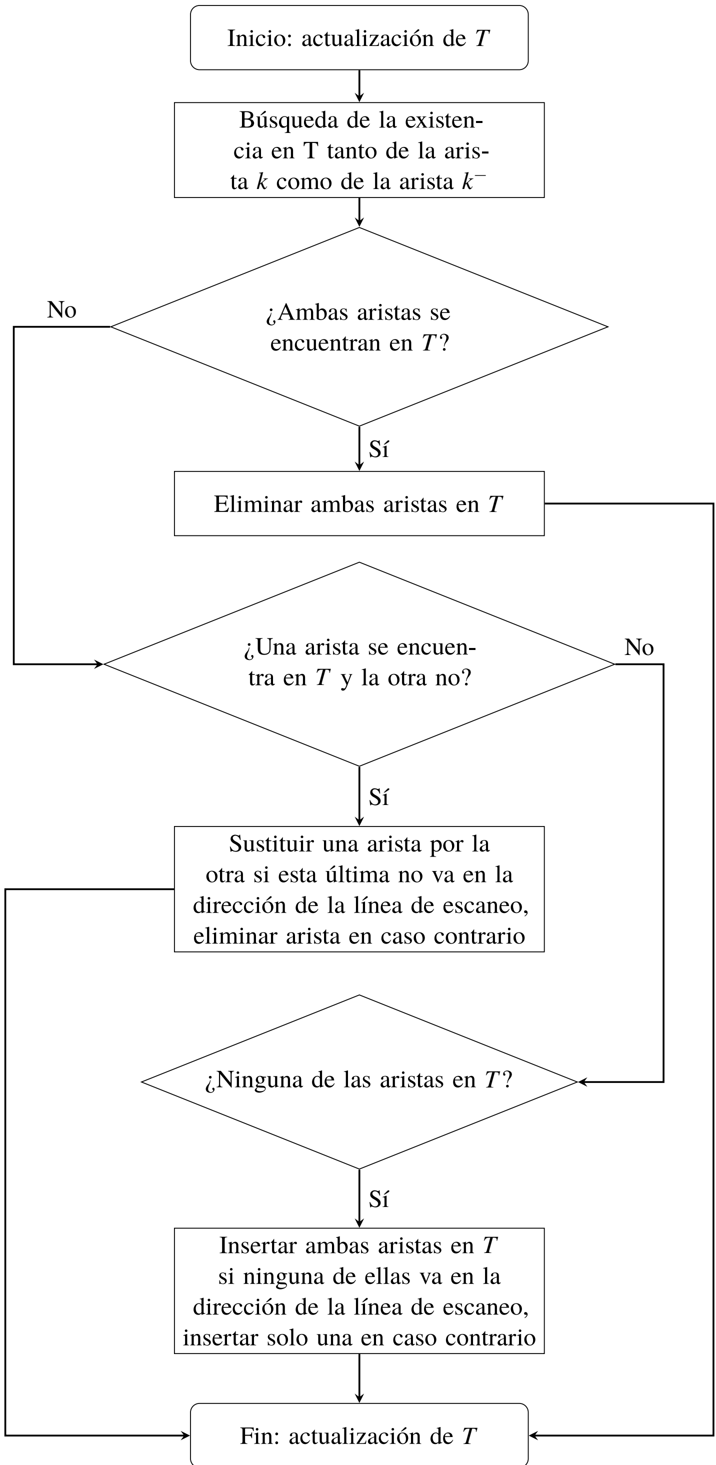


Figura 3.14 Diagrama de flujo de la actualización de T en la función VisibleVerticesVA(i,V).

3.2.3 Funciones VISIBLEVA(i, j_{j0}, e_1) y VISIBLEOD(i, j_{j0}, e_1)

Para cerrar el algoritmo de *Lee*, se va a desarrollar la función VISIBLEVA(i, j_{j0}, e_1), empleada para comprobar la visibilidad de puntos pertenecientes a polígonos obstáculo. La rutina para testear la visibilidad de los nodos del grafo desde los puntos de origen O y destino D se programará a partir de la anterior.

Debido a las entradas que recibe esta función, j denotará, en este caso, un único vértice y no un conjunto como en las funciones VISIBLEVERTICESVA(i,V) y VISIBLEVERTICESOD(i,V).

Esta rutina, en primer lugar, creará una variable bandera denominada *vis*, que será igual a 1 (verdadero) si los vértices son mutuamente visibles y 0 (falso) si no lo son.

Seguidamente, se deberá comprobar que el segmento $\overline{ij}$ no pasa por el interior de un polígono del que i es un vértice. Para ello, se volverá a aplicar el criterio basado en las aristas adyacentes al vértice i que se desarrolló en la sección 2.3.1. Los vectores $\vec{r}$ y $\vec{s}$ se computarán del mismo modo que se realizó para la función VISIBLEVERTICESVA(i,V).

```
1  function vis = VisibleVA(j, i, WPs_Mercator, e_1, E, Einv)
2
3  vis = 0;
4
5  % Comprobación de que el segmento ij no pase por el interior de un polígono del que
6  % i es vértice
7
8  % Cálculo del vector v
9  v = (WPs_Mercator(j,:) - WPs_Mercator(i,:))';
10
11 % Formación de la base no ortogonal de vectores
12 r = (WPs_Mercator(E(Einv(i, 1), 2),:) - WPs_Mercator(i,:))';
13 s = (WPs_Mercator(E(Einv(i, 2), 1),:) - WPs_Mercator(i,:))';
14
15 % Producto vectorial de r y s
16 q_z = r(1)*s(2) - r(2)*s(1);
17
18 % Resolución del sistema de ecuaciones
19 a = (v(1)*s(2) - v(2)*s(1))/q_z;
20 b = (r(1)*v(2) - r(2)*v(1))/q_z;
21
22 % Aplicación del criterio con una cierta tolerancia
23 if (sign(q_z) == -1 && ~(a>1e-12 && b>1e-12)) || (sign(q_z) == 1 && (a>=-1e-12 && ...
24     b>=-1e-12))
25     %
```

Código 3.16 Comprobación de que el segmento $\overline{ij}$ no pasa por el interior de un polígono del que i es un vértice en la función VISIBLEVA(i, j_{j0}, e_1).

Si lo anterior se cumple, se comprobará si existe la intersección del segmento $\overline{ij}$ con la primera arista del conjunto T, e_1. Para ello, se empleará el siguiente procedimiento: considérese la situación genérica mostrada en la Figura 3.15:

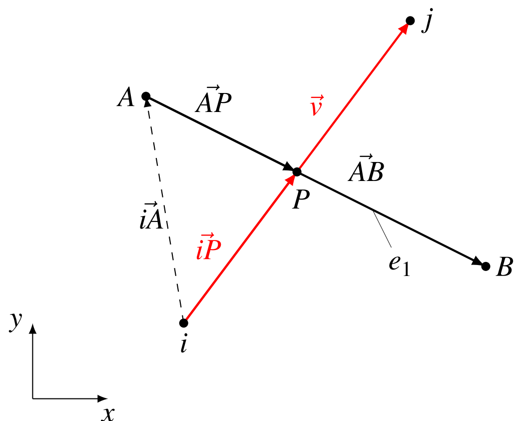


Figura 3.15 Intersección de la recta que pasa por i y j con e_1.

Sea P el punto de intersección de la recta que pasa por los vértices i y j con e_1 y sean A y B los nodos extremos de e_1. Sea el $\vec{iP}$ el vector dirigido del nodo i a P, $\vec{iA}$ el dirigido i al punto A y $\vec{AP}$ del dirigido de A al nodo P. El vector $\vec{iP}$ se puede expresar del siguiente modo:

$$\vec{iP} = \vec{iA} + \vec{AP} \tag{3.15}$$

De 3.15 se conoce $\vec{iA}$ (pues se conocen las coordenadas de los nodos i y A), pero tanto $\vec{iP}$ como $\vec{AP}$ son desconocidos. No obstante, son dato tanto $\vec{v}$ (recuérdese que es el vector dirigido de i a j) como $\vec{AB}$, vector dirigido de A a B, por lo que es posible expresar:

$$\vec{iP} = \beta\vec{v}; \qquad \vec{AP} = \gamma\vec{AB} \tag{3.16}$$

Donde β y γ son dos escalares que, a priori, no son conocidos. Sin embargo, introduciendo 3.16 en 3.15 y desarrollando:

$$\beta\vec{v} = \vec{iA} + \gamma\vec{AB} \rightarrow \beta\vec{v} - \gamma\vec{AB} = \vec{iA} \rightarrow \begin{bmatrix} v_x & -AB_x \\ v_y & -AB_y \end{bmatrix} \begin{Bmatrix} \beta \\ \gamma \end{Bmatrix} = \begin{Bmatrix} iA_x \\ iA_y \end{Bmatrix} \tag{3.17}$$

Se obtiene un sistema del que se hallará β empleando la regla de *Cramer*:

$$\beta = \frac{\begin{vmatrix} iA_x & -AB_x \\ iA_y & -AB_y \end{vmatrix}}{\begin{vmatrix} v_x & -AB_x \\ v_y & -AB_y \end{vmatrix}} = \frac{-iA_xAB_y + iA_yAB_x}{-v_xAB_y + v_yAB_x} \tag{3.18}$$

Con el objetivo de calcular los vectores $\vec{iA}$, e $\vec{iB}$, se buscarán los índices de los puntos A y B entrando con el índice de la arista obstáculo e_1 en la matriz E, para luego buscar las coordenadas de los nodos en la matriz $WPs_{Mercator}$.

A fin de comprobar entonces tanto si hay intersección como si la hay pero es justamente uno de los vértices A o B, la norma del vector $\vec{v}$, $\|\vec{v}\|$, ha de ser menor o igual que la del vector $\vec{iP}$, $\left\|\vec{iP}\right\|$, lo cual lleva a lo siguiente:

$$\|\vec{v}\| \leq \left\|\vec{iP}\right\| = \|\beta\vec{v}\| = \beta\|\vec{v}\| \rightarrow 1 \leq \beta \tag{3.19}$$

Si se cumple la condición anterior, i y j serán mutuamente visibles.

```
26     % Vectores iA y AB
27     iA = WPs_Mercator(E(e_1,1),:)' - WPs_Mercator(i,:)';
28     AB = WPs_Mercator(E(e_1,2),:)' - WPs_Mercator(E(e_1,1),:)';
29
30     % Cálculo de beta
31     beta = (-iA(1)*AB(2) + iA(2)*AB(1))/(-v(1)*AB(2) + v(2)*AB(1));
32
33     % Aplicación del criterio con una cierta tolerancia
34     if 1 <= beta + 1e-10
35         vis = 1;
36     end
37 end
38 end
```

Código 3.17 Comprobación de la intersección de $\overline{ij}$ con e_1 en la función VISIBLEVA(i,j_{j0},e_1).

En la Figura 3.16, se muestra un diagrama de flujo de la función VISIBLEVA(i,j_{j0},e_1).

Por otra parte, el código desarrollado para la función VISIBLEOD(i,j_{j0},e_1) resulta muy similar al desarrollado para la función VISIBLEVA(i,j_{j0},e_1) con una diferencia fundamental: no es necesario comprobar si el segmento $\overline{ij}$ pasa por el interior de un polígono del que i es vértice, pues los puntos de origen O y destino D no están contenidos en ningún polígono.

Puesto que la función VISIBLEOD(i,j_{j0},e_1) es muy semejante a la que se acaba de desarrollar, el código MATLAB implementado se adjunta en el Apéndice D.3.5.

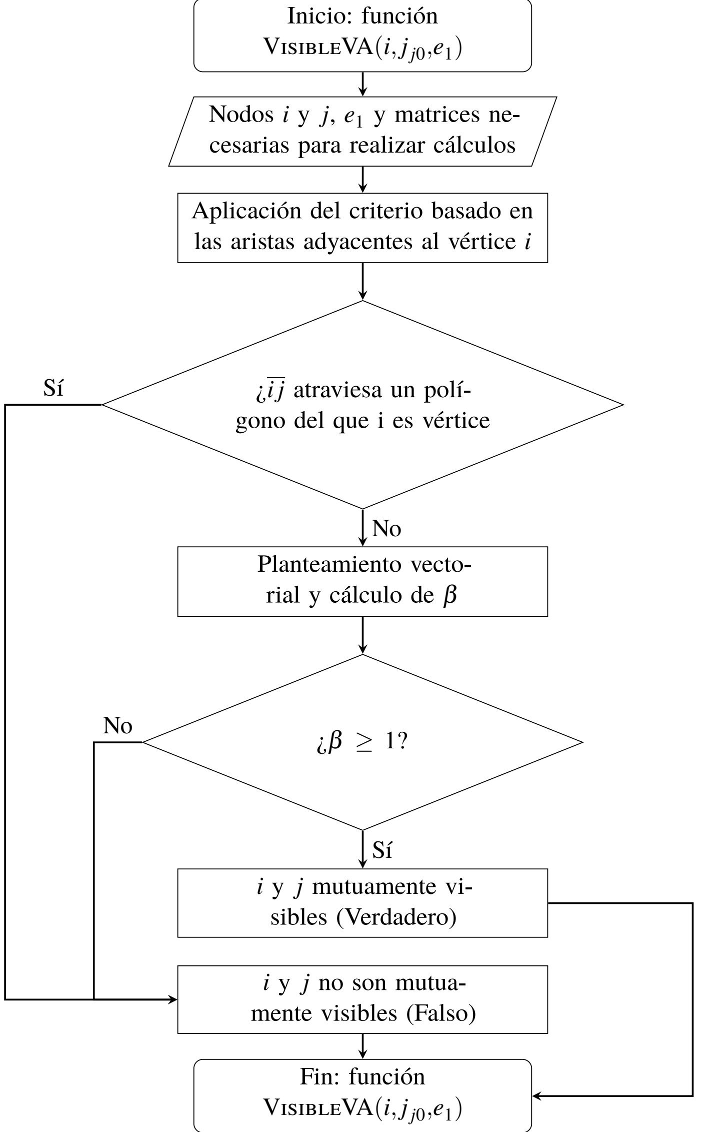


Figura 3.16 Diagrama de flujo de la función VisibleVA(i, j_{j0}, e_1).

4 Resultados

Tras el desarrollo de códigos basados tanto en el método *Ingenuo* como en el método de *Lee*, es necesario comprobar su correcto funcionamiento y obtener una serie de resultados con el fin de analizar la mejoría temporal del algoritmo de *Lee* con respecto al *Ingenuo*.

En este capítulo se mostrará, en primer lugar, la región de meteorología adversa que se considerará en este trabajo. Tras ello, se seleccionarán puntos de origen y destino para la generación de problemas y se reducirá el número de puntos de la región tormentosa con el propósito de analizar la dependencia del tiempo de ejecución de los algoritmos con el mencionado número de puntos.

4.1 Región de meteorología adversa en consideración y generación de problemas

La región tormentosa que se considerará en este trabajo se muestra en la Figura 4.1:

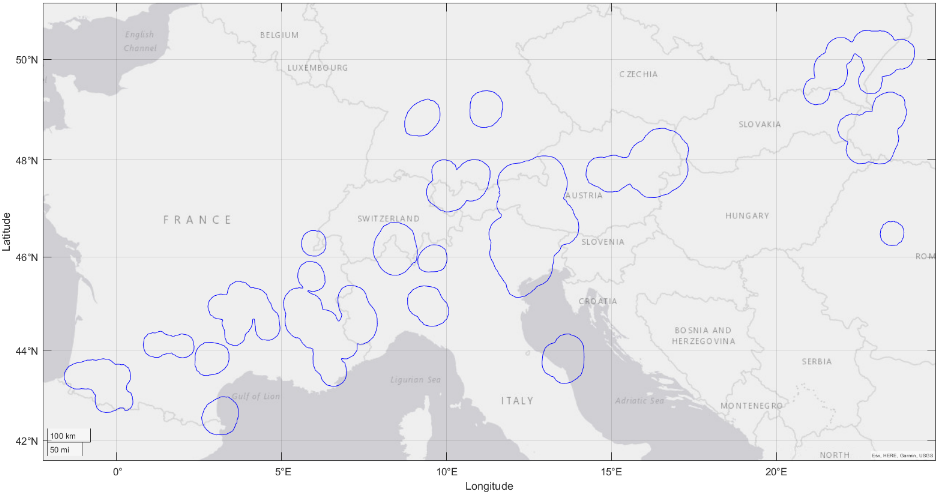


Figura 4.1 Región de meteorología adversa en consideración.

Si se traza la envolvente convexa de la región, es observable que las tormentas constituyen una franja de polígonos en Europa.

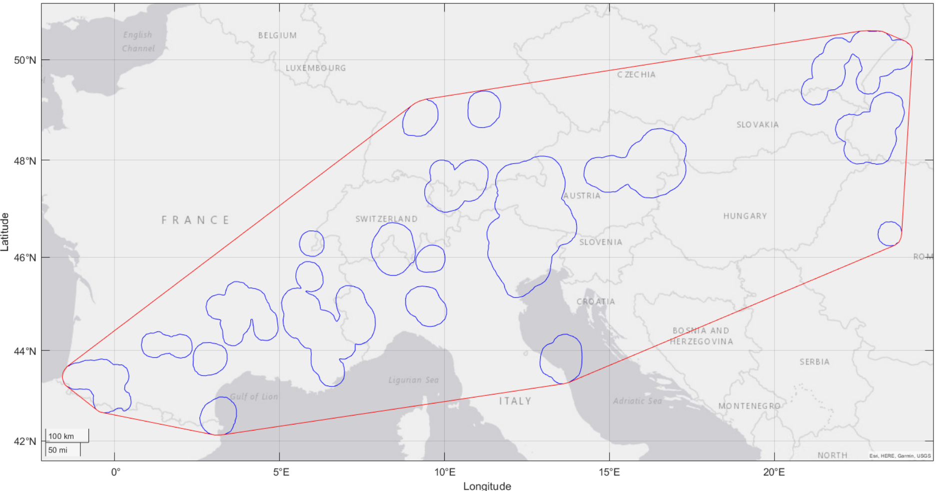


Figura 4.2 Envolvente convexa de la región de meteorología adversa en consideración.

Dicha región está compuesta por un total de $n = 1196$ puntos, a los que será necesario añadir los puntos de origen O, y destino D. Para la selección de estos puntos, se considerarán vuelos entre grandes ciudades de Europa que se encuentren tanto en la proximidad de la envolvente convexa de la región como en el interior cuyas coordenadas geográficas se han tomado de [17]. Por otra parte, también se ejecutarán trayectos entre puntos artificiales para los que la determinación de la trayectoria más corta presente una dificultad superior.

Cada uno de los problemas considerados se resolverá tanto a través de los códigos desarrollados según el método *Ingenuo* como mediante el código implementado siguiendo el método de *Lee*, comprobando siempre que los resultados obtenidos para cada problema, empleando los distintos códigos, son idénticos.

Por otro lado, será necesario analizar los tiempos de ejecución empleados por cada programa. Además, resulta interesante obtener la sensibilidad de los tiempos de ejecución con respecto al número de nodos del grafo en consideración para analizar empíricamente la complejidad de los métodos y comprobar si se corresponde con los modelos teóricos. Con este fin, se reducirá el número de puntos de la región de meteorología adversa mostrada en la Figura 4.1 empleando el algoritmo de *Ramer-Douglas-Peucker* ([18]). Este algoritmo trata de reducir el número de vértices existentes en una polilínea. Se basa en el cálculo de puntos críticos a partir de una tolerancia lineal los cuales constituirán la línea simplificada. Estos puntos, serán los que vayan alcanzando una distancia perpendicular superior a la tolerancia impuesta, respecto a la línea base a tener en cuenta que, en principio, es la limitada por los puntos inicial y final. Después, se calcularán las distancias perpendiculares de todos los puntos intermedios:

- Si ninguna de estas distancias excede la tolerancia, la simplificación de vértices habrá terminado y quedarán en la simplificación los puntos inicial y final de la línea.
- En el caso de que sí se exceda la tolerancia, el punto con mayor distancia se tomará como punto crítico que dividirá a la línea base en dos segmentos, y así, se seguirá repitiendo el proceso con cada segmento hasta que no sea necesario realizar más divisiones.

A continuación, se muestra un diagrama de flujo del algoritmo de *Ramer-Douglas-Peucker*:

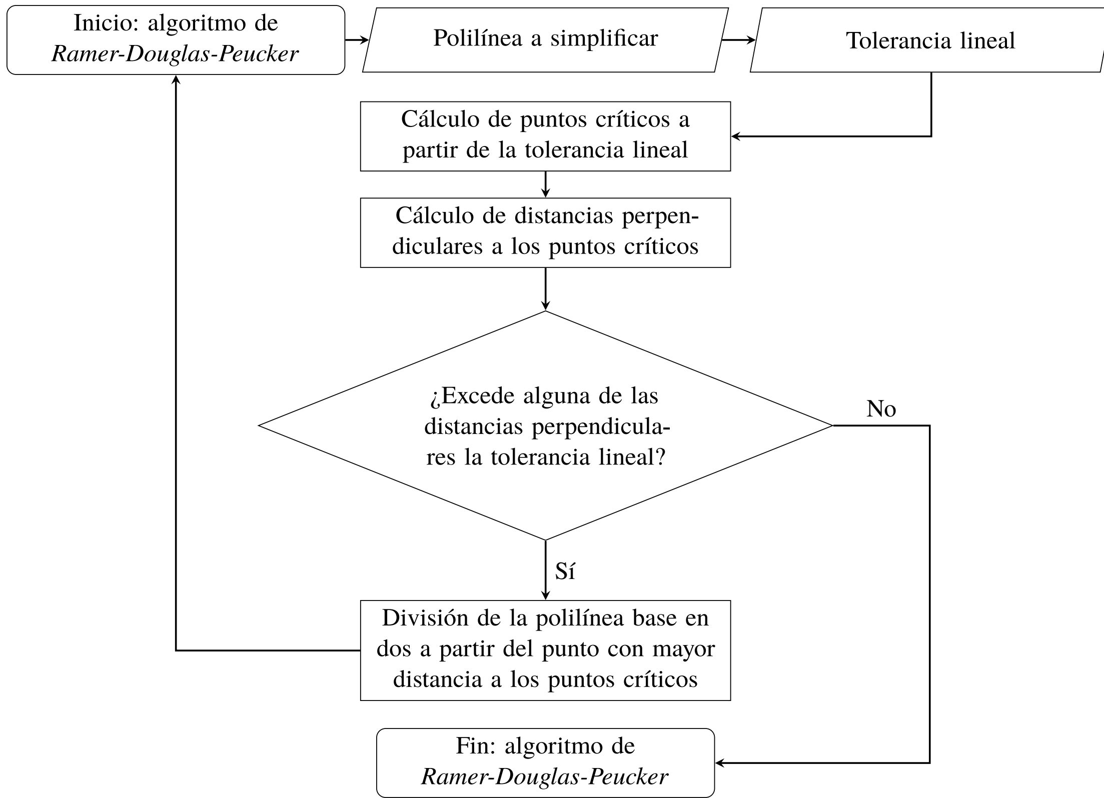


Figura 4.3 Diagrama de flujo del algoritmo de *Ramer-Douglas-Peucker*.

El algoritmo de *Ramer-Douglas-Peucker* se aplicará a la región tormentosa a través del comando *reducem* en MATLAB. Este, implementa el método recibiendo directamente los vértices en coordenadas sobre la Tierra (ϕ,λ). Las tolerancias se escogerán ajustándolas al número de puntos que se pretenda conseguir.

Del mismo modo, tras la aplicación del comando, es necesario comprobar que los polígonos resultantes no se solapen. En el caso de la región en consideración y para las tolerancias escogidas, esto no sucede.

En conclusión, y pese a que el autor ha resuelto muchos más, se mostrarán seis problemas, cuya elección de puntos de origen y destino es variada, cada uno de ellos resuelto para cinco regiones con distinto número de puntos. Se mostrará el grafo de visibilidad únicamente para el primer problema resuelto debido a su similitud con el resto, así como la trayectoria más corta para todos los problemas en el caso del número de puntos original de la región.

Tras lo anterior, se mostrarán los tiempos medios de ejecución resultantes de los seis problemas (de nuevo, no se muestran los tiempos para cada problema por separado debido a su similitud) y se realizará un análisis de los mismos.

4.2 Problemas resueltos

Se adjuntan a continuación seis problemas resueltos para la puesta en marcha de los algoritmos y la comprobación de su correcto funcionamiento.

4.2.1 Problema 1. Vuelo Bruselas-Sarajevo

En este primer problema, el transcurso de la aeronave a lo largo de la región tormentosa se realizará de forma transversal. El grafo de visibilidad resultante se muestra a modo de ejemplo en la Figura 4.4:

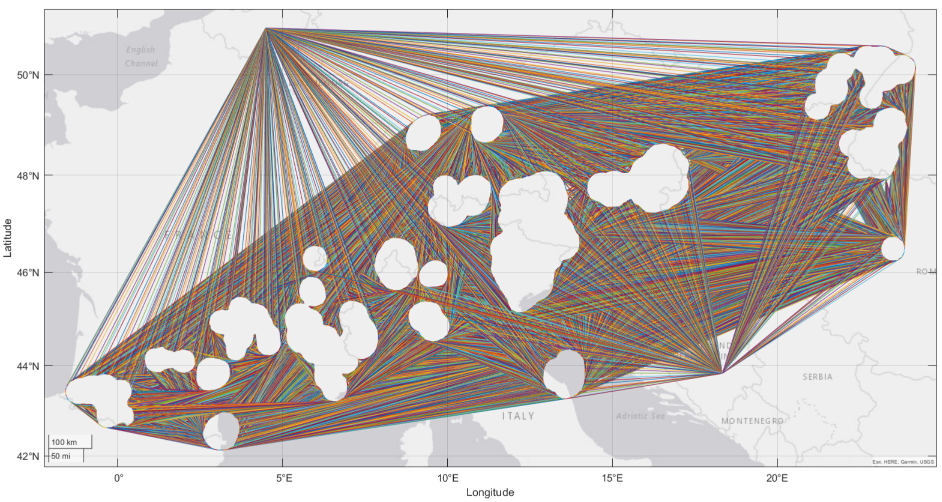


Figura 4.4 Grafo de visibilidad del vuelo Bruselas-Sarajevo.

Asimismo, la trayectoria más corta obtenida se muestra en la Figura 4.5. Se tiene un camino desarrollado a lo largo de ocho nodos incluyendo origen y destino.

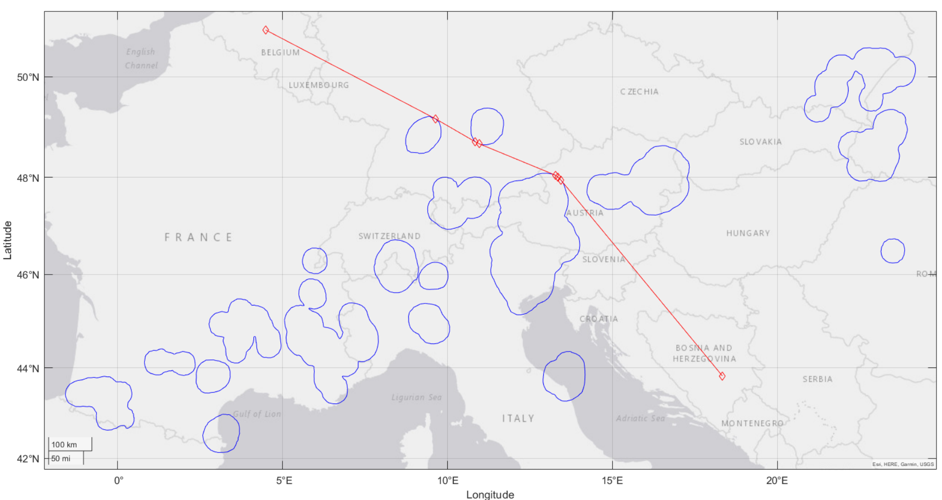


Figura 4.5 Trayectoria más corta del vuelo Bruselas-Sarajevo.

4.2.2 Problema 2. Vuelo Madrid-Kiev

Se propone en este caso un problema en el que el transcurso de la aeronave a lo largo de la región tormentosa se realiza de forma longitudinal. La trayectoria más corta obtenida se muestra en la Figura 4.6. Se tiene un camino desarrollado a lo largo de diez nodos incluyendo origen y destino.

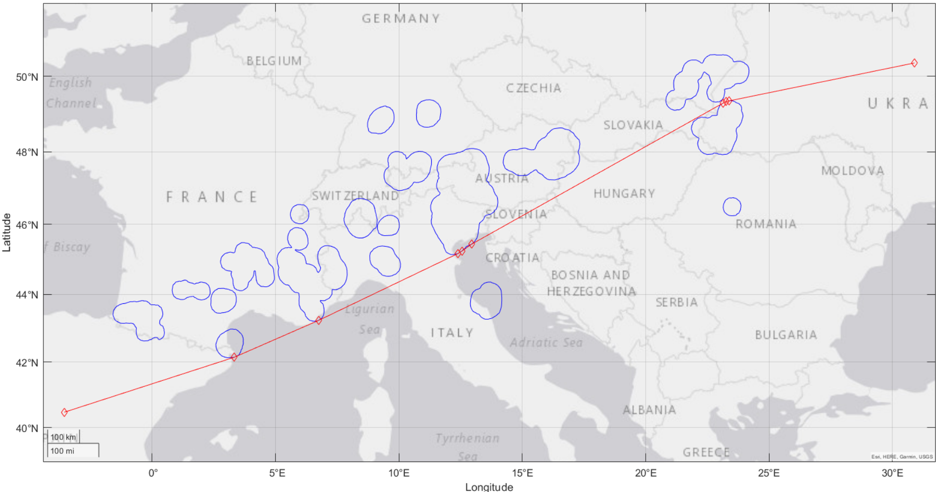


Figura 4.6 Trayectoria más corta del vuelo Madrid-Kiev.

4.2.3 Problema 3. Vuelo Turín-Varsovia

En esta situación, se tiene un problema cuyo punto de origen es una ciudad interior a la región tormentosa y cuyo punto de destino es una ciudad exterior. La trayectoria más corta obtenida se muestra en la Figura 4.7. Se tiene un camino desarrollado a lo largo de 14 nodos incluyendo origen y destino.

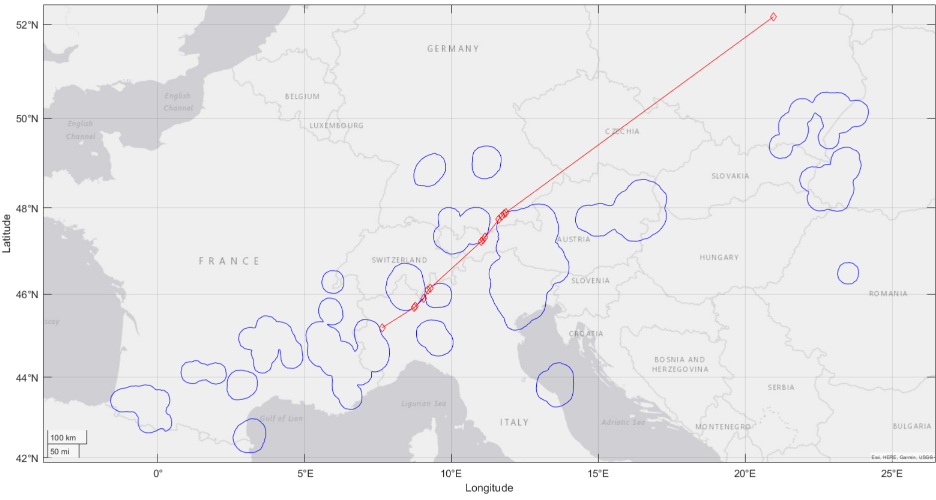


Figura 4.7 Trayectoria más corta del vuelo Turín-Varsovia.

4.2.4 Problema 4. Vuelo Ljubljana-Berna

Para este problema se ha escogido un vuelo entre dos ciudades interiores a la región de meteorología adversa. La trayectoria más corta obtenida se muestra en la Figura 4.8. Se tiene un camino desarrollado a lo largo de 15 nodos incluyendo origen y destino.

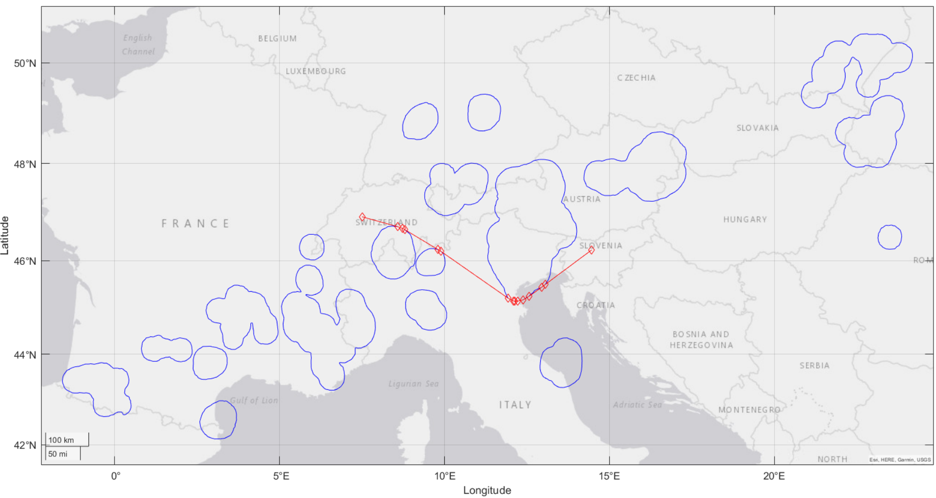


Figura 4.8 Trayectoria más corta del vuelo Ljubljana-Berna.

4.2.5 Problema 5. Vuelo entre $O = (43.1°N, 0.730°W)$ y $D = (44.7°N, 6.75°E)$

Con el objetivo de obtener trayectorias óptimas de mayor complejidad, se va a resolver un problema cuyos puntos de origen y destino no se correspondan con coordenadas de aeropuertos reales pero que parezcan inducir una mayor dificultad. La trayectoria más corta obtenida se muestra en la Figura 4.9. Se tiene un camino desarrollado a lo largo de 24 nodos incluyendo origen y destino.

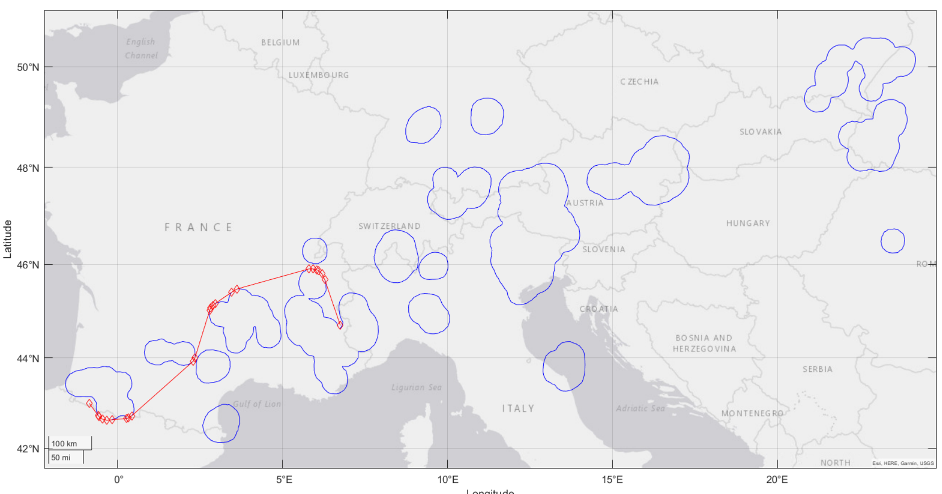


Figura 4.9 Trayectoria más corta del vuelo entre $O = (43.1°N, 0.730°W)$ y $D = (44.7°N, 6.75°E)$.

4.2.6 Problema 6. Vuelo entre $O = (44.5°N, 4.20°E)$ y $D = (50.0°N, 22.5°E)$

Al igual que en el problema anterior, se hallará una trayectoria cuyos puntos de origen y destino sean artificiales. La trayectoria más corta obtenida se muestra en la Figura 4.10. Se tiene un camino desarrollado a lo largo de 41 nodos incluyendo origen y destino.

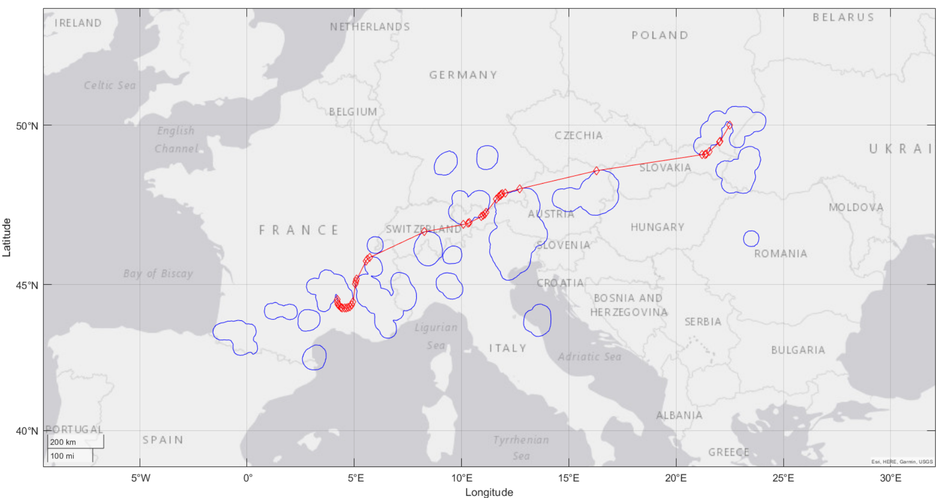


Figura 4.10 Trayectoria más corta del vuelo entre $O = (44.5°N, 4.20°E)$ y $D = (50.0°N, 22.5°E)$.

4.3 Análisis de resultados

En la Tabla 4.1, se muestran los tiempos medios de ejecución obtenidos a través de los seis problemas resueltos con distinto número de puntos de las regiones según los diferentes algoritmos programados en este trabajo. Para dar estabilidad a la solución reduciendo la variabilidad estadística, se han resuelto varias veces los mismos problemas y se han promediado los tiempos de ejecución.

Tabla 4.1 Tiempos medios de ejecución de los algoritmos.

N	$t_{Ing_{v1}}(s)$	$t_{Ing_{v2}}(s)$	$t_{Lee}(s)$
1198	599.1	454.5	22.27
935	419.0	269.5	13.34
811	275.4	191.1	10.75
554	98.58	87.08	5.932
381	60.18	38.57	3.446

Tal como se esperaba y para los tres algoritmos, cuanto mayor es el número de nodos considerado, mayor es el tiempo de ejecución. Por otra parte, se tiene una ligera mejora temporal de la segunda versión del algoritmo *Ingenuo* con respecto al primero. Más significativa es la mejora producida gracias al método de *Lee*, dado que el tiempo de ejecución de este algoritmo es un orden de magnitud menor que los de los algoritmos *Ingenuo*.

En la Tabla 4.2, se muestra el porcentaje de tiempo empleado en la resolución a través del método de *Lee* con respecto al *Ingenuo*. Es observable que, conforme el número de puntos de la región disminuye, la mejora temporal del método de *Lee* con respecto al *Ingenuo* disminuye paulatinamente.

Tabla 4.2 Porcentaje de tiempo empleado en la resolución a través del método de *Lee* respecto a los algoritmos *Ingenuo*.

N	$t_{Lee}/t_{Ing_{v1}}$	$t_{Lee}/t_{Ing_{v2}}$
1198	3.72 %	4.90 %
935	3.18 %	4.95 %
811	3.90 %	5.63 %
554	6.02 %	6.81 %
381	5.73 %	8.93 %

Para culminar el análisis de resultados, se muestra en la Figura 4.11 un gráfico en doble escala logarítmica del tiempo de ejecución t frente al número de puntos N tras realizar un ajuste lineal de la nube de puntos obtenida en la resolución de los seis problemas planteados, cada uno para el distinto número de nodos de las regiones consideradas.

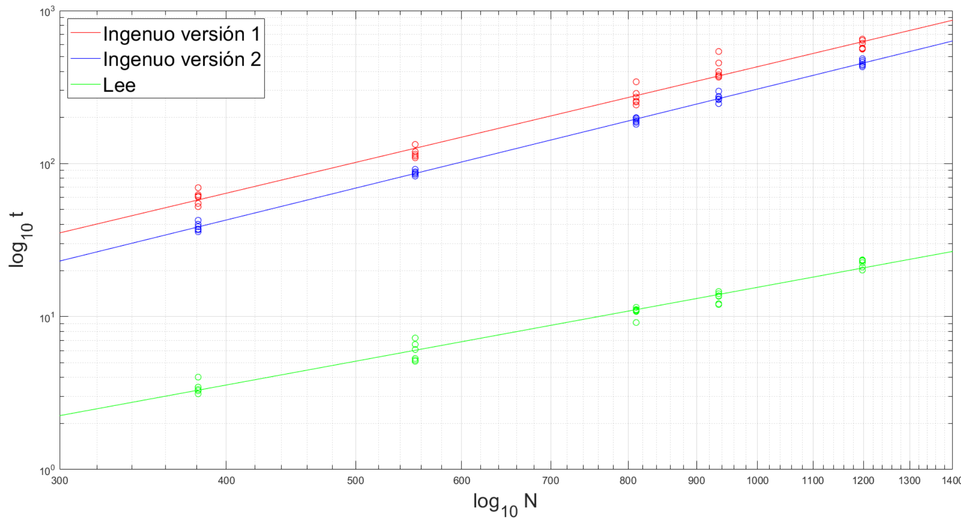


Figura 4.11 Representación del ajuste de los puntos obtenidos en la resolución de problemas.

Teóricamente, la pendiente de las rectas características del método de *Ingenuo* en esta gráfica debería corresponderse con lo siguiente:

$$O(\log_{10} N^3) \sim O(3\log_{10} N) \rightarrow m_{Ing} \sim 3 \tag{4.1}$$

Del mismo modo, en el caso del algoritmo de *Lee* es posible realizar un desarrollo análogo al anterior:

$$O[\log_{10}(N^2\log_2 N)] \sim O[2\log_{10}N+\log_{10}(\log_2 N)] \sim O[\log_{10}N\{2+\frac{\log_{10}(\log_2 N)}{\log_{10}N}\}] \sim \\ \sim O[\log_{10}N\{2+\varepsilon\}] \tag{4.2}$$

Para conocer la pendiente en el tramo doblemente logarítmico, se ha de derivar $\log_{10}(\log_2 N)$ respecto de $\log_{10}N$:

$$\varepsilon = \frac{d}{d\log_{10}N}\log_{10}(\log_2 N) = \frac{1}{\log_{10}N\ln 10} = \frac{1}{\ln N} \tag{4.3}$$

Considerando el número de nodos de las regiones que se han estudiado, se tendría por tanto:

$$2.14 \sim m_{Lee} \sim 2.17 \tag{4.4}$$

Las pendientes obtenidas tras el ajuste lineal realizado son de $m_{Ing_{v1}} = 2.08$, $m_{Ing_{v2}} = 2.15$ y $m_{Lee} = 1.60$, valores del orden de magnitud de los modelos teóricos y que además confirman que, conforme el número de puntos de la región disminuye, la mejora temporal del método de *Lee* con respecto al *Ingenuo* disminuye. No obstance, estas pendientes no se corresponden exactamente con las supuestas por los modelos teóricos. Cabe por tanto preguntarse por qué ocurre esto.

En el caso de los algoritmos *Ingenuo*, se realiza un uso continuado del comando *polyxpoly*, cuyo funcionamiento interno es desconocido por el autor, por lo que quizás el costo computacional de la comprobación de la intersección del segmento $\overline{ij}$ con las aristas obstáculo podría ser menor que el teórico $O(n) \sim O(N)$. Además, se ha realizado un aprovechamiento del carácter simétrico del problema, lo cual ayuda a rebajar el costo temporal del mismo.

En el caso del algoritmo de *Lee*, el aprovechamiento del carácter simétrico del problema juega un papel aún superior, pues conforme mayor sea la latitud del nodo i, menos vértices se tratarán al considerar únicamente aqueyos presentes en el plano $y - i_y \geq 0$ en la función $\textsc{VisibleVertices}\text{VA}(i,V)$. Por tanto, el costo computacional real aproximado del algoritmo sería el siguiente:

$$O[\sum_{j=1}^{n-1} j\log_2 j + 2(N-1)\log_2(N-1)] \tag{4.5}$$

Donde $O(\sum_{j=1}^{n-1} j\log_2 j)$ es el costo computacional debido a la generación del "autografo" de visibilidad mediante la aplicación recurrente de la función $\textsc{VisibleVertices}\text{VA}(i,V)$, y $O[2(N-1)\log_2(N-1)]$ es el costo computacional debido a la introducción de los puntos de origen O y destino D al autografo mediante la aplicación de la función $\textsc{VisibleVertices}\text{OD}(i,V)$.

La obtención de las pendientes reales que deberían presentar las rectas en la Figura 4.11 se considera un problema que queda fuera del alcance de este trabajo.

En todo caso, el análisis realizado con las complejidades teóricas de $O(N^3)$ y $O(N^2 log_2 N)$ debe interpretarse siempre como un límite cuando N tiende a un número grande debido a que se han despreciado órdenes inferiores, por lo que esto constituye algo aproximado.

5 Conclusiones y líneas futuras

En este último capítulo, se pretende concluir el presente trabajo recapitulando todo lo realizado a lo largo del mismo y resumiendo las conclusiones extraídas. Asimismo, se señalarán una serie de líneas futuras a realizar en otros proyectos.

En primer lugar, se planteó una introducción en la que se exponía la necesidad de mejorar la eficiencia computacional de los algoritmos de evitación de tormentas basados en la generación del grafo de visibilidad.

En el Capítulo 2, se programaron los algoritmos basados en el método *Ingenuo* que constituyen la implementación actual de la que dispone el *Grupo de Ingeniería Aeroespacial* para la generación del grafo de visibilidad.

En el Capítulo 3, se programó un algoritmo basado en el método de *Lee* cuyo objetivo era obtener una mejora significativa en cuanto a costo temporal se refiere con respecto a los algoritmos programados en el Capítulo 2.

Tras la resolución de problemas y el análisis de los resultados obtenidos, se concluye que el algoritmo basado en el método de *Lee* es significativamente más eficiente que aquellos programados bajo el método *Ingenuo*, por lo que el autor propone el empleo del algoritmo de *Lee* desarrollado en este trabajo en detrimento de los algoritmos *Ingenuo* que constituyen la implementación actual del *Grupo de Ingeniería Aeroespacial*.

Por último, se van a apuntar una serie de líneas futuras que se pueden desarrollar a partir de este proyecto y no se han abordado en el mismo debido a las limitaciones en cuanto al tiempo y el alcance del trabajo:

- **Implementación de otros algoritmos de programación cuyo costo computacional teórico sea menor que aquellos que se han desarrollado en este trabajo**

  En este proyecto, para un grafo de N nodos, se han considerado algoritmos para la generación del grafo de visibilidad basados en el método de *Ingenuo* y en el método de *Lee*, cuyos costos computacionales teóricos son de $O(N^3)$ y $O(N^2 \log_2 N)$ respectivamente, siendo este último significativamente menor que el primero.

  No obstante, existen otros métodos para la generación del grafo de visibilidad que podrían presentan costos computacionales menores como, por ejemplo, el presentado por *Ghosh* y *Mount* en [21], cuya complejidad es de $O(E + N \log_2 N)$, donde E es, en este caso, el número de aristas del grafo de visibilidad. La implementación de algoritmos basados en estos métodos

podría contribuir a mejorar aún más la solución obtenida a través del método de *Lee*.

- **Implementación de algoritmos con un enfoque distinto al de la generación del grafo de visibilidad empleando todos los nodos de los polígonos obstáculo**

  Tras el empleo de los algoritmos desarrollados en este trabajo, se obtiene un conjunto de conexiones entre nodos muy extenso. No obstante, un gran número de las trayectorias derivadas de estas conexiones podrían despreciarse debido a que, en situaciones reales, una aeronave nunca las seguiría.

  En consecuencia, sería interesante la implementación de algoritmos que generasen únicamente conexiones relevantes para la ejecución de los planes de vuelo de un avión y eliminen el costo de computación asociado a las trayectorias irrelevantes. El planteamiento de estos algoritmos se podría basar en el cálculo de las envolventes convexas de los polígonos que conforman la región tormentosa para, tras ello, calcular 4 tangentes para cada posible emparejamiento de polígonos, y, posteriormente, pasar a comprobar si son visibles o no (lo serán si no cortan un tercer polígono). Por otra parte, también se trazarían las tangentes desde el origen a todos los polígonos y del destino a todos los polígonos, procediéndose a eliminar aquellas que no sean visibles.

  Este planteamiento podría contribuir a disminuir significativamente el costo computacional del problema, eso sí, generando un número de conexiones entre nodos considerablemente menor.

Apéndice A
Cálculo de distancias loxodrómicas

En este apéndice, se muestra el procedimiento empleado en este trabajo para calcular las distancias loxodrómicas d_{lox} entre los vértices mutuamente visibles del grafo. Con el fin de evitar el uso de comandos de MATLAB que conllevan un alto costo computacional, se va a recurrir a un planteamiento basado en una serie de fórmulas analíticas.

Sean (ϕ_i,λ_i) y (ϕ_j,λ_j) las coordenadas de dos puntos i y j entre los que se quiere calcular la distancia loxodrómica, y sean R_T el radio de la Tierra y h la altura de vuelo. El proceso para computar la distancia loxodrómica d_{lox} es el siguiente ([5]):

1. Si $\phi_j \neq \phi_i$

a) Cálculo del rumbo loxodrómico χ_{lox}:

$$\chi_{lox} = \arctan\left\{\frac{\lambda_j - \lambda_i}{\ln\left[\frac{\tan(\pi/4-\phi_i/2)}{\tan(\pi/4-\phi_j/2)}\right]}\right\} \quad \text{Si } \left|\lambda_j - \lambda_i\right| \leq \pi \tag{A.1}$$

$$\chi_{lox} = \arctan\left\{\frac{\lambda_j + 2\pi - \lambda_i}{\ln\left[\frac{\tan(\pi/4-\phi_i/2)}{\tan(\pi/4-\phi_j/2)}\right]}\right\} \quad \text{Si } \left|\lambda_j - \lambda_i\right| > \pi \text{ y } \lambda_j < 0 \tag{A.2}$$

$$\chi_{lox} = \arctan\left\{\frac{\lambda_j - 2\pi - \lambda_i}{\ln\left[\frac{\tan(\pi/4-\phi_i/2)}{\tan(\pi/4-\phi_j/2)}\right]}\right\} \quad \text{Si } \left|\lambda_j - \lambda_i\right| > \pi \text{ y } \lambda_j > 0 \tag{A.3}$$

- Corrección del rumbo loxodrómico:

$$\chi_{lox} = \pi + \chi_{lox} \quad \text{Si } \phi_j < \phi_i \tag{A.4}$$

b) Cálculo de la distancia angular loxodrómica α_{lox}:

$$\alpha_{lox} = \frac{\phi_j - \phi_i}{\cos\chi_{lox}} \tag{A.5}$$

2. Si $\phi_i = \phi_j = \phi$

$$\alpha_{lox} = |\lambda_j - \lambda_i| \cos\phi \tag{A.6}$$

3. Cálculo de la distancia loxodrómica:

$$d_{lox} = \alpha_{lox}(R_T + h) \tag{A.7}$$

Nótese que, en este trabajo, puesto que se considera una altitud de vuelo constante para la generación de la trayectoria más corta, se ha tomado $h = 0$, pues no es influyente en el resultado.

A continuación se muestra la función implementada en MATLAB para el cálculo de las distancias loxodrómicas.

```
function d_lox = LoxodromicDistance(phi_i, lambda_i, phi_j, lambda_j, h)

R_T = 6378.14; % Radio de la Tierra en kilómetros

if phi_j == phi_i
    alpha_lox = abs(lambda_j - lambda_i)*cos(phi_i);

else
    if abs(lambda_j - lambda_i) > pi
        if lambda_j < 0
            ji_lox = atan((lambda_j + 2*pi - lambda_i)/log((tan(pi/4 - phi_i/2)/...
                tan(pi/4 - phi_j/2))));
        else
            ji_lox = atan((lambda_j - 2*pi - lambda_i)/log((tan(pi/4 - phi_i/2)/...
                tan(pi/4 - phi_j/2))));
        end
    else
        ji_lox = atan((lambda_j - lambda_i)/log((tan(pi/4 - phi_i/2)/tan(pi/4 - ...
            phi_j/2))));
    end

    if phi_j < phi_i
        ji_lox = ji_lox + pi;
    end

    alpha_lox = (phi_j - phi_i)/cos(ji_lox);

end

d_lox = (R_T + h)*alpha_lox;

end
```

Código A.1 Función para el cálculo de distancias loxodrómicas.

Apéndice B
Algoritmos de ordenación

En este apéndice, se muestran los algoritmos de ordenación empleados cuando ha sido requerido en el desarrollo del método de *Lee*. Estos, se basan en el método *HeapSort* que garantiza que, para realizar una ordenación de n elementos, el costo computacional de la misma en el peor de los casos es de $O(n \log n)$.

Puesto que el objetivo de este trabajo no es la codificación del método, *HeapSort* ([6]), no se va a explicar su funcionamiento. No obstante, sí se va a señalar que los algoritmos *HeapSort* usualmente programados ordenan un vector según el valor de sus elementos, mientras que los métodos *HeapSort* empleados en este trabajo han empleado otras condiciones para la ordenación de sus elementos. Estas se detallan en las siguientes secciones.

B.1 Primera versión del método *HeapSort*

La primera versión del algoritmo de ordenación *HeapSort* se ha empleado para realizar la ordenación de vértices en sentido antihorario a partir del sentido positivo del eje x alrededor de un nodo i. Para ello, se ha hecho uso de las coordenadas de los vértices alrededor del vértice i (citando [6]).

Considérese dos nodos genéricos w_1 y w_2 tales que w_1 se encuentra en una posición inferior que w_2 en la lista donde se han de ordenar los vértices de los distintos nodos alrededor del nodo i. El procedimiento seguido para decidir si se han de permutar las posiciones de w_1 y w_2 se muestra en el Pseudocódigo B.1.

En primer lugar, se hacen comprobaciones basadas en el incremento de la coordenada y de los puntos w_1 y w_2 con respecto a la coordenada y del nodo i. Por otra parte, se comparan los nodos w_1 y w_2 cuando tienen la misma coordenada y que el vértice i. Si lo anterior no permite la ordenación de los nodos, se comprueban las desigualdades relativas al producto vectorial de la componente no nula de lo que serían los vectores $\vec{iw}_1$ y $\vec{iw}_2$. En caso de que estos vectores tengan la misma dirección, se comprueba la distancia de w_1 y w_2 al nodo i.

Pseudocódigo B.1 LESS(w_1,w_2,i)

Entrada: Las coordenadas de los nodos entre los que se plantea la permutación, $(w_{1,x},w_{1,y})$ y $(w_{2,x},w_{2,y})$, y las coordenadas del nodo alrededor del que se realiza la ordenación de vértices, (i_x,i_y).

Salida: **Verdadero** o **Falso**.

1: **si** $w_{2,y} - i_y \geq 0$ y $w_{1,y} - i_y < 0$ **entonces**
2:     **devolver verdadero**
3: **si no si** $w_{2,y} - i_y < 0$ y $w_{1,y} - i_y \geq 0$ **entonces**
4:     **devolver falso**
5: **si no si** $w_{2,y} - i_y = 0$ y $w_{1,y} - i_y = 0$ **entonces**
6:     **si** $w_{2,x} - i_x > 0$ **entonces**
7:         **si** $w_{1,x} - i_x > 0$ **entonces**
8:             **si** $w_{2,x} < w_{1,x}$ **entonces**
9:                 **devolver verdadero**
10:             **si no**
11:                 **devolver falso**
12:         **si no**
13:             **devolver verdadero**
14:     **si no**
15:         **si** $w_{1,x} - i_x < 0$ **entonces**
16:             **si** $w_{2,x} > w_{1,x}$ **entonces**
17:                 **devolver verdadero**
18:             **si no**
19:                 **devolver falso**
20:         **si no**
21:             **devolver falso**
22: **si no**
23:     $det = (w_{2,x} - i_x)(w_{1,y} - i_y) - (w_{2,y} - i_y)(w_{1,x} - i_x)$
24:     **si** $det > 0$ **entonces**
25:         **devolver verdadero**
26:     **si no si** $det < 0$ **entonces**
27:         **devolver falso**
28:     **si no**
29:         **si** $(w_{2,x} - i_x)^2 + (w_{2,y} - i_y)^2 < (w_{1,x} - i_x)^2 + (w_{1,y} - i_y)^2$ **entonces**
30:             **devolver verdadero**
31:         **si no**
32:             **devolver falso**

La primera versión de la función *HeapSort* recibirá como argumentos de entrada una matriz A (nótese que en este caso, A no es uno de los nodos extremos de e_1 como se mostró en la sección 3.2.3) que se corresponderá con la matriz mostrada en 3.7 y, además, las coordenadas del nodo i alrededor del que se realiza la ordenación, en este caso denotado como c.

A continuación, se muestra el código de las funciones desarrolladas para la implementación de la primera versión del método *HeapSort* en este trabajo.

```
1 function A = HeapSort_v1(A, c)
2 Dim = size(A); N = Dim(1);
```

```
LastTopPos = floor(N/2);

for i = LastTopPos:-1:1
    A = Heapify_v1(A, N, i, c);
end

for i = N-1:-1:1
    aux = A(i+1,:);
    A(i+1,:) = A(1,:);
    A(1,:) = aux;
    A = Heapify_v1(A, i, 1, c);
end
end
```

Código B.1 Primera versión del algoritmo *HeapSort*.

```
function A = Heapify_v1(A, N, i, c)
BiggerPos = i;
LeftPos = 2*i;
RightPos = 2*i + 1;

if LeftPos <= N
    if A(BiggerPos,2) >= c(2) && A(LeftPos,2) < c(2)
        flag = 1;

    elseif A(BiggerPos,2) < c(2) && A(LeftPos,2) >= c(2)
        flag = 0;

    elseif A(BiggerPos,2) == c(2) && A(LeftPos,2) == c(2)
        if A(BiggerPos,1) > c(1)
            if A(LeftPos,1) > c(1)
                if A(BiggerPos,1) < A(LeftPos,1)
                    flag = 1;
                else
                    flag = 0;
                end
            else
                flag = 1;
            end

        else
            if A(LeftPos,1) < c(1)
                if A(BiggerPos,1) > A(LeftPos,1)
                    flag = 1;
                else
                    flag = 0;
                end
            else
                flag = 0;
            end
        end

    else
        det = (A(BiggerPos,1) - c(1))*(A(LeftPos,2) - c(2)) - (A(BiggerPos,2) - ...
            c(2))*(A(LeftPos,1) - c(1));

        if det > 0
            flag = 1;
        elseif det < 0
```

```
44              flag = 0;
45          else
46              if (A(BiggerPos,1) - c(1))^2 + (A(BiggerPos,2) - c(2))^2 ...
47                  < (A(LeftPos,1) - c(1))^2 + (A(LeftPos,2) - c(2))^2
48                  flag = 1;
49              else
50                  flag = 0;
51              end
52          end
53      end
54      if flag == 1
55          BiggerPos = LeftPos;
56      end
57  end
58
59  if RightPos <= N
60      if A(BiggerPos,2) >= c(2) && A(RightPos,2) < c(2)
61          flag = 1;
62
63      elseif A(BiggerPos,2) < c(2) && A(RightPos,2) >= c(2)
64          flag = 0;
65
66      elseif A(BiggerPos,2) == c(2) && A(RightPos,2) == c(2)
67          if A(BiggerPos,1) > c(1)
68              if A(RightPos,1) > c(1)
69                  if A(BiggerPos,1) < A(RightPos,1)
70                      flag = 1;
71                  else
72                      flag = 0;
73                  end
74              else
75                  flag = 1;
76              end
77
78          else
79              if A(RightPos,1) < c(1)
80                  if A(BiggerPos,1) > A(RightPos,1)
81                      flag = 1;
82                  else
83                      flag = 0;
84                  end
85              else
86                  flag = 0;
87              end
88          end
89
90      else
91          det = (A(BiggerPos,1) - c(1))*(A(RightPos,2) - c(2)) - (A(BiggerPos,2) - ...
92              c(2))*(A(RightPos,1) - c(1));
93
94          if det > 0
95              flag = 1;
96          elseif det < 0
97              flag = 0;
98          else
99              if (A(BiggerPos,1) - c(1))^2 + (A(BiggerPos,2) - c(2))^2 < ...
100                     (A(RightPos,1) - c(1))^2 + (A(RightPos,2) - c(2))^2
101                 flag = 1;
102             else
103                 flag = 0;
104             end
105         end
```

```
106     end
107     if flag == 1
108         BiggerPos = RightPos;
109     end
110 end
111
112 if BiggerPos ~= i
113     aux = A(i,:);
114     A(i,:) = A(BiggerPos,:);
115     A(BiggerPos,:) = aux;
116
117     A = Heapify_v1(A, N, BiggerPos, c);
118 end
119 end
```

Código B.2 Primera versión de la función auxiliar *Heapify* del algoritmo *HeapSort*.

B.2 Segunda versión del método HeapSort

La segunda versión del algoritmo de ordenación *HeapSort* se ha empleado para realizar la ordenación de las aristas que se encuentran almacenadas en el conjunto ordenado T. Para ello, se ha hecho uso tanto de las distancias de las aristas al nodo i almacenadas en ρ como del conjunto x_{comp} que almacena la componente en el eje x del vector característico normalizado de las aristas para desempatar en cuanto a la ordenación en el caso de que dos aristas presentasen la misma distancia al nodo i.

En este caso, las condiciones para la ordenación son sencillas:

1. Las aristas con menor distancia al nodo i irán primero.
2. En caso de empates, la arista con menor componente en el eje x de su vector característico normalizado irá primero.

La segunda versión de la función *HeapSort* recibirá como argumento de entrada una matriz A que, en este caso, se corresponde con la mostrada en 3.9.

A continuación, se muestra el código de las funciones desarrolladas para la implementación de la segunda versión del método *HeapSort* en este trabajo.

```
1  function A = HeapSort_v2(A)
2  Dim = size(A); N = Dim(1);
3  LastTopPos = floor(N/2);
4
5  for i = LastTopPos:-1:1
6      A = Heapify_v2(A, N, i);
7  end
8
9  for i = N-1:-1:1
10     aux = A(i+1,:);
11     A(i+1,:) = A(1,:);
12     A(1,:) = aux;
13     A = Heapify_v2(A, i, 1);
14 end
```

```
15 end
```

Código B.3 Segunda versión del algoritmo *HeapSort*.

```
1  function A = Heapify_v2(A, N, i)
2  rho = A(:, length(A(1,:)));
3  x_comp = A(:, length(A(1,:)) - 1);
4  BiggerPos = i;
5  LeftPos = 2*i;
6  RightPos = 2*i + 1;
7
8  if [LeftPos <= N && rho(BiggerPos) < rho(LeftPos)] || [LeftPos <= N && ...
9          rho(BiggerPos) == rho(LeftPos) && x_comp(BiggerPos) < x_comp(LeftPos)]
10     BiggerPos = LeftPos;
11 end
12
13 if [RightPos <= N && rho(BiggerPos) < rho(RightPos)] || [RightPos <=N && ...
14         rho(BiggerPos) == rho(RightPos) && x_comp(BiggerPos) < x_comp(RightPos)]
15     BiggerPos = RightPos;
16 end
17
18 if BiggerPos ~= i
19     aux = A(i,:);
20     A(i,:) = A(BiggerPos,:);
21     A(BiggerPos,:) = aux;
22
23     A = Heapify_v2(A, N, BiggerPos);
24 end
25 end
```

Código B.4 Segunda versión de la de la función auxiliar *Heapify* del algoritmo *HeapSort*.

Apéndice C
Algoritmo de *Búsqueda Binaria e Inserción*

En este apéndice, se muestra el algoritmo responsable de realizar la correcta inserción del conjunto de aristas obstáculo dadas en el conjunto *newedges* en T cuando ha sido requerido en el desarrollo del método de *Lee* con un costo computacional, en el peor de los casos, de $O(\log_2 n_T)$. Puesto que el objetivo de este trabajo no es la codificación del algoritmo de *Búsqueda Binaria*, no se va a explicar su implementación. No obstante, sí se va a señalar la diferencia del algoritmo empleado con respecto a los métodos de *Búsqueda Binaria* convencionales ([7]).

Un algoritmo de *Búsqueda Binaria* usual encuentra la posición de un determinado valor en un vector ordenado. La necesidad de este trabajo es encontrar en un determinado conjunto (ρ) la posición a insertar *newedges* en T, así como su distancia al nodo i en ρ.

Nótese que a lo largo de los algoritmos VISIBLEVERTICESVA(i,V) y VISIBLEVERTICESOD(i,V) nunca se actualiza la distancia de las aristas obstáculo almacenadas en T al nodo i desde su inserción inicial. Debido a ello, en este algoritmo será necesario recalcular la distancia de las aristas que se encuentren introducidas en T al nodo i, pero únicamente de aquellas con las que el algoritmo vaya realizando comprobaciones.

Para actualizar esta distancia, se volverá a aplicar el procedimiento descrito en la sección 3.2.3 basado en la Figura 3.15, donde, en este caso, e_1 sería la arista de T con la que se está realizando la comparación, y la búsqueda de las coordenadas de los nodos A y B se realizará directamente en la matriz $V_{Mercator}$.

A continuación, se muestra el código de la función desarrollada para la implementación del algoritmo de *Búsqueda Binaria e Inserción* en este trabajo. Nótese que en este algoritmo se emplean índices i y j pero su interpretación no tiene nada que ver con aquella asociada a los índices de los nodos que se ha estado empleando a lo largo del trabajo.

```
1 function [Pos, rho, T] = BinarySearchAndInsertion(rho, rho_k0, T, newedges, ...
2     V_Mercator, origin, destiny)
3 n_T = length(rho);
4 i = 1;
5 j = n_T;
6 while i <= j
7     medium = fix((i+j)/2);
```

```
8
9       %% Actualización de la distancia de la arista con la que se realiza la comparación
10      %% al nodo i
11
12      % Cálculo del vector v
13      v = destiny' - origin';
14
15      % Vectores iA y AB
16      iA = V_Mercator(T(medium),:)' - origin';
17      AB = V_Mercator(T(medium)+1, :)' - V_Mercator(T(medium),:)';
18
19      % Cálculo de beta
20      beta = (-iA(1)*AB(2) + iA(2)*AB(1))/(-v(1)*AB(2) + v(2)*AB(1));
21
22      % Vector dirigido del vértice i al punto de intersección P
23      iP = beta*v;
24
25      % Actualización de la distancia de la arista al nodo i
26      rho(medium) = norm(iP);
27
28      if rho(medium) < rho_k0
29          i = medium + 1; % Búsqueda en la mitad derecha
30      else
31          j = medium - 1; % Búsqueda en la mitad izquierda
32      end
33  end
34
35  % Actualización de los conjuntos rho y T y obtención de la posición de las nuevas
36  % aristas en T
37  rho = [rho(1:j); rho_k; rho(i:n_T)];
38  T = [T(1:j); newedges; T(i:n_T)];
39  Pos = j + 1 : j + length(rho_k0);
40  end
```

Código C.1 Algoritmo de *Búsqueda Binaria e Inserción*.

Apéndice D
Códigos de MATLAB

A lo largo del documento se han ido introduciendo trozos de código cuya finalidad era facilitar al lector la compresión de los algoritmos. No obstante, alguna de las rutinas desarrolladas no se han incluido a lo largo del documento debido a su similitud con otros programas expuestos.

En este apéndice se adjunta la totalidad de los códigos desarrollados en MATLAB por el autor en este trabajo.

D.1 Primera versión del algoritmo *Ingenuo*

```
1  clear all; close all; clc
2
3  %% Insertar puntos de origen y destino (Madrid y Kiev en este caso)
4  O = [40.471926, -3.562640]; D = [50.345001, 30.894699];
5
6  %% Insertar polígonos obstáculo
7
8  storms = load('TFG_Narciso.mat');
9  storms = struct2cell(storms);
10 V = [storms{1} storms{2}];
11
12 %% CÁLCULOS PRELIMINARES
13
14 V = [V; NaN(1,2); O; NaN(1,2); D]*pi/180;
15
16 % Matriz de WPs y orden del grafo
17 WPs = rmmissing(unique(V, 'rows', 'stable')); % Primero se suprimen vértices repetidos
18 % con el comando 'unique' y tras ello se suprimen los NaN con el comando 'rmmissing'
19 N = length(WPs);
20
21 % Implementación de la transformación de Mercator
22 Lon_0 = mean(WPs(:,2));
23 x_V = (V(:,2) - Lon_0); y_V = log(tan(pi/4 + V(:,1)/2)); V_Mercator = [x_V, y_V];
24 x_WPs = (WPs(:,2) - Lon_0); y_WPs = log(tan(pi/4 + WPs(:,1)/2)); WPs_Mercator = ...
25     [x_WPs, y_WPs];
26
27 % Inicialización de la matriz de adyacencia lógica
28 Conex = false(N,N);
29
30 %% CÓMPUTO DEL GRAFO DE VISIBILIDAD
```

```
31 for i = 1 : N - 1
32     for j = i + 1 : N
33         % Intersección entre la recta que une i con j y el conjunto de polígonos
34         % obstáculo
35         [xk, yk] = polyxpoly([WPs_Mercator(i,1), WPs_Mercator(j,1)], ...
36             [WPs_Mercator(i,2), WPs_Mercator(j,2)], V_Mercator(:,1), ...
37             V_Mercator(:,2),'unique');
38 
39         if length(xk) == 2 % Único caso en que se considera que i y j puedan ser
40         % visibles
41             % Cálculo del punto medio
42             midpoint = [(xk(1) + xk(2))/2, (yk(1) + yk(2))/2];
43 
44             % Se comprueba tanto si el punto medio está dentro del polígono como si
45             % está en una arista
46             [in, on] = inpolygon(midpoint(1), midpoint(2), V_Mercator(:,1), ...
47                 V_Mercator(:,2));
48             if in == 0 || (in == 1 && on == 1)
49                 Conex(i,j) = 1; Conex(j,i) = 1;
50             end
51         end
52     end
53 end
54 
55 %% OBTENCIÓN DEL CAMINO MÁS CORTO ENTRE O Y D
56 
57 % Creación de la matriz de adyacencia con pesos
58 [i_vec, j_vec] = find(Conex);
59 for n_c = 1:length(i_vec)
60     d_lox(n_c) = LoxodromicDistance(WPs(i_vec(n_c),1), WPs(i_vec(n_c),2), ...
61         WPs(j_vec(n_c),1), WPs(j_vec(n_c),2), 0);
62 end
63 
64 significant_numbers = 9; % Número de cifras significativas que se considera
65 factor = 10.^(significant_numbers - ceil(log10(abs(d_lox))));
66 d_lox = round(d_lox .* factor) ./ factor;
67 
68 Adj = sparse(i_vec,j_vec,d_lox);
69 
70 % Cálculo del camino más corto
71 G = graph(Adj);
72 [path, distance] = shortestpath(G, N - 1, N);
73 
74 %% OPERACIONES AUXILIARES AL ALGORITMO
75 
76 %%% Para trazar el grafo de visibilidad, descomentar las siguientes líneas de código
77 % figure(1)
78 % for i = 1 : N
79 %     for j = i : N
80 %         if Conex(i,j) == 1
81 %             geoplot([WPs(i,1), WPs(j,1)]*180/pi, [WPs(i,2), WPs(j,2)]*180/pi);
82 %             hold on
83 %         end
84 %     end
85 % end
86 %
87 %%% Para trazar el camino más corto, descomentar las siguientes líneas de código
88 % figure(2)
89 % geoplot(V(:,1)*180/pi, V(:,2)*180/pi,'b');
90 % hold on
91 % geoplot(WPs(path,1)*180/pi, WPs(path,2)*180/pi,'r');
92 % hold on
```

```
93 % for i=1:length(path)
94 %     geoplot(WPs(path(i),1)*180/pi,WPs(path(i),2)*180/pi,'rd')
95 % end
```

Código D.1 Primera versión del algoritmo *Ingenuo*.

D.2 Segunda versión del algoritmo *Ingenuo*

```
1  close all; clear all; clc;
2
3  %% Insertar puntos de origen y destino (Bruselas y Sarajevo en este caso)
4  O = [50.901402, 4.484440]; D = [43.824600, 18.331499];
5
6  %% Insertar polígonos obstáculo
7
8  storms = load('TFG_Narciso.mat');
9  storms = struct2cell(storms);
10 V = [storms{1} storms{2}];
11
12 %% CÁLCULOS PRELIMINARES
13
14 V = [V; NaN(1,2); O; NaN(1,2); D]*pi/180;
15
16 % Vector que contiene los índices de la matriz V donde se sitúan los NaN's
17 W = find(isnan(V(:,1)));
18
19 % Matriz de WPs y orden del grafo
20 V_O = V; V_O(W,:) = []; % Primero se suprimen los NaN
21 WPs = unique(V_O,'rows','stable'); % Seguidamente se suprimen los vértices repetidos
22 N = length(WPs);
23
24 % Implementación de la transformación de Mercator
25 Lon_0 = mean(WPs(:,2));
26 x_V = (V(:,2) - Lon_0); y_V = log(tan(pi/4 + V(:,1)/2)); V_Mercator = [x_V, y_V];
27 x_WPs = (WPs(:,2) - Lon_0); y_WPs = log(tan(pi/4 + WPs(:,1)/2)); WPs_Mercator = ...
28     [x_WPs, y_WPs];
29
30 % Inicialización de la matriz de adyacencia lógica
31 Conex = false(N,N);
32
33 %% CÓMPUTO DEL GRAFO DE VISIBILIDAD
34
35 p = 0; % Inicialización del contador asociado al número de polígonos
36 for i = 1 : N - 1
37     if i <= N - 2 % Para no formar la base no ortogonal con el punto de origen O
38
39         % Formación de la base no ortogonal de vectores
40         if i == 1 || i == W(p) - 2*p + 1 % Caso en que i es el primer vértice del
41         % primer polígono o del resto de polígonos (téngase en cuenta que cuando p = 0,
42         % i = 1. Cuando se compruebe si i == 1, no se entrará a mirar si
43         % i == W(p) - 2*p + 1, por lo que no se no se evaluará W(p) cuando p = 0)
44             p = p + 1;
45             r = WPs_Mercator(i+1,:)' - WPs_Mercator(i,:)';
46
47             if p == 1 % Caso en que i sea el primer vértice del primer polígono
48                 s = WPs_Mercator(i - 1 + W(p) - 2,:)' - WPs_Mercator(i,:)';
49             else % Caso en que i sea el primer vértice del resto de polígonos
```

```
50                  s = WPs_Mercator(i - 1 + W(p) - W(p-1) - 2,:)' - WPs_Mercator(i,:)';
51              end
52
53          elseif i == W(p) - 2*p % Caso en que i es el último vértice de algún polígono
54              if p == 1 % Caso en que i sea el último vértice del primer polígono
55                  r = WPs_Mercator(1, :)' - WPs_Mercator(i,:)';
56              else % Caso en que i sea el último vértice del resto de polígonos
57                  r = WPs_Mercator(i + 1 - (W(p) - W(p-1) - 2), :)' - WPs_Mercator(i,:)';
58              end
59              s = WPs_Mercator(i - 1,:)' - WPs_Mercator(i,:)';
60
61          else % Caso genérico
62              r = WPs_Mercator(i+1,:)' - WPs_Mercator(i,:)';
63              s = WPs_Mercator(i-1,:)' - WPs_Mercator(i,:)';
64          end
65
66          % Producto vectorial de r y s
67          q_z = r(1)*s(2) - r(2)*s(1);
68      end
69
70      for j = i + 1 : N
71          vis = 0; % vis = 0 si los vértices no son mutuamente visibles y vis = 1 si se
72          % demuestra que son mutuamente visibles
73
74          [xk, yk] = polyxpoly([WPs_Mercator(i,1) WPs_Mercator(j,1)], ...
75              [WPs_Mercator(i,2) WPs_Mercator(j,2)], V_Mercator(:,1), ...
76              V_Mercator(:,2),'unique');
77
78          if length(xk) == 2
79              if i == N - 1
80                  vis = 1;
81              else
82
83                  % Cálculo del vector v
84                  v = WPs_Mercator(j,:)' - WPs_Mercator(i,:)';
85
86                  % Resolución del sistema de ecuaciones
87                  a = (v(1)*s(2) - v(2)*s(1))/q_z;
88                  b = (r(1)*v(2) - r(2)*v(1))/q_z;
89
90                  % Aplicación del criterio con una cierta tolerancia
91                  if sign(q_z) == -1 && ~(a > 1e-12 && b > 1e-12)
92                      vis = 1;
93                  elseif sign(q_z) == 1 && (a >= -1e-12 && b >= -1e-12)
94                      vis = 1;
95                  end
96              end
97          end
98          if vis == 1
99              Conex(i,j) = 1; Conex(j,i) = 1;
100         end
101     end
102 end
103
104 %% OBTENCIÓN DEL CAMINO MÁS CORTO ENTRE O Y D
105
106 % Creación de la matriz de adyacencia con pesos
107 [i_vec, j_vec] = find(Conex);
108 for n_c = 1:length(i_vec)
109     d_lox(n_c) = LoxodromicDistance(WPs(i_vec(n_c),1), WPs(i_vec(n_c),2), ...
110         WPs(j_vec(n_c),1), WPs(j_vec(n_c),2), 0);
111 end
```

```
112
113 significant_numbers = 9; % Número de cifras significativas que se considera
114 factor = 10.^(significant_numbers - ceil(log10(abs(d_lox))));
115 d_lox = round(d_lox .* factor) ./ factor;
116
117 Adj = sparse(i_vec,j_vec,d_lox);
118
119 % Cálculo del camino más corto
120 G = graph(Adj);
121 [path, distance] = shortestpath(G, N - 1, N);
122
123 %% OPERACIONES AUXILIARES AL ALGORITMO
124
125 %%% Para trazar el grafo de visibilidad, descomentar las siguientes líneas de código
126 % figure(1)
127 % for i = 1 : N
128 %     for j = i : N
129 %         if Conex(i,j) == 1
130 %             geoplot([WPs(i,1), WPs(j,1)]*180/pi, [WPs(i,2), WPs(j,2)]*180/pi);
131 %             hold on
132 %         end
133 %     end
134 % end
135 %
136 %%% Para trazar el camino más corto, descomentar las siguientes líneas de código
137 % figure(2)
138 % geoplot(V(:,1)*180/pi, V(:,2)*180/pi,'b');
139 % hold on
140 % geoplot(WPs(path,1)*180/pi, WPs(path,2)*180/pi,'r');
141 % hold on
142 % for i=1:length(path)
143 %     geoplot(WPs(path(i),1)*180/pi,WPs(path(i),2)*180/pi,'rd')
144 % end
```

Código D.2 Segunda versión del algoritmo *Ingenuo*.

D.3 Algoritmo de *Lee*

D.3.1 Programa principal

```
1 clear all; close all; clc;
2
3 %% Insertar puntos de origen y destino (Puntos artificiales en este caso)
4 O = [43, -0.86]; D = [44.7, 6.75];
5
6 %% Insertar polígonos obstáculo
7
8 storms = load('TFG_Narciso.mat');
9 storms = struct2cell(storms);
10 V = [storms{1} storms{2}];
11
12 %% CÁLCULOS PRELIMINARES
13
14 V = [V; NaN(1,2); O ; NaN(1,2); D]*pi/180; N_in = length(V);
15
16 % Vector que contiene los índices de la matriz V donde se sitúan los NaN's y obtención
```

```
17 % del número de polígonos obstáculo del grafo
18 W = find(isnan(V(:,1)));
19 npol = length(W) - 1;
20
21 % Matriz de WPs, orden del grafo y número de aristas obstáculo del grafo
22 V_0 = V; V_0(W,:) = []; % Primero se suprimen los NaN
23 WPs = unique(V_0, 'rows', 'stable'); % Seguidamente se suprimen los vértices repetidos
24 N = length(WPs); n = N - 2;
25
26 % Implementación de la transformación de Mercator
27 Lon_0 = mean(WPs(:,2));
28 x_V = (V(:,2) - Lon_0); y_V = log(tan(pi/4 + V(:,1)/2)); V_Mercator = [x_V, y_V];
29 x_WPs = (WPs(:,2) - Lon_0); y_WPs = log(tan(pi/4 + WPs(:,1)/2)); WPs_Mercator = ...
30     [x_WPs, y_WPs];
31
32 % Vector de índices de los puntos del grafo
33 index = 1:N;
34
35 % Cálculo de las matrices E y Einv
36 E = NaN(length(V),2); Einv = zeros(N,2);
37 for p = 1:npol
38     if p == 1
39         E(1: W(p) - 2,:) = [1 : W(p) - 2; 2 : W(p) - 2, 1]';
40         Einv(1: W(p) - 2,:) = [1 : W(p) - 2; W(p) - 2, 1 : W(p) - 3]';
41     else
42         E(W(p-1) + 1: W(p) - 2,:) = [W(p-1) + 1 - 2*(p-1) : W(p) - 2*p; ...
43             W(p-1) + 2 - 2*(p-1) : W(p) - 2*p, W(p-1) + 1 - 2*(p-1)]';
44         Einv(W(p-1) - 2*(p-1) + 1 : W(p) - 2*p,:) = [W(p-1) + 1 : W(p) - 2; ...
45             W(p) - 2, W(p-1) + 1 : W(p) - 3]';
46     end
47 end
48
49 % Inicialización de la matriz de adyacencia lógica
50 Conex = false(N,N);
51
52 %% CÓMPUTO DEL GRAFO DE VISIBILIDAD
53 for i = 1:N
54     if i <= n
55         Conex = VisibleVerticesVA(i, V_Mercator(1:N_in - 4,:), WPs_Mercator(1:n,:), ...
56             index(1:n), E, Einv, Conex, N_in);
57     else
58         Conex = VisibleVerticesOD(i, V_Mercator, WPs_Mercator, N, index, E, Einv, ...
59             Conex, N_in);
60     end
61 end
62
63 %% OBTENCIÓN DEL CAMINO MÁS CORTO ENTRE O Y D
64
65 % Creación de la matriz de adyacencia con pesos
66 [i_vec, j_vec] = find(Conex);
67 for n_c = 1:length(i_vec)
68     d_lox(n_c) = LoxodromicDistance(WPs(i_vec(n_c),1), WPs(i_vec(n_c),2), ...
69         WPs(j_vec(n_c),1), WPs(j_vec(n_c),2), 0);
70 end
71
72 significant_numbers = 9; % Número de cifras significativas que se considera
73 factor = 10.^(significant_numbers - ceil(log10(abs(d_lox))));
74 d_lox = round(d_lox .* factor) ./ factor;
75
76 Adj = sparse(i_vec,j_vec,d_lox);
77
78 % Cálculo del camino más corto
```

```
79 G = graph(Adj);
80 [path, distance] = shortestpath(G, N - 1, N);
81
82 %% OPERACIONES AUXILIARES AL ALGORITMO
83
84 %%% Para trazar el grafo de visibilidad, descomentar las siguientes líneas de código
85 % figure(1)
86 % for i = 1 : N
87 %     for j = i : N
88 %         if Conex(i,j) == 1
89 %             geoplot([WPs(i,1), WPs(j,1)]*180/pi, [WPs(i,2), WPs(j,2)]*180/pi);
90 %             hold on
91 %         end
92 %     end
93 % end
94 %
95 %%% Para trazar el camino más corto, descomentar las siguientes líneas de código
96 % figure(2)
97 % geoplot(V(:,1)*180/pi, V(:,2)*180/pi,'b');
98 % hold on
99 % geoplot(WPs(path,1)*180/pi, WPs(path,2)*180/pi,'r');
100 % hold on
101 % for i=1:length(path)
102 %     geoplot(WPs(path(i),1)*180/pi,WPs(path(i),2)*180/pi,'rd')
103 % end
```

Código D.3 Programa principal del algoritmo de *Lee*.

D.3.2 Función VisibleVerticesVA(i,V)

```
1 function Conex = VisibleVerticesVA(i, V_Mercator, WPs_Mercator, N, index, E, Einv, ...
2     Conex, N_in)
3
4 %% ORDENACIÓN DE VÉRTICES EN SENTIDO ANTIHORARIO ALREDEDOR DEL NODO i
5
6 VerticesData = [WPs_Mercator, index'];
7 VerticesData(i,:) = [];
8 VerticesData((VerticesData(:,2) - WPs_Mercator(i,2)) < 0,:) = [];
9 %
10 % Caso en el que solo queda un vértice restante, para el que se comprobará directamente
11 % la visibilidad
12 if length(VerticesData(:,3)) == 1
13
14     j = VerticesData(3);
15
16     % Cálculo del vector v
17     v = (WPs_Mercator(j, :) - WPs_Mercator(i, :))';
18
19     % Formación de la base no ortogonal de vectores
20     r = (WPs_Mercator(E(Einv(i,1), 2),:) - WPs_Mercator(i,:))';
21     s = (WPs_Mercator(E(Einv(i,2), 1),:) - WPs_Mercator(i,:))';
22
23     % Producto vectorial de r y s
24     q_z = r(1)*s(2) - r(2)*s(1);
25
26     % Resolución del sistema de ecuaciones
27     a = (v(1)*s(2) - v(2)*s(1))/q_z;
28     b = (r(1)*v(2) - r(2)*v(1))/q_z;
```

```
29
30      % Aplicación del criterio con una cierta tolerancia
31      if (sign(q_z) == -1 && ~(a>1e-12 && b>1e-12)] || (sign(q_z) == 1 && ...
32              (a>=-1e-12 && b>=-1e-12))
33          Conex(i,j) = 1; Conex(j,i) = 1;
34      end
35
36  % Caso en que queda más de un vértice, para el que se recorrerá el algoritmo completo
37  elseif length(VerticesData(:,3)) > 1
38
39      %% ORDENACIÓN DE VÉRTICES EN SENTIDO ANTIHORARIO
40      VerticesData= HeapSort_v1(VerticesData, WPs_Mercator(i,:));
41      j = VerticesData(:,3);
42
43      %% INICIALIZACIÓN DEL CONJUNTO ORDENADO T
44
45      % Intersección del segmento sigmagorro con todos los polígonos
46      [xk, yk, K] = polyxpoly([WPs_Mercator(i,1), WPs_Mercator(i,1) + 2*pi], ...
47          [WPs_Mercator(i,2), WPs_Mercator(i,2)], V_Mercator(:,1), V_Mercator(:,2));
48
49      % Inicialización de estructuras para introducir el conjunto ordenado T, distancia
50      % de las aristas en T al punto i en rho, posición de las aristas de T en Tinv y
51      % la componente x del vector característico de las aristas en x_comp
52      T = zeros(0,1); rho = zeros(0,1); Tinv = NaN(N_in,1); x_comp = zeros(0,1);
53
54      % Bucle para tratar las aristas que se deban insertar en T
55      for k0 = 1 : length(xk)
56
57          % Primera comprobación: que el punto de intersección no sea el punto i del que
58          % parte el segmento sigmagorro
59          if xk(k0) ~= WPs_Mercator(i,1)
60
61              % Arista de intersección
62              k = K(k0,2);
63
64              % Segunda comprobación: si la intersección es justamente un vértice o si es
65              % un vértice repetido
66              isvertice = 0; % Variable que se igualará a 1 si se detecta el caso de
67              % vértice
68              isverticerep = 0; % Variable que se igualará a 1 si se detecta el caso de
69              % vértice repetido
70
71              if V_Mercator(k,1) == xk(k0) && V_Mercator(k,2) == yk(k0)
72                  isvertice = 1;
73              elseif V_Mercator(k+1,1) == xk(k0) && V_Mercator(k+1,2) == yk(k0)
74                  isverticerep = 1;
75              end
76
77              if isvertice == 1
78                  % Caso en que la intersección es un vértice
79                  kminus = Einv(E(k,1),2);
80
81                  % Cálculo del vector t
82                  t = V_Mercator(kminus,:) - V_Mercator(k,:); t_0 = t/norm(t);
83
84                  % Para incluir la arista o no en T, se observa la componente en el eje
85                  % y del vector t
86                  if t_0(2) > 0
87
88                      % Introducción de la arista correspondiente en T
89                      T = [T; kminus];
90
```

```
91                      % Cálculo de la distancia de la arista al punto i
92                      rho_k0 = xk(k0) - WPs_Mercator(i,1);
93                      rho = [rho; rho_k0];
94
95                      % Almacenamiento de la componente x del vector t normalizado por si
96                      % se da el caso de que haya que desempatar en cuanto a la
97                      % ordenación en T
98                      x_comp = [x_comp; t_0(1)];
99                  end
100
101                 % Las operaciones con el vector u son análogas a las realizadas con el
102                 % vector t
103                 u = V_Mercator(k + 1,:) - V_Mercator(k,:); u_0 = u/norm(u);
104                 if u_0(2) > 0
105                     T = [T; k];
106                     rho_k0 = xk(k0) - WPs_Mercator(i,1);
107                     rho = [rho; rho_k0];
108                     x_comp = [x_comp; u_0(1)];
109                 end
110
111             elseif isverticerep ~= 1
112                 % La intersección ya no puede ser un vértice
113                 T = [T; k];
114                 rho_k0 = xk(k0) - WPs_Mercator(i,1);
115                 rho = [rho; rho_k0];
116                 x_comp = [x_comp; NaN]; % Puesto que en este caso no será necesario
117                 % desempatar en cuanto a la ordenación en T, se introduce un NaN
118             end
119         end
120     end
121
122     TreeData = [T, x_comp, rho];
123
124     % Ordenación a través del método HeapSort
125     TreeData = HeapSort_v2(TreeData);
126     T = TreeData(:,1); rho = TreeData(:,3);
127
128     % Relleno del conjunto Tinv
129     for y = 1:length(T)
130         Tinv(T(y)) = y;
131     end
132
133     %% COMPROBACIÓN DE LA VISIBILIDAD DE LOS VÉRTICES
134
135     for j0 = 1:length(j)
136         if j0  == 1 || (j0 > 1 && (WPs_Mercator(j(j0-1),1) - WPs_Mercator(i,1))* ...
137                 (WPs_Mercator(j(j0),2) - WPs_Mercator(i,2)) ~= ...
138                 (WPs_Mercator(j(j0),1) - WPs_Mercator(i,1))* ...
139                 (WPs_Mercator(j(j0-1),2) - WPs_Mercator(i,2))) % No se considerarán
140                 % casos redundantes
141
142             if isempty(T)
143                 vis = 1;
144             else
145                 vis = VisibleVA(j(j0), i, WPs_Mercator, T(1), E, Einv);
146             end
147             if vis == 1
148                 Conex(i,j(j0)) = 1; Conex(j(j0),i) = 1;
149             end
150         end
151
152         if ~(WPs_Mercator(j(j0), 2) == WPs_Mercator(i,2) && WPs_Mercator(j(j0),1) ...
```

```
153                     > WPs_Mercator(i,1)) && j0 < length(j) % Porque en el primer caso ya se
154                     % ha realizado la inicialización de T y para el último vértice no es
155                     % necesario actualizar % T pues no se comprobará la visibilidad
156                     % de ningún otro nodo
157
158                 %% ACTUALIZACIÓN DE T
159
160                 % Búsqueda de la existencia en T tanto de la arista k como de la arista
161                 % k^{-}
162                 k = Einv(j(j0),1); kminus = Einv(j(j0),2);
163                 pos_k = Tinv(k); pos_kminus = Tinv(kminus);
164
165                 % Caso en que sea necesario eliminar ambas aristas en T
166                 if ~isnan(pos_k) && ~isnan(pos_kminus)
167
168                     Tinv([k; kminus]) = NaN;
169                     if pos_kminus > pos_k
170                         Tinv(T(pos_kminus + 1:length(T))) = Tinv(T(pos_kminus + ...
171                             1:length(T))) - 2;
172                     else
173                         Tinv(T(pos_k + 1:length(T))) = Tinv(T(pos_k + 1:length(T))) - 2;
174                     end
175                     T([pos_k; pos_kminus]) = [];
176                     rho([pos_k; pos_kminus]) = [];
177
178                 % Casos en que sea necesario eliminar una arista e introducir otra (si esta
179                 % no va en la dirección del segmento ij_{j0})
180                 elseif ~isnan(pos_k) && isnan(pos_kminus)
181
182                     % Vector director del segmento ij_{j0}
183                     v = (WPs_Mercator(j(j0),:) - WPs_Mercator(i,:))'; v_0 = v/norm(v);
184
185                     % Vector t
186                     t = V_Mercator(kminus,:) - V_Mercator(k,:); t_0 = t/norm(t);
187
188                     if (abs(t_0(1) - v_0(1)) < 1e-12 && abs(t_0(2) - v_0(2)) < 1e-12) ...
189                             || (abs(t_0(1) + v_0(1)) < 1e-12 && abs(t_0(2) + v_0(2)) ...
190                             < 1e-12) % Se ha aplicado una cierta tolerancia
191                         % Caso en que solo se suprime una arista
192                         Tinv(k) = NaN; Tinv(T(pos_k + 1:length(T))) = Tinv(T(pos_k + 1:...
193                             length(T))) - 1;
194                         T(pos_k) = [];
195                         rho(pos_k) = [];
196                     else
197                         % Caso en que se sustituye una arista por la otra
198                         Tinv(k) = NaN; Tinv(kminus) = pos_k;
199                         T(pos_k) = kminus;
200                     end
201
202                 % Caso análogo al anterior
203                 elseif isnan(pos_k) && ~isnan(pos_kminus)
204                     v = (WPs_Mercator(j(j0),:) - WPs_Mercator(i,:))'; v_0 = v/norm(v);
205                     u = V_Mercator(k+1,:) - V_Mercator(k,:); u_0 = u/norm(u);
206
207                     if (abs(u_0(1) - v_0(1)) < 1e-12 && abs(u_0(2) - v_0(2)) < 1e-12) ...
208                             || (abs(u_0(1) + v_0(1)) < 1e-12 && abs(u_0(2) + v_0(2)) ...
209                             < 1e-12)
210                         Tinv(kminus) = NaN; Tinv(T(pos_kminus + 1:length(T))) = ...
211                             Tinv(T(pos_kminus + 1:length(T))) - 1;
212                         T(pos_kminus) = [];
213                         rho(pos_kminus) = [];
214                     else
```

```
215                     Tinv(kminus) = NaN; Tinv(k) = pos_kminus;
216                     T(pos_kminus) = k;
217                 end
218 
219             % Caso en que hay que añadir dos aristas (si la dirección de estas no va
220             % en la del segmento ij_{j0})
221             else
222 
223                 % Vector director del segmento ij_{j0}
224                 v = (WPs_Mercator(j(j0),:) - WPs_Mercator(i,:))'; v_0 = v/norm(v);
225 
226                 % Vectores u y t
227                 u = V_Mercator(k + 1,:) - V_Mercator(k,:); u_0 = u/norm(u);
228                 t = V_Mercator(kminus,:) - V_Mercator(k,:); t_0 = t/norm(t);
229 
230                 % Distancia al nodo i
231                 rho_k0 = norm(v);
232 
233                 % Definición de las aristas a introducir en T
234                 if (abs(u_0(1) - v_0(1)) < 1e-12 && abs(u_0(2) - v_0(2)) < 1e-12) ...
235                         || (abs(u_0(1) + v_0(1)) < 1e-12 && abs(u_0(2) + v_0(2)) ...
236                         < 1e-12)
237                     newedges = kminus; % Solo se añadirá una arista
238 
239                 elseif (abs(t_0(1) - v_0(1)) < 1e-12 && abs(t_0(2) - v_0(2)) ...
240                         < 1e-14) || (abs(t_0(1) + v_0(1)) < 1e-14 && abs(t_0(2) + ...
241                         v_0(2)) < 1e-12)
242                     newedges = k; % Solo se añadirá una arista
243 
244                 else
245 
246                     %Se añadirán dos aristas
247                     rho_k0 = [rho_k0; rho_k0];
248 
249                     % Las dos aristas se ordenan a través del producto escalar
250                     udotv = dot(u_0,v);
251                     tdotv = dot(t_0,v);
252 
253                     if udotv > tdotv
254                         newedges = [kminus; k];
255                     else
256                         newedges = [k; kminus];
257                     end
258 
259                 end
260 
261                 % Introducción en T y actualización de rho y Tinv
262                 if isempty(T)
263                     T = newedges;
264                     rho = rho_k0;
265                     Tinv(T(1:length(newedges))) = 1:length(newedges);
266                 else
267                     [pos_newedges, rho, T] = BinarySearchAndInsertion(rho, rho_k0, ...
268                         T, newedges, V_Mercator, WPs_Mercator(i,:), ...
269                         WPs_Mercator(j(j0),:));
270                     Tinv(T(pos_newedges)) = pos_newedges;
271                     Tinv(T(pos_newedges(length(pos_newedges)) + 1:length(T))) = ...
272                         Tinv(T(pos_newedges(length(pos_newedges)) + 1:length(T))) ...
273                         + length(pos_newedges);
274                 end
275             end
276         end
```

```
277     end
278 end
279 end
```

Código D.4 Función VisibleVerticesVA(i,V).

D.3.3 Función VisibleVerticesOD(i,V)

```
1  function Conex = VisibleVerticesOD(i, V_Mercator, WPs_Mercator, N, index, E, Einv, ...
2      Conex, N_in)
3  
4  %% ORDENACIÓN DE VÉRTICES EN SENTIDO ANTIHORARIO
5  
6  VerticesData = [WPs_Mercator, index'];
7  VerticesData(i,:) = [];
8  
9  % En esta función, siempre será necesario recorrer el algoritmo completo
10 VerticesData= HeapSort_v1(VerticesData, WPs_Mercator(i,:));
11 j = VerticesData(:,3);
12 
13 %% INICIALIZACIÓN DEL CONJUNTO ORDENADO T
14 
15 % Intersección del segmento sigmagorro con todos los polígonos
16 [xk, yk, K] = polyxpoly([WPs_Mercator(i,1), WPs_Mercator(i,1) + 2*pi], ...
17     [WPs_Mercator(i,2), WPs_Mercator(i,2)], V_Mercator(:,1), V_Mercator(:,2));
18 
19 % Inicialización de estructuras para introducir el conjunto ordenado T, distancia de
20 % las aristas en T al punto i en rho, posición de las aristas de T en Tinv y la
21 % componente x del vector característico de las aristas en x_comp
22 T = zeros(0,1); rho = zeros(0,1); Tinv = NaN(N_in,1); x_comp = zeros(0,1);
23 
24 % Bucle para tratar las aristas que se deban insertar en T
25 for k0 = 1 : length(xk)
26 
27     N = length(WPs_Mercator);
28     % En este caso la primera comprobación es que el punto de intersección no ni sea el
29     % nodo de origen O ni el de destino D
30     if xk(k0) ~= WPs_Mercator(N, 1) && xk(k0) ~= WPs_Mercator(N-1, 1)
31 
32         % Arista de intersección
33         k = K(k0,2);
34 
35         % Segunda comprobación: si la intersección es justamente un vértice o si es un
36         % vértice repetido
37         isvertice = 0; % Variable que se igualará a 1 si se detecta el caso de vértice
38         isverticerep = 0; % Variable que se igualará a 1 si se detecta el caso de
39         % vértice repetido
40 
41         if V_Mercator(k,1) == xk(k0) && V_Mercator(k,2) == yk(k0)
42             isvertice = 1;
43         elseif V_Mercator(k + 1,1) == xk(k0) && V_Mercator(k + 1,2) == yk(k0)
44             isverticerep = 1;
45         end
46 
47         if isvertice == 1
48             % Caso en que la intersección es un vértice
49             kminus = Einv(E(k,1),2);
50 
```

```
51              % Cálculo del vector t
52              t = V_Mercator(kminus,:) - V_Mercator(k,:); t_0 = t/norm(t);
53
54              % Para incluir la arista o no en T, se observa la componente en el eje y
55              % del vector t
56              if t_0(2) > 0
57
58                  % Introducción de la arista correspondiente en T
59                  T = [T; kminus];
60
61                  % Cálculo de la distancia de la arista al punto i
62                  rho_k0 = xk(k0) - WPs_Mercator(i,1);
63                  rho = [rho; rho_k0];
64
65                  % Almacenamiento de la componente x del vector t normalizado por si se
66                  % da el caso de que haya que desempatar en cuanto a la ordenación en T
67                  x_comp = [x_comp; t_0(1)];
68              end
69
70              % Las operaciones con el vector u son análogas a las realizadas con el
71              % vector t
72              u = V_Mercator(k + 1,:) - V_Mercator(k,:); u_0 = u/norm(u);
73              if u_0(2) > 0
74                  T = [T; k];
75                  rho_k0 = xk(k0) - WPs_Mercator(i,1);
76                  rho = [rho; rho_k0];
77                  x_comp = [x_comp; u_0(1)];
78              end
79
80          elseif isverticerep ~= 1
81              % La intersección ya no puede ser un vértice
82              T = [T; k];
83              rho_k0 = xk(k0) - WPs_Mercator(i,1);
84              rho = [rho; rho_k0];
85              x_comp = [x_comp; NaN]; % Puesto que en este caso no será necesario
86              % desempatar en cuanto a la ordenación en T, se introduce un NaN
87          end
88      end
89  end
90
91  TreeData = [T, x_comp, rho];
92
93  % Ordenación a través del método HeapSort
94  TreeData = HeapSort_v2(TreeData);
95  T = TreeData(:,1); rho = TreeData(:,3);
96
97  % Se rellena el conjunto Tinv
98  for y = 1:length(T)
99      Tinv(T(y)) = y;
100 end
101
102 %% COMPROBACIÓN DE LA VISIBILIDAD DE LOS VÉRTICES
103
104 for j0 = 1:length(j)
105     if j0  == 1 || (j0 > 1 && (WPs_Mercator(j(j0-1),1) - WPs_Mercator(i,1))*...
106             (WPs_Mercator(j(j0),2) - WPs_Mercator(i,2)) ~= (WPs_Mercator(j(j0),1) - ...
107             WPs_Mercator(i,1))*(WPs_Mercator(j(j0-1),2) - WPs_Mercator(i,2))) % No se
108             % considerarán casos redundantes
109
110         if isempty(T)
111             vis = 1;
112         else
```

```
113             vis = VisibleOD(j(j0), i, WPs_Mercator, T(1), E);
114         end
115     end
116     if vis == 1
117         Conex(i,j(j0)) = 1; Conex(j(j0),i) = 1;
118     end
119 
120     if (WPs_Mercator(j(j0), 2) ~= WPs_Mercator(i,2) || WPs_Mercator(j(j0),1) < ...
121             WPs_Mercator(i,1)) && j0 < length(j) && j(j0) ~= N && j(j0) ~= N-1 % Porque
122             % en el primer caso ya se ha realizado la inicialización de T y para el
123             % último vértice no es necesario actualizar T pues no se comprobará la
124             % visibilidad de ningún otro vértice
125 
126         %% ACTUALIZACIÓN DE T
127 
128         % Búsqueda de la existencia en T tanto de la arista k como de la arista k^{-}
129         k = Einv(j(j0),1); kminus = Einv(j(j0),2);
130         pos_k = Tinv(k); pos_kminus = Tinv(kminus);
131 
132         % Caso en que sea necesario eliminar ambas aristas en T
133         if ~isnan(pos_k) && ~isnan(pos_kminus)
134 
135             Tinv([k; kminus]) = NaN;
136             if pos_kminus > pos_k
137                 Tinv(T(pos_kminus + 1:length(T))) = Tinv(T(pos_kminus + 1: ...
138                 length(T))) - 2;
139             else
140                 Tinv(T(pos_k + 1:length(T))) = Tinv(T(pos_k + 1:length(T))) - 2;
141             end
142             T([pos_k; pos_kminus]) = [];
143             rho([pos_k; pos_kminus]) = [];
144 
145         % Casos en que sea necesario eliminar una arista e introducir otra (si esta no
146         % va en la dirección del segmento ij_{j0})
147         elseif ~isnan(pos_k) && isnan(pos_kminus)
148 
149             % Vector director del segmento ij_{j0}
150             v = (WPs_Mercator(j(j0),:) - WPs_Mercator(i,:))'; v_0 = v/norm(v);
151 
152             % Vector t
153             t = V_Mercator(kminus,:) - V_Mercator(k,:); t_0 = t/norm(t);
154 
155             if (abs(t_0(1) - v_0(1)) < 1e-12 && abs(t_0(2) - v_0(2)) < 1e-12) || ...
156                 (abs(t_0(1) + v_0(1)) < 1e-12 && abs(t_0(2) + v_0(2)) < 1e-12) % Se ha
157                     % aplicado una cierta tolerancia
158                 % Caso en que solo se suprime una arista
159                 Tinv(k) = NaN; Tinv(T(pos_k + 1:length(T))) = Tinv(T(pos_k + 1: ...
160                 length(T))) - 1;
161                 T(pos_k) = [];
162                 rho(pos_k) = [];
163             else
164                 % Caso en que se sustituye una arista por la otra
165                 Tinv(k) = NaN; Tinv(kminus) = pos_k;
166                 T(pos_k) = kminus;
167             end
168 
169         % Caso análogo al anterior
170         elseif isnan(pos_k) && ~isnan(pos_kminus)
171             v = (WPs_Mercator(j(j0),:) - WPs_Mercator(i,:))'; v_0 = v/norm(v);
172             u = V_Mercator(k+1,:) - V_Mercator(k,:); u_0 = u/norm(u);
173 
174             if (abs(u_0(1) - v_0(1)) < 1e-12 && abs(u_0(2) - v_0(2)) < 1e-12) || ...
```

```
175                    (abs(u_0(1) + v_0(1)) < 1e-12 && abs(u_0(2) + v_0(2)) < 1e-12)
176                Tinv(kminus) = NaN; Tinv(T(pos_kminus + 1:length(T))) = ...
177                Tinv(T(pos_kminus + 1:length(T))) - 1;
178                T(pos_kminus) = [];
179                rho(pos_kminus) = [];
180            else
181                Tinv(kminus) = NaN; Tinv(k) = pos_kminus;
182                T(pos_kminus) = k;
183            end
184
185        % Caso en que hay que añadir dos aristas (si la dirección de estas no va en la
186        % del segmento ij_{j0})
187        else
188
189            % Vector director del segmento ij_{j0}
190            v = (WPs_Mercator(j(j0),:) - WPs_Mercator(i,:))'; v_0 = v/norm(v);
191
192            % Vectores u y t
193            u = V_Mercator(k + 1,:) - V_Mercator(k,:); u_0 = u/norm(u);
194            t = V_Mercator(kminus,:) - V_Mercator(k,:); t_0 = t/norm(t);
195
196            % Distancia al nodo i
197            rho_k0 = norm(v);
198
199            % Definición de las aristas a introducir en T
200            if (abs(u_0(1) - v_0(1)) < 1e-12 && abs(u_0(2) - v_0(2)) < 1e-12) || ...
201                    (abs(u_0(1) + v_0(1)) < 1e-12 && abs(u_0(2) + v_0(2)) < 1e-12)
202                newedges = kminus; % Solo se añadirá una arista
203
204            elseif (abs(t_0(1) - v_0(1)) < 1e-12 && abs(t_0(2) - v_0(2)) < 1e-12) ...
205                    || (abs(t_0(1) + v_0(1)) < 1e-12 && abs(t_0(2) + v_0(2)) < 1e-12)
206                newedges = k; % Solo se añadirá una arista
207
208            else
209
210                %Se añadirán dos aristas
211                rho_k0 = [rho_k0; rho_k0];
212
213                % Las dos aristas se ordenan a través del producto escalar
214                udotv = dot(u_0,v);
215                tdotv = dot(t_0,v);
216
217                if udotv > tdotv
218                    newedges = [kminus; k];
219                else
220                    newedges = [k; kminus];
221                end
222
223            end
224
225            % Introducción en T y actualización de rho y Tinv
226            if isempty(T)
227                T = newedges;
228                rho = rho_k0;
229                Tinv(T(1:length(newedges))) = 1:length(newedges);
230            else
231                [pos_newedges, rho, T] = BinarySearchAndInsertion(rho, rho_k0, T, ...
232                    newedges, V_Mercator, WPs_Mercator(i,:), WPs_Mercator(j(j0),:));
233                Tinv(T(pos_newedges)) = pos_newedges;
234                Tinv(T(pos_newedges(length(pos_newedges)) + 1:length(T))) = ...
235                    Tinv(T(pos_newedges(length(pos_newedges)) + 1:length(T))) + ...
236                    length(pos_newedges);
```

```
237                 end
238             end
239         end
240 end
241 end
```

Código D.5 Función VisibleVerticesOD(i,V).

D.3.4 Función VisibleVA(i,j_{j0},e_1)

```
1 function vis = VisibleVA(j, i, WPs_Mercator, e_1, E, Einv)
2
3 vis = 0;
4
5 % Comprobación de que el segmento ij no pase por el interior de un polígono del que
6 % i es vértice
7
8 % Cálculo del vector v
9 v = (WPs_Mercator(j,:) - WPs_Mercator(i,:))';
10
11 % Formación de la base no ortogonal de vectores
12 r = (WPs_Mercator(E(Einv(i, 1), 2),:) - WPs_Mercator(i,:))';
13 s = (WPs_Mercator(E(Einv(i, 2), 1),:) - WPs_Mercator(i,:))';
14
15 % Producto vectorial de r y s
16 q_z = r(1)*s(2) - r(2)*s(1);
17
18 % Resolución del sistema de ecuaciones
19 a = (v(1)*s(2) - v(2)*s(1))/q_z;
20 b = (r(1)*v(2) - r(2)*v(1))/q_z;
21
22 % Aplicación del criterio con una cierta tolerancia
23 if (sign(q_z) == -1 && ~(a>1e-12 && b>1e-12)) || (sign(q_z) == 1 && (a>=-1e-12 && ...
24     b>=-1e-12))
25
26     % Vectores iA y AB
27     iA = WPs_Mercator(E(e_1,1),:)' - WPs_Mercator(i,:)';
28     AB = WPs_Mercator(E(e_1,2),:)' - WPs_Mercator(E(e_1,1),:)';
29
30     % Cálculo de beta
31     beta = (-iA(1)*AB(2) + iA(2)*AB(1))/(-v(1)*AB(2) + v(2)*AB(1));
32
33     % Aplicación del criterio con una cierta tolerancia
34     if 1 <= beta + 1e-10
35         vis = 1;
36     end
37 end
38 end
```

Código D.6 Función VisibleVA(i,j_{j0},e_1).

D.3.5 Función VisibleOD(i,j_{j0},e_1)

```
1 function vis = VisibleOD(j, i, WPs_Mercator, e_1, E)
```

```
2
3 vis = 0;
4
5 % Cálculo del vector v
6 v = WPs_Mercator(j,:)' - WPs_Mercator(i,:)';
7
8 % Vectores iA y AB
9 iA = WPs_Mercator(E(e_1,1),:)' - WPs_Mercator(i,:)';
10 AB = WPs_Mercator(E(e_1,2),:)' - WPs_Mercator(E(e_1,1),:)';
11
12 % Cálculo de beta
13 beta = (-iA(1)*AB(2) + iA(2)*AB(1))/(-v(1)*AB(2) + v(2)*AB(1));
14
15 % Aplicación del criterio con una cierta tolerancia
16 if 1 <= beta + 1e-10
17     vis = 1;
18 end
19 end
```

Código D.7 Función VISIBLEOD(i, j_{j0}, e_1).

Bibliografía

[1] WIKIPEDIA: *Radar meteorológico*. Disponible en: *https://es.wikipedia.org/wiki/Radar_meteorol%C3%B3gico#p-lang-btn*.

[2] WIKIPEDIA: *Zona de convergencia intertropical*. Disponible en: *https://es.wikipedia.org/wiki/Zona_de_convergencia_intertropical*.

[3] AIRGWAYS: *40º aniversario del accidente 242 Southern Airways*. Disponible en: *https://airgways.com/2017/04/04/40o-aniversario-del-accidente-vuelo-242-de-southern-airways/*.

[4] HOSTELSUR: *Avión desaparecido bajó a 1.500 metros para evitar los radares*. Disponible en: *https://www.hosteltur.com/lat/127340_avion-desaparecido-1500-metros-evitar-radares.html*.

[5] A. FRANCO, F. R. GAVILÁN, G. PACHECO Y R. VÁZQUEZ (2022): *Fundamentos de Navegación Aérea*. Escuela Técnica Superior de Ingeniería, Universidad de Sevilla.

[6] STACK OVERFLOW: *Sort points in clockwise order?*. Disponible en: *https://stackoverflow.com/questions/6989100/sort-points-in-clockwise-order*.

[7] C. VIVAS Y J. A. ACOSTA (2020): *Informática*. Escuela Técnica Superior de Ingeniería, Universidad de Sevilla.

[8] J. M. VIÑAS (2011): *Tormentas en vuelo*. Disponible en: *https://www.divulgameteo.es/uploads/Tormentas-en-vuelo.pdf*.

[9] B. GONZÁLEZ (2000): *Meteorología Aeronáutica*. 1ª Edición, España, Actividades Varias Aeronáuticas.

[10] GITHUB: *Visibility graphs & the naïve algorithm*. Disponible en: *https://taipanrex.github.io/2016/09/17/Distance-Tables-Part-1-Defining-the-Problem.html*.

[11] J. KITZINGER (2003): *The Visibility Graph Among Polygonal Obstacles: a Comparison of Algorithms*. Disponible en: *https://www.cs.unm.edu/~moore/tr/03-05/Kitzingerthesis.pdf*.

[12] WIKIPEDIA: *Mercator projection*. Disponible en: *https://en.wikipedia.org/wiki/Mercator_projection*.

[13] M. DE BERG, O. CHEONG, M. VAN KREVELD AND M. OVERMARS (2000): *Computational Geometry: Algorithms and Applications*. 3ª Edición, Alemania, Springer.

[14] GitHub: *Lee's visibility graph algorithm*. Disponible en: *https://taipanrex.github.io/2016/10/19/Distance-Tables-Part-2-Lees-Visibility-Graph-Algorithm.html*.

[15] Sciences at Smith College: *An $O(n^2 \log n)$ Algorithm for Computing Visibility Graphs*. Disponible en: *https://www.science.smith.edu/~istreinu/Teaching/Courses/274/Spring98/Projects/Philip/fp/algVisibility.htm*.

[16] D. Coleman (2012): *Lee's $O(n^2 \log n)$ Visibility Graph Algorithm Implementation and Analysis*. Disponible en: *https://dav.ee/papers/Visibility_Graph_Algorithm.pdf*.

[17] OurAirports. Disponible en: *https://ourairports.com/*.

[18] Wikipedia: *Algoritmo de Ramer–Douglas–Peucker*. Disponible en: *https://es.wikipedia.org/wiki/Algoritmo_de_Ramer%E2%80%93Douglas%E2%80%93Peucker*.

[19] GeeksforGeeks: *Heap Sort – Data Structures and Algorithms Tutorials*. Disponible en: *https://www.geeksforgeeks.org/heap-sort/*.

[20] A. Íñiguez (2020): *Desarrollo de una herramienta de planificación de trayectorias de evitación de regiones de riesgo meteorológico*. Trabajo Fin de Grado, Escuela Técnica Superior de Ingeniería, Universidad de Sevilla.

[21] S. K. Ghosh and D. M. Mount (1991): *An output-sensitive algorithm for computing visibility graphs*. Disponible en: *https://www.cs.umd.edu/~mount/Papers/sicomp91-visib.pdf*.