Trabajo de Fin de Máster
Máster en Ingeniería Aeronáutica

Adaptación e implementación en Matlab del método de Ghosh y Mount para el cálculo del grafo de visibilidad

Autor: Guillermo Vallejo Soto

Tutor: Antonio Franco Espín

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

Sevilla, 2024

Trabajo de Fin de Máster
Máster en Ingeniería Aeronáutica

Adaptación e implementación en Matlab del método de Ghosh y Mount para el cálculo del grafo de visibilidad

Autor:
Guillermo Vallejo Soto

Tutor:
Antonio Franco Espín
Profesor Titular

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

Sevilla, 2024

Trabajo de Fin de Máster: Adaptación e implementación en Matlab del método de Ghosh y Mount para el cálculo del grafo de visibilidad

Autor: Guillermo Vallejo Soto
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

La realización de este trabajo ha supuesto todo un reto a nivel personal. Parafraseando al ilustre filósofo Ortega y Gasset: *Yo soy yo y mi circunstancia*, y en esta ocasión salvarla ha implicado la entrada al mercado laboral, emigrar al extranjero y aprender un nuevo idioma desde sus bases. Todo esto no hubiera sido posible sin el apoyo de mi pareja, mi familia, mis amigos y la confianza mostrada por mi tutor, Antonio Franco.

Muchas gracias, sin vosotros, esto no hubiera sido posible.

Dankbarkeit ist das Gedächtnis des Herzens.

Guillermo Vallejo Soto
Máster en Ingeniería Aeronáutica

Hannover, 2024

Resumen

El grafo de visibilidad de un dominio es un grafo no direccionado cuyos vertices son los vértices del dominio y cuyas aristas son las líneas de visión directa entre los vértices no interrumpidas por los obstáculos del dominio, es por ello que su cálculo es crítico a la hora de resolver problemas de trazado óptimo de trayectorias. A lo largo del siguiente documento se hará una presentación acerca de las motivaciones que pudieran llevar a querer resolver este problema y una concisa revisión acerca del estado del arte. Se desarrollará una implementación en Matlab™ realizada en base al algoritmo propuesto por Ghosh & Mount y se establecerán una serie de metodologías para ponerlo a prueba y comprobar su eficacia y efectividad.

Abstract

The visibility graph associated with a domain is an undirected graph whose vertices are those of the domain, and whose edges represent direct lines of sight between vertices, unblocked by obstacles. This calculation is essential for solving optimal trajectory planning problems. Throughout this document, the motivations for addressing this problem will be introduced, along with a concise review of the state of the art. An implementation based on the Ghosh & Mount method in MATLAB™ will be presented, followed by a series of methodologies to test its efficacy and effectiveness.

Índice Abreviado

Índice

Notación

V	Matriz de coordenadas
WPs	Vector de puntos de paso
N	Número de puntos de paso diferentes
P	Dominio P
e_k	Conjunto de aristas perteneciente al dominio
S	Recta horizontal que pasa por un punto
T	Conjunto de aristas intersecados por S
x,y	Coordenadas en espacio cartesiano
xv	Base izquierda de la familia de embudos (x,y) dividida por ii
vy	Base derecha de la familia de embudos (x,y) dividida por ii
EVG	Diagrama de Visibilidad extendido
$\overline{ij}$	Segmento de recta desde el punto i al punto j
$\vec{ij}$	Vector director de i a j
$\vec{v}$	Vector $\vec{v}$
$\|\mathbf{v}\|$	Norma del vector **v**
$\vec{v}\cdot\vec{w}$	Producto escalar de los vectores **v** y **w**
$\vec{v}\times\vec{w}$	Producto vectorial de los vectores **v** y **w**
$\vec{q}_z$	Vector unitario en la dirección perpendicular al plano z
i,j,k	Índices comunes
NaN	Not a Number
$\in$	Que se encuentra en
$\|\mathbf{A}\|$	Determinante de la matriz cuadrada **A**
$\det(\mathbf{A})$	Determinante de la matriz (cuadrada) **A**
log	Logaritmo de base natural
e	Número e
sen	Función seno
tg	Función tangente
$\leqslant$	Menor o igual
$\geqslant$	Mayor o igual
$\eqsim$	Aproximadamente igual
™	Trade Mark

1 Motivación, estado del arte y objetivos del trabajo

Existen varias y diversas razones por las cuales resulta que en la práctica, la computación del diagrama de visibilidad es una necesidad ineludible. Dado que el contexto del redactor es aeroespacial, se partirá de esta base.

1.1 Importancia del trazado de ruta en el mundo aeronáutico

Las operaciones aeronáuticas están fuertemente reguladas con el propósito de mantener los niveles de seguridad operacional y física a la vanguardia del mundo ingenieril, estableciendo estándares de calidad que superan las de otros sectores estratégicos.

La preparación y ejecución de las rutas de los vuelos comerciales es una de estas actividades críticas, dado que un buen trazado proporciona beneficios a varios niveles:

- Ecológicos: Con los objetivos de la Agenda 2030 en mente, enfatizando en los objetivos número 13 *Acción por el clima* y el número 9 *Industria, Innovación y Crecimiento económico*, la optimización de trayectorias de ruta es una alternativa factible para reducir el consumo de combustible combinada con otras estrategias. Reduciendo las emisiones de gases de efecto invernadero y haciendo la industria aeronáutica más sostenible.
- Sociales: Existen zonas o bien cerradas al tráfico aéreo por razones de seguridad nacional, o bien por zonas de limitación de ruido dependientes del horario. Una ruta óptima deberá añadir estos factores a la ecuación para minimizar el impacto de las actividades aeronáuticas.
- Económicos: Las rutas óptimas no sólo reducen el consumo de combustible, si no que, además, tienen un doble efecto de ahorro para las compañías que ofrecen estos servicios al gran público. El coste operacional de las aerolíneas es fuertemente dependiente de los gastos derivados del combustible empleado, [1]. Es por ello que, pequeñas reducciones en estos gastos pueden tener un gran efecto en la estrategia financiera. Por otro lado, reducir las emisiones de CO_2 dispone en la Unión Europea a pagar cánones menores, disminuyendo por partida doble los costos operativos. Una buena planificación de ruta aplicada a todos los agentes de la industria puede evitar retrasos y los costes de los mismos para las aerolíneas.
- Seguridad: Un trazado de rutas óptimo tiene impacto en la seguridad operacional, evitando accidentes que pudieran producirse debido a eventos climatológicos de alto impacto en las operaciones aeronáuticas, como las borrascas o nubes que pudieran reducir la visibilidad para los pilotos. Además conforma una herramienta de apoyo excelente para control de tráfico

aéreo, *ATM*, pudiendo aumentar la capacidad de los sectores sin perjudicar los estándares de seguridad mencionados anteriormente.

Esto es sólo el enfoque referente a la aviación comercial, pero a pequeña escala, por ejemplo, la industria de los vehículos aéreos no tripulados, el diseño y optimización de rutas acorde a la misión propuesta, permite una operación controlada y libre de riesgos de colisión. Estas aplicaciones serían de alto interés en la entrega urgente y autónoma de mercancía, como en el caso del hospital de la Paz en Madrid, que participa de forma activa en el desarrollo de un proyecto que permita estandarizar el uso de *UAV* con objeto de optimizar la entrega de material sanitario de alto interés, [2].

1.2 Otras aplicaciones de interés

Estas ventajas no se limitan exclusivamente al entorno aeroespacial, por lo que se hará a lo largo de esta sección un pequeño desglose acerca del impacto del cálculo de rutas, y por ende del grafo de visibilidad, en otras industrias.

1.2.1 Robótica

A día de hoy, los robots autónomos no son ciencia ficción, y aunque distan de la visión popular que se tenía de los mismos, provocada en gran parte por la industria del entretenimiento, son muchas las empresas que los usan para optimizar sus procesos. Entre estos procesos se encuentran, por ejemplo, la gestión de almacenes y paquetería, donde el trazado de rutas óptimas en espacios reducidos y con presencia de otros agentes dinámicos, es imprescindible. Amazon y Tesla se presentan como referentes en la industria, tanto en sacarle ventaja al uso de estas tecnologías como en accidentes laborales en trabajadores de la empresa a causa de estándares de seguridad deficientes, [3, 4].

1.2.2 Geoinformática y sistemas de información geográfica

Un trazado de rutas óptimo puede llegar a tener impacto a la hora de representar rutas y mapear zonas accesibles o intransitables. Esto puede ser altamente beneficioso a la hora de evaluar zonas seguras en zonas propensas a sufrir deslizamientos, avalanchas o erupciones volcánicas, y trazado rutas óptimas en situaciones de emergencia y rescate.

1.2.3 Ingeniería en redes y telecomunicaciones

Los grafos de visibilidad pueden ser empleados para determinar las posiciones óptimas para ubicar antenas de radio, telefonía y televisión, maximizando la cobertura mediante la elusión de obstáculos naturales y artificiales. Garantizando la cobertura de estos servicios en áreas pobladas, o disminuyendo los costes operacionales, facilitando que estos mismos lleguen a zonas más apartadas, como por ejemplo, la España vaciada.

1.2.4 Arquitectura y urbanismo

De forma similar a como se hacía en la aplicación propuesta de gestión de almacenes, resulta de interés usar un enfoque similar para poder trazar rutas de accesibilidad y evacuación eficientes. Pudiéndose llegar a optimizar estas rutas de cara a la accesibilidad. Planteando, entre otras alternativas, rutas posibles para personas con movilidad reducida y mejorando la infraestructura urbana.

Por estos y otros tantos motivos, el trazado y optimización de rutas, en especial el cálculo de grafos de visibilidad, es un problema cuyo perfeccionamiento es de alto interés estratégico de cara a los retos del mañana.

1.3 Estado del arte

En esta sección se repasarán diferentes métodos de cálculo de grafos de visibilidad empleados en la actualidad, junto a la complejidad asociada a los mismos.

1.3.1 Método Ingenuo

A la hora de resolver un problema, suele ser interesante comprobar cual es la resolución posible que podría alcanzarse mediante el empleo de la *fuerza bruta*, para poder tener un caso de referencia y comprobar como de bueno es el algoritmo que se propone frente a él. Esta serie de métodos son comúnmente denominados como *Ingenuos* o *Naive*.

Para el caso de la determinación del grafo de visibilidad el método Ingenuo propone recorrer cada punto de paso, denominados como *Waypoints*, **WPs**, comprobando si la visibilidad con el resto de puntos de paso,**WPs** se ve interrumpida por alguna arista obstáculo e_P del dominio **P**, de forma adicional, es necesario comprobar si esta linea de visibilidad no cruza dentro de un obstáculo.

Este problema puede simplificarse ligeramente si se consideran las propiedades simétricas del problema, si $\overline{ij}$ es visible, $\overline{ji}$ también lo será, y viceversa. Esto reduce el número de comprobaciones necesarias y acelera el proceso de cálculo.

Se adjunta el pseudocódigo del programa que ejecuta lo anteriormente expresado, este método fue implementado satisfactoriamente en Matlab en el Trabajo de Fin de Grado de Narciso Valverde [5].

Pseudocódigo 2.1 Cálculo del grafo de Visibilidad mediante algoritmo Ingenuo

Grafo de visibilidad = función_algoritmo_ingenuo(Matriz de coordenadas **V**)
1: **para** cada punto de paso i∈ **WPs**, **hacer**
2: **para** cada punto de paso j∈ ($i+1$,**nWPs**), **hacer**
3: **para** cada arista obstáculo k∈ e_P, **hacer**
4: **si** no hay intersección entre la arista $\overline{ij}$ y e_k **y además** $\overline{ij}$ no pasa dentro de un obstáculo contenido en **P**, **entonces**
5: Añadir al Grafo de visibilidad $\overline{ij}$
6: **devolver** Grafo de visibilidad

La comprobación de si la potencial línea de visibilidad pasa por dentro del polígono al que pertenecen puede resolverse si se tiene información de antemano acerca de los puntos del polígono ordenados. Esto forma parte de la premisa, por lo que se puede utilizar estos datos proporcionados junto con los productos vectoriales para minimizar el número de operaciones. El fundamento se basa en el siguiente precepto: todo punto en un polígono plano convencional está conectado a otros dos puntos, uno que lo precede y otro que lo sucede. Este arco que forman puede ser cóncavo o convexo, pero cualquier línea de visión que parta de ese punto saldrá hacia el exterior del polígono o hacia el interior del polígono y eso puede comprobarse usándo productos vectoriales. Se muestra un ejemplo visual en la figura 1.1.

Determinar si el arco formado es cóncavo o convexo para un punto y su predecesor/sucesor puede hacerse realizando el producto vectorial entre ambos, teniendo en mente que están proporcionados los puntos del polígono respecto a su centroide y se disponen en el sentido contrario a las agujas del reloj, manteniendo la nomenclatura de la figura 1.1:

$$a)\ \vec{v}_1 \times \vec{v}_2 \cdot \vec{q}_z > 0, \tag{1.1}$$

$$b)\ \vec{v}_1 \times \vec{v}_2 \cdot \vec{q}_z < 0, \tag{1.2}$$

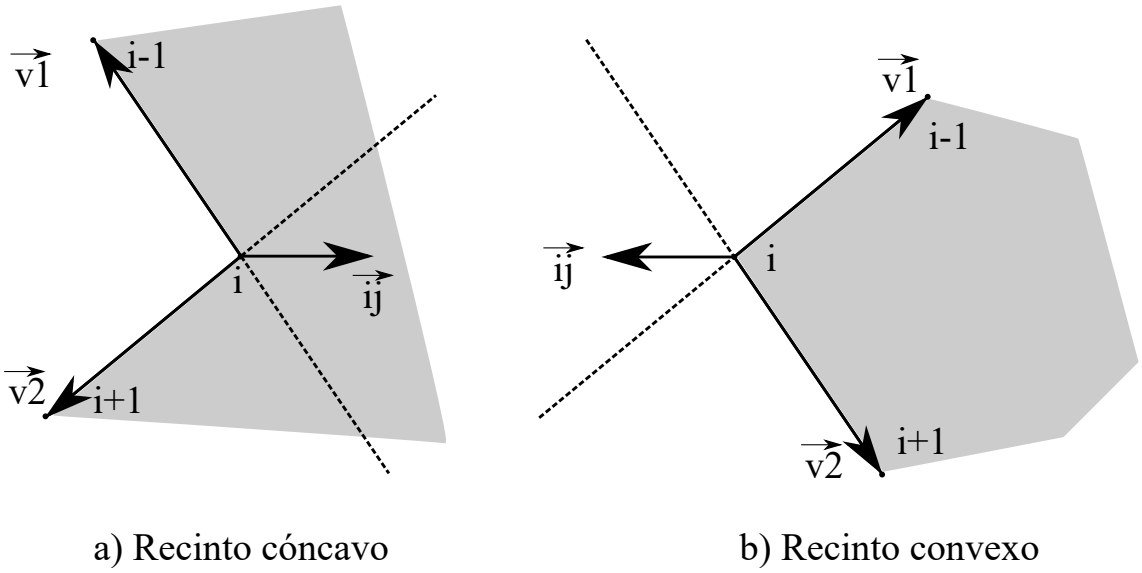


Figura 1.1 Uso de productos vectoriales para determinar si una línea cruza por dentro de un polígono.

donde el caso $a)$ representa concavidad localizada y $b)$ convexidad localizada. Una vez conocedores de esto, se puede distinguir a partir del uso de inecuaciones si pasa por dentro o no del polígono dependiendo del caso:

1. Caso concavidad localizada, $\vec{ij}$ fuera del polígono:

$$\vec{ij} \times \vec{v}_1 \cdot \vec{q}_z < 0,\ \vec{ij} \times \vec{v}_2 \cdot \vec{q}_z > 0. \tag{1.3}$$

2. Caso convexidad localizada, $\vec{ij}$ dentro del polígono:

$$\vec{ij} \times \vec{v}_1 \cdot \vec{q}_z > 0,\ \vec{ij} \times \vec{v}_2 \cdot \vec{q}_z < 0. \tag{1.4}$$

Esta propiedad ha sido implementada en la algoritmia del método de Ghosh & Mount, [6], dado que permite estandarizar cálculos en recintos cóncavos con bajo coste computacional. Es uno de los códigos empleados en la sección 2.2.3.

La complejidad algorítmica de este método propuesto en notación $Big(O)$ es:

$$O(N) \approx N^3, \tag{1.5}$$

debido a que es necesario anidar bucles tres veces recorriendo el conjunto de puntos completo, siendo N el número de diferentes puntos de paso.

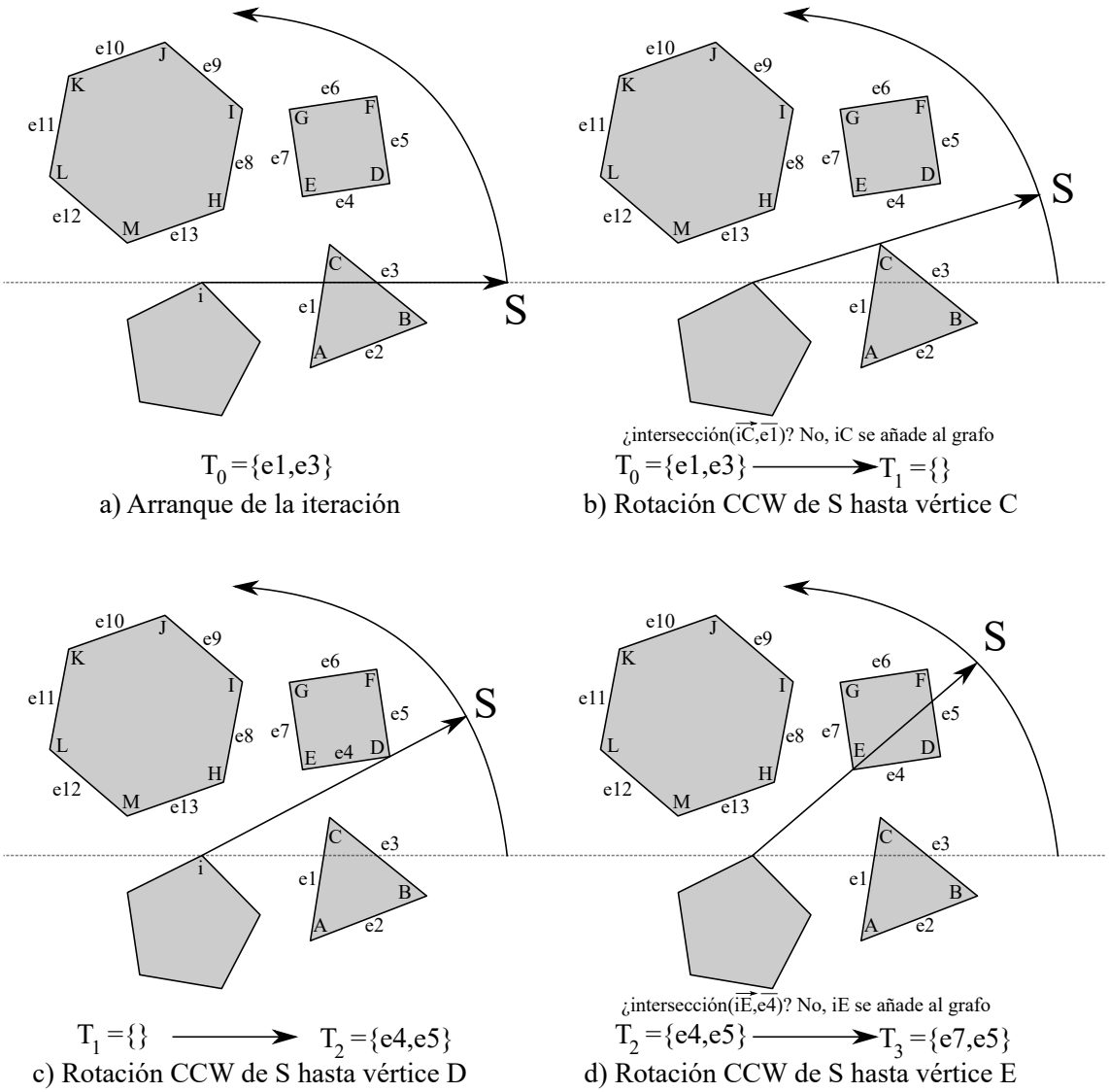
Partiendo de esta base, cualquier algoritmo que mejore estas condiciones manteniendo la validez de los resultados representará una ventaja desde un punto de vista computacional.

1.3.2 Método de Lee

El método de Lee es otra metodología que permite obtener el grafo de visibilidad empleando una estrategia que solo comprueba las visibilidades con la arista más cercana al punto que se está añadiendo al grafo. Toda la explicación desarrollada a continuación está basada en el trabajo de investigación realizado por Narciso Valverde [5], y sus resultados han sido utilizados como referencia a la hora de trazar comparaciones con el método de Ghosh & Mount.

Para realizar el cálculo del grafo se recorren todos los puntos de paso y para cada uno de ellos se procesa el semiplano superior a la horizontal que pasa por el punto a añadir al grafo. Es posible

realizar esto debido a que el problema es simétrico, es decir, los puntos que están en contacto visual de norte a sur, son los mismos que están en contacto visual de sur a norte. Por lo que procesando los semiplanos superiores para todos los puntos de paso queda cubierto completamente el dominio.



a) Arranque de la iteración
b) Rotación CCW de S hasta vértice C
c) Rotación CCW de S hasta vértice D
d) Rotación CCW de S hasta vértice E

Figura 1.2 Ejecución método de Lee para un punto de paso.

Todo este proceso está reflejado en la figura 1.2, y se irán mencionando los diferentes puntos y etapas referenciando a este soporto visual. El siguiente procedimiento se ejecuta para cada punto de paso. Primero se traza una línea horizontal a través del punto de estudio *i*, se comprueban los cortes de esta recta *S* con el resto de polígonos del dominio, cada una de las aristas con las que corta se almacenan en orden de corte en el vector *T*, paso *a*) de la figura. Una vez inicializado este vector, se visitan en el sentido contrario a las agujas del reloj los puntos de paso ubicados en el semiplano superior. Cada vez que se visita un punto de paso se comprueba si existe un corte entre la línea de visión y el punto que se visita. En caso de que no existan cortes o no haya aristas almacenadas en *T* el segmento se añade al grafo de visibilidad. Después de esto, se actualiza el vector *T*, siempre que se visite un punto puede darse cualquiera de los tres siguientes casos:

1. El punto que se visita pertenece a un nuevo polígono, por lo que hay que añadir dos nuevas aristas a *T*. Esto puede observarse en la figura en el caso *c*.

2. El punto que se visita pertenece a un polígono pero no es el último nodo del mismo, por lo que habría que añadir una arista y retirar otra. Esto puede observarse en la figura en el caso d).
3. El punto que se visita es el último a visitar del polígono, por lo que habría que retirar las dos aristas asociadas al mismo. Esto puede observarse en la figura en el caso b).

Este método presenta una complejidad algorítmica más eficiente que el presentado en el método Ingenuo, siendo esta: $O(N^2 \log N)$.

1.3.3 Otros métodos

Existen técnicas de cálculo rápido de grafos de visibilidad que, basados en técnicas heurísticas, dan como resultado grafos de visibilidad incompletos. El estudio de estos grafos de visibilidad incompletos es también de interés dado que para la mayoría de las rutas, las trayectorias más eficientes son aquellas que recorren la frontera de los obstáculos o las tangentes internas y externas entre los diferentes polígonos.

Por otro lado investigadores del campo de la geometría computacional, como Joseph O'Rourke, han planteado una alternativa al grafo de visibilidad, denominada *vertex-edge* grafo de visibilidad, en el cual no sólo se registra la información de cómo son visibles unos nodos frente a otros, si no que extiende el concepto a la visibilidad entre vértice y aristas, [7].

Existen también estrategias incrementales, para las que se construye de forma aditiva el grafo en función de las inmediaciones de los puntos a visitar, que en aplicaciones en tiempo real pueden llegar a ser altamente eficientes, [8].

Por último existen técnicas de *Ray-Casting*, que son ampliamente utilizadas en simulaciones de iluminación dinámica y sombras, en campos como el renderizado de modelos $3D$ y en especial en el mundo de los videojuegos y el cine, con la *Computer Generated Imagery*, que es la base de los efectos especiales digitales.

1.3.4 Método de Ghosh & Mount: Objetivos y alcance

Con estas ideas en mente, el enfoque de este proyecto se hará en un modelo concreto, el método de Ghosh & Mount, [6], que es un algoritmo que usando las propiedades de unas entidades matemáticas llamadas embudos, busca resolver el problema del cálculo del grafo de visibilidad.

Los siguientes capítulos tendrán como propósito desarrollar la investigación y el trabajo de interpretación e implementación realizado por el redactor en el contexto de este método. Los objetivos del trabajo son los siguientes:

1. Implementar una versión funcional del modelo en el lenguaje de programación Matlab™, buscando mantener la complejidad algorítmica inicialmente propuesta tanto como sea posible en base a lo propuesto en el documento original.
2. Comparar el método con los otros propuestos en este apartado, en específico el de Lee y el Ingenuo. Considerando los siguientes parámetros: Comparativa de grafos, tiempos de ejecución y escalado respecto al número de puntos de diferentes dominios.
3. Prueba de diferentes dominios, tantos cóncavos como convexos, con el objetivo de comprobar la efectividad del método.
4. Realizar un análisis costo-beneficio y extraer conclusiones acerca del método originalmente propuesto.

El alcance del proyecto, será entonces, llegar a una implementación plausible y funcional, y reflejar como se ha realizado en este documento.

2 Método de Ghosh & Mount

El código empleado y sus bases teóricas se verán desarrolladas a lo largo del siguiente capítulo. En primer lugar se explicará el método en términos simples para cimentar una base de entendimiento. A continuación se intercalarán fragmentos de código, la casuística cubierta y su correspondencia con las bases planteadas por Ghosh & Mount en el artículo *An output-sensitive algorithm for computing visibility graphs*, [6], al que se le llamará indistintamente como algoritmo *SPLIT*. La estructura de este capítulo se basará en el código de la implementación, conforme se vaya profundizando en el mismo, se mencionará su correspondencia con la base teórica presente en el ya mencionado artículo.

2.1 Acercamiento teórico

Para entender las bases del algoritmo, se adjunta la imagen 2.1 como soporte visual para el lector. A diferencia del método de Lee, el de Ghosh & Mount plantea un recinto poligonal cerrado con zonas prohibidas, *agujeros*, que deben cumplir la condición de ser pares interiores disjuntos, es decir, estos polígonos no pueden intersecar entre sí. En la implementación realizada, el recinto exterior se calcula tomando como referente los obstáculos propuestos, y disponiendo un trapezoide. Los detalles serán desarrollados más adelante. Para simplificar el proceso de entendimiento del problema, se plantea al lector el ejercicio mental de imaginar un laberinto en el cual polígonos macizos conforman las paredes del mismo. Una vez dentro del laberinto, para cada vértice de estos polígonos, se observará en dirección oeste y se intentarán identificar los vértices ya visitados y si estos son visibles o no desde la posición tomada. Para ello se seguirán diferentes criterios y se empleará un artefacto matemático llamado embudo.

El cono de visión queda geométricamente determinado por el punto en el que se sitúa el espectador, actuando como vértice, y como base otros dos puntos del dominio a través de los cuales se mira. Con la información de qué puntos son visibles para esa línea de visión, se sabría cómo la geometría de esos puntos obstaculiza la visión al espectador. En la figura 2.1, puede comprobarse que existen vértices que tienen línea de visión directa con el espectador, esos se añadirían al grafo de visibilidad. Pero por otro lado, existen puntos y aristas que no son deducibles para este a priori, marcados con línea discontinua. Nótese que en la parte inferior de la figura la línea de visión sería el margen inferior, y el espectador, tal y como se plantea la perspectiva cenital, se encontraría en el infinito en esta proyección.

El funcionamiento general del código es el que se plantea a continuación, cada una de las diferentes secciones se irán, paso a paso, detallando, justificando los porqués de las decisiones tomadas en la implementación.

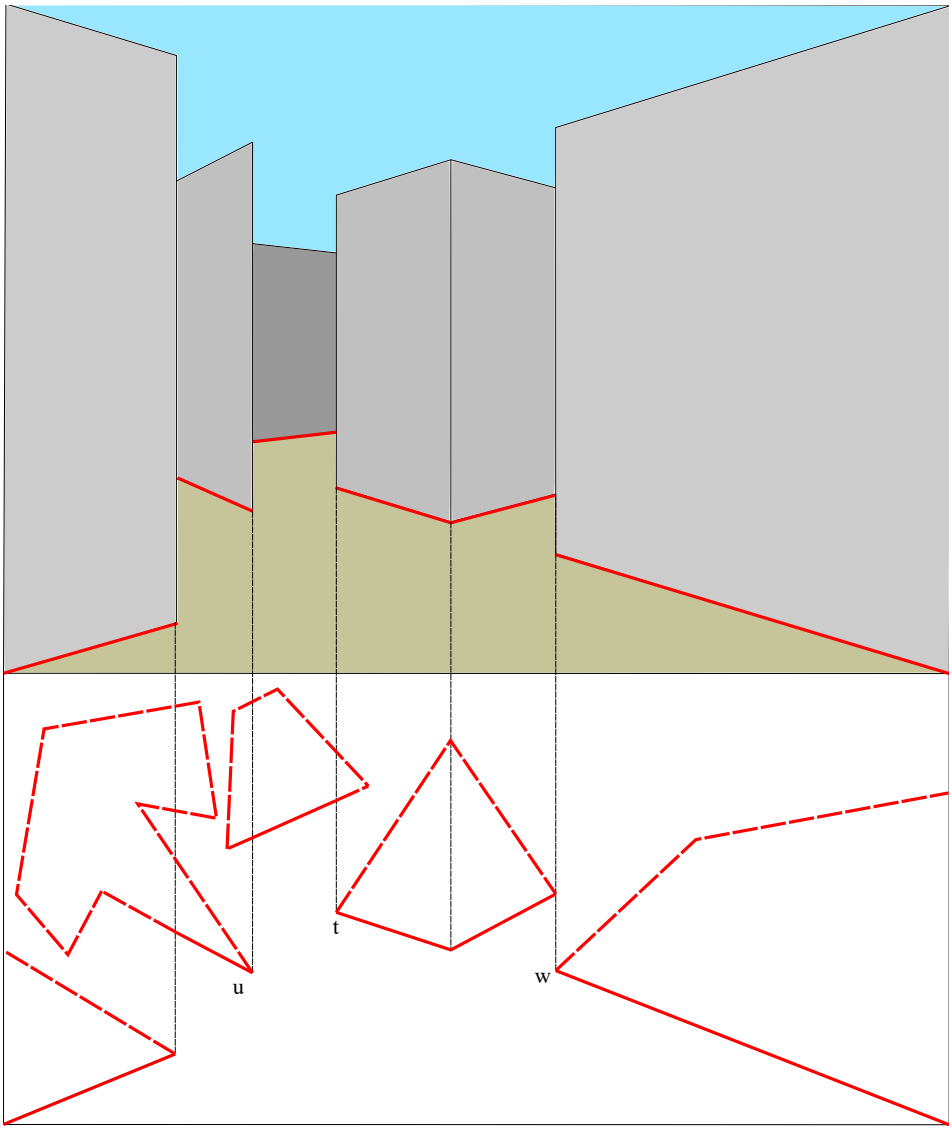


Figura 2.1 Ejemplo práctico.

Pseudocódigo 3.1 Cálculo del grafo de Visibilidad mediante algoritmo *SPLIT*

[t, Grafo de visibilidad] = Main_v13(Matriz de coordenadas **V**)
1: Ejecución del algoritmo de preprocesado de los datos
2: **para** cada punto de paso i∈ **WPs**, **hacer**
3: Adición mediante triangulación de i al dominio P_{i-1}
4: Ejecución de **SPLIT** en los puntos de contacto al dominio
5: Adición de los nodos computados al diagrama de visibilidad extendido
5: **devolver** Tiempo de computación, Grafo de visibilidad

¿De qué manera puede el algoritmo identificar las esquinas del laberinto? ¿Y los diferentes pasillos? En las siguientes secciones se desarrollará el mecanismo matemático que hace esto posible.

2.2 Arranque del programa

Los datos de entrada de un programa que busque calcular un grafo de visibilidad serán, por supuesto, geométricos. El punto de partida es una estructura matricial dada en Matlab ™con la disposición que se encuentra en la expresión 2.1:

$$\mathbf{V} = \begin{bmatrix} x_{a,1} & y_{a,1} \\ x_{a,2} & y_{a,2} \\ x_{a,3} & y_{a,3} \\ x_{a,4} & y_{a,4} \\ x_{a,1} & y_{a,1} \\ \text{NaN} & \text{NaN} \\ x_{b,1} & y_{b,1} \\ x_{b,2} & y_{b,2} \\ x_{b,3} & y_{b,3} \\ x_{b,1} & y_{b1} \\ \text{NaN} & \text{NaN} \\ \vdots & \vdots \\ x_{n,7} & y_{n,7} \\ x_{n,1} & y_{n,1} \\ \text{NaN} & \text{NaN} \end{bmatrix} \tag{2.1}$$

Los polígonos son dados como en el método de Lee a través de sus coordenadas, separados por NaN, repitiendo al final de la secuencia el primer valor del polígono para así cerrarlo. Estos vértices no están dispuestos al azar, están almacenados tomando como referencia el centroide y partiendo de un vértice concreto, estando los nodos en orden antihorario, añadiendo los vértices consecutivos entre sí, tal y como se ve en la figura 2.2. El primer polígono representa el recinto exterior, a diferencia del método de Lee, [5], que no precisaba de un recinto.

El número de puntos de paso será registrado como N, y cada uno de ellos se identificará con su posición en este vector, es decir, el nodo k es el que tiene coordenadas $WPs(k,:)$. Estos valores de V deberán ser procesados para ser convertidos en *Waypoints*, o puntos de paso. Para ello se hará uso del comando *rmmissing*, que removerá los *NaN* y de *unique*, que removerá los valores duplicados.

```
12 tic
13 close all
14
15 if ~isnan(V(end))
16     V=[V;NaN(1,2)];
17 end
18
19 EVG.V=V;
20
21 WPs=rmmissing(unique(V,'rows','stable'));
22 WPs = sortrows(WPs, 1); %Ordenar según posición en X
23
24 N=length(WPs);
25 EVG.N=N;
```

Código 2.1 Arranque del programa: Identificación de puntos de paso.

Para que el algoritmo de triangulación funcione, estos puntos de paso necesitan cumplir dos condiciones adicionales para poder ser procesados:

- **Ninguna combinación de tres puntos puede resultar en estos siendo colineares**: Esta condición se comprobará en la sección 2.2.4).

- **No existen dos puntos que compartan coordenada x**: Esta condición se comprobará justo después de la definición de los puntos de paso, dado que tiene fuerte injerencia en el preprocesado.

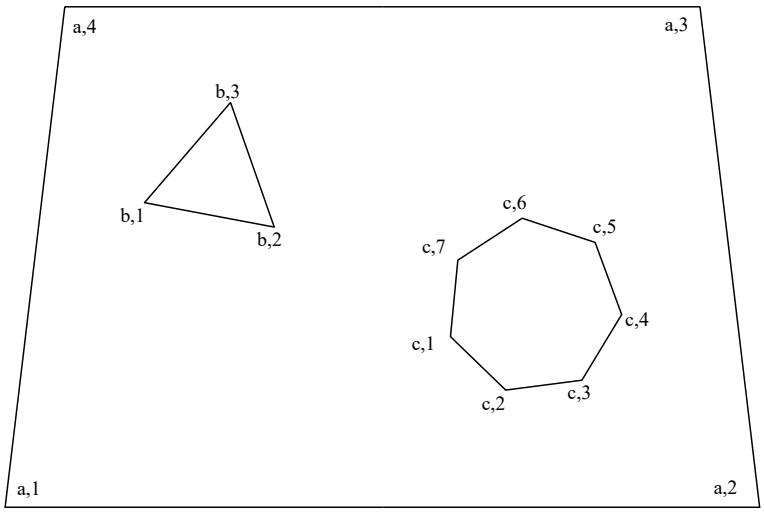


Figura 2.2 Ejemplo de dominio con poligonos.

La corrección de los equivalores de coordenada x se corregirán con la siguiente metodología recursiva: primero se identifican usando el comando *diff* (que calcula un vector restando la componente WP_{k-1} a WP_k para cada valor $k \in \{2,3,4,\ldots,N\}$). Se identifican los valores iguales a 0, que son necesariamente dos puntos que comparten coordenada x. Estos valores identificados se meten en un bucle que no cesará hasta que no quede ningún valor repetido. La clave para modificar los puntos consiste en un doble sistema de verificación, en el que se desplazan los puntos repetidos, menos el primero, un pequeño porcentaje de la distancia al siguiente punto legítimo. Nótese que al contener V información acerca de los diferentes nodos es necesario actualizarlo de manera acorde a como se hace con el vector *Waypoints*. Es por ello que para distinguir los puntos, dado que sus coordenadas en x son iguales, se usará la coordenada y. Si dos puntos diferentes comparten la misma coordenada son, de hecho, el mismo punto.

Es necesario destacar que el recinto es trapezoidal precisamente para evitar que el polígono externo genere valores iguales de coordenada x. Si se parte de un dominio que no plantea de base un recinto externo, como en el caso de la implementación del método de Lee propuesta en la sección 1.3.2, puede ubicarse libremente, por lo que se ha decidido establecer el siguiente convenio.

- Los nodos 1 y 2 conforman la esquina inferior y superior izquierda, respectivamente.
- Los nodos $N-1$ y N conforman la esquina inferior y superior derecha, respectivamente.

A la hora de plantear los recintos externos, se calcularán y plantearán las coordenadas del trapecio o trapezoide, de acorde a la implementación, ambas formas son válidas siempre y cuando se sigan las directrices de posición anteriormente propuestas.

Existen dos casos diferenciados en los que un punto pueda darse repetido en los datos: que sea el inicio y el cierre de un mismo polígono, caso ya contemplado; o que sean vértices de dos polígonos diferentes, lo cual rompe la regla de agujeros internos disjuntos por pares, por lo que habría que desechar la matriz de coordenadas.

La regla para identificar el punto a partir de su coordenada y es en esencia la misma para identificar valores repetidos. Se usa una vez más el comando *diff*, para localizar valores repetidos en el vector V, todo esto para cada valor registrado como repetido en el vector de puntos de paso, una vez se identifica el valor repetido, se desplaza un porcentaje de la distancia al siguiente punto de paso. Se plantea así para que no se dé el caso de que se modifique el inicio de un polígono pero su cierre no.

```
28 %% Comprobación valores repetidos
29
30 %Se comprueba si hay valores repetidos
31 %diff calcula la diferencia entre WPs(i-1) y WPs(i)
32 val_rep= diff(WPs(:,1))==0;%Siempre y cuando dos valores en x sean iguales, una componente del
       vector será un true
33
34 indrepes=find(val_rep==1);
35 while ~isempty(indrepes)
36
37     %Si hay algún true, se mandará un warning
38     if any(val_rep)
39         warning('Hay valores repetidos en X para WPs, se procede a corregir.');
40
41         for jj=1:length(indrepes)
42             %Corregimos WPs y Vs
43             indrepe=indrepes(jj);
44             vx=V(:,1)-WPs(indrepe+1,1);
45             indcerovx=find(vx==0);
46
47             vy=V(:,2)-WPs(indrepe+1,2);
48             indcerovy=find(vy==0);
49
50             for kk=1:length(indcerovx)
51                 a=indcerovx(kk);
52                 for ll=1:length(indcerovy)
53                     b=indcerovy(ll);
54
55                     if a==b
56                         %No se usa un break para que todas las instancias sean
57                         %modificadas (en el caso de que sea inicio/final de polígono)
58
59                         %Se mueve un porcentaje del valor entre el siguiente WP
60                         %y el valor, para no afectar a la geometría
61                         V(a,1)=WPs(indrepe+1,1)+0.01*(WPs(indrepe+2,1)-WPs(indrepe+1,1));
62                     end
63                 end
64             end
65             WPs(indrepe+1,1)=WPs(indrepe+1,1)+0.01*(WPs(indrepe+2,1)-WPs(indrepe+1,1));
66         end
67     end
68     val_rep= diff(WPs(:,1))==0;
69     indrepes=find(val_rep==1);
70 end
```

Código 2.2 Eliminación de valores repetidos en el vector de coordenadas y puntos de paso.

Una vez el vector *indrepes* está vacío, todos los puntos de paso están ordenados en orden ascendente de coordenada x y no existen valores repetidos. Gracias a esto puede continuar el preprocesado.

2.2.1 Nodos consecutivos

Como ya se adelantó en la sección anterior el vector V contiene información del dominio y como se distribuyen los diferentes polígonos y que puntos lo conforman. Cada polígono tiene información acerca de que puntos lo componen, ordenados en sentido antihorario alrededor del centroide, y respetando la información de que vértice conecta con cada cual.

$Consec_{nodos}(i,j)$ vale 1 si son consecutivos el punto de paso i y el punto de paso j, independientemente de si es consecutivo en sentido antihorario u horario. $Consec_{nodos2}$ considera solo los nodos consecutivos en sentido antihorario. Esto será necesario más adelante durante la triangulación y el algoritmo de ordenación de embudos.

La metodología para identificarlos consiste en avanzar a través de los valores del vector V, registrando una combinación de (i,j) y su simétrica, en el caso de $Consec_{nodos2}$ sólo se registrará si

$ii+1$ sucede a ii.

```
%% Cálculo de las conexiones entre nodos de un polígono
% Esta matriz es importante ya que es la que regulará el cálculo de u vector,
% y es lo que evitará que se creen y consideren embudos que provocarían
% falsos positivos en el grafo de visibilidad.


%Consec_aristas registrará solo las aristas consecutivas
Consec_aristas=zeros(N,N);
for ii=2:length(V(:,1))
    if ~isnan(V(ii-1,1)) && ~isnan(V(ii,1))
        A=find(WPs(:,1)==V(ii-1,1));
        B=find(WPs(:,1)==V(ii,1));
        Consec_aristas(A,B)=1;
        Consec_aristas(B,A)=1;
    end
end
EVG.Consec_aristasrec=zeros(N,2);
for ii=1:N
    ind=find(Consec_aristas(ii,:)==1);
    EVG.Consec_aristasrec(ii,:)=ind;
end

Consec_aristas2=zeros(N,N);
for ii=2:length(V(:,1))
    if ~isnan(V(ii-1,1)) && ~isnan(V(ii,1))
        Consec_aristas2(WPs(:,1)==V(ii-1,1),WPs(:,1)==V(ii,1))=1;
    end

end

EVG.Consec_aristas2=Consec_aristas2;
EVG.Consec_aristas=Consec_aristas;
```

Código 2.3 Registro de consecución de los nodos en los polígonos.

Nótese que de existir valores repetidos en la coordenada x, pudiera darse que para un punto de paso dado existieran más de dos aristas confluyendo hacia un mismo vértice.

2.2.2 Matrices de polígonos

En esta sección se guardará información de rápido acceso acerca de cuantos polígonos hay y que vértices los componen. La información que se extraerá es la siguiente:

- **poligono2nodo**: Es una matriz de dimensiones $npol\,x\,N$, siendo npol el número de polígonos, que pudieran causar interferencias en la visión, es decir, sin considerar el polígono exterior. El polígono que conforma el recinto exterior se considera el polígono 0. Si se quisiera consultar que vértices tiene el polígono i, se accedería a la fila i de la matriz, y se recortarían los valores distintos de 0, que se usan de relleno para mantener consistencia en el tamaño de la matriz.
- **nodo2poligono**: Es una matriz $N\,x\,1$ que proporciona la información inversa a poligono2nodo. Es decir, si se tuviera un punto de paso a que se quisiera saber a que polígono pertenece sólo sería necesario acceder a la información presente en la fila a.
- **limitespoligono**: Es una matriz $npol\,x\,2$, que registra cual son los puntos de paso de menor y mayor coordenada x para un polígono dado. Esto es interesante de cara a la identificación de aristas de intersección durante la triangulación.

poligono2nodo

Primero se definen dos variables bandera: $flag$, que se mantiene bajada hasta que se encuentra el primer valor de NaN, que si está bien definida la matriz, será una vez pasadas las coordenadas

del polígono 0; por otro lado $flag2$ sirve para indicarle al programa que cambie de fila una vez identifica que se ha repetido el punto inicial del polígono.

Una vez definidas las banderas, se recorren todos los valores pertenecientes a la matriz completa de coordenadas V. Cuando $flag2$ se levanta, se guarda la información en *poligono2nodo* junto con una fila de ceros para cubrir todas las columnas.

```
105 %% Matriz polígonos
106 %Esta matriz será tal que npol x N, contendrá información
107 %acerca de que aristas contiene cada poligono (Ojo, el recinto exterior no cuenta como polígono.)
108 flag=0;
109 npol=1;
110 ii=1;
111 jj=1;
112
113 while ii<=length(V(:,1))
114     if flag==1
115         flag2=0;
116         while flag2==0
117
118             %Solo hay un valor de x en Waypoints que coincida
119             P=find(WPs(:,1)==V(ii,1),1);
120
121             if jj~=1
122                 %Si no es el primer vértice del poligono, miramos si está repetido
123                 if EVG.poligono2nodo(npol,1)==P
124                     flag2=1;%Asi se sale del bucle que corre para cada secuencia de vértices que
     identifica el polígono
125                     jj=1;
126                 else
127                     EVG.poligono2nodo(npol,jj)=P;
128                     jj=jj+1;
129                 end
130             else
131                 EVG.poligono2nodo(npol,1)=P;
132                 jj=2;
133             end
134             ii=ii+1;
135         end
136         EVG.poligono2nodo(npol,1:N)=[EVG.poligono2nodo(npol,:),zeros(1,N-length(EVG.poligono2nodo
     (npol,:)))];
137         npol=npol+1;
138     end
139
140     if flag==0 && isnan(V(ii,1))
141         flag=1;
142     end
143     ii=ii+1;%Vale tanto para avanzar en el caso del recinto externo como para salir del NaN en el
       que dejaría salir del bucle de identificación de vértices.
144 end
```

Código 2.4 Identificación poligono2nodo.

nodo2poligono

Usando la información de *poligono2nodo* y recorriendo sus filas se puede registrar en un vector columna la relación de pertenencia de un nodo a un polígono. Es decir realizando la consulta $nodo2poligono(i)$, se obtendría el número de polígono (sin contar el recinto externo) al que pertenecería. Esto es especialmente útil para identificar a partir de un nodo concreto, cuales son los otros nodos que pertenecen al mismo polígono. Esto se emplea para calcular posibles interferencias de visibilidad con las aristas. Se recuerda al lector que todos los nodos están identificados por orden de x creciente, tal y como están dispuestos en WPs.

```
146 EVG.nodo2poligono=zeros(N,1);
147 for ii=1:length(EVG.poligono2nodo(:,1))
148     for jj=1:length(EVG.poligono2nodo(1,:))
```

```
149         if EVG.poligono2nodo(ii,jj)==0
150             break
151         end
152         EVG.nodo2poligono(EVG.poligono2nodo(ii,jj))=ii;
153     end
154 end
```

Código 2.5 Identificación nodo2poligono.

limitespoligono

Sacando partido de las propiedades de $nodo2poligono$ se recorren todos sus valores, no es necesario realizar este registro para el polígono 0, en caso contrario, hay que comprobar para la fila asociada al polígono si hay un valor en la primera columna ya asociado, si no es así, se guarda, si ya había un valor en la columna izquierda, se guarda en la columna de la derecha.

Esto es así porque al estar ordenados en orden creciente de coordenada x en el vector WPs, el primer valor que aparezca siempre será el más occidental, mientras que el último el más oriental. Para evitar meter varias comprobaciones redundantes, se sobrescriben datos en la columna de la derecha, la de mayor valor de coordenada x, hasta alcanzar el último valor, que provocará que a partir de ahí, no vuelvan a sobreescribirse valores.

```
156 EVG.limitespoligono=zeros(length(EVG.poligono2nodo(:,1)),2);
157
158 for ii=1:length(EVG.nodo2poligono)
159
160     if EVG.nodo2poligono(ii)~=0
161         if EVG.limitespoligono(EVG.nodo2poligono(ii),1)==0
162             EVG.limitespoligono(EVG.nodo2poligono(ii),1)=ii;
163         else
164             EVG.limitespoligono(EVG.nodo2poligono(ii),2)=ii;
165         end
166     end
167 end
168 % EVG.limitespoligono
169 %Esta variable limitespoligono guarda el primer punto y el último, en orden de polígono, de cada
      polígono
```

Código 2.6 Identificación limitespoligono.

2.2.3 Caminos prohibidos

Para optimizar los tiempos de procesamiento, se excluirán los caminos internos que atraviesan los polígonos, tanto en su totalidad como de manera parcial (en el caso de polígonos no convexos), ya que por definición son invisibles al cruzar áreas consideradas prohibidas. En esta fase del preprocesado, se extraerán los valores de los nodos vinculados a cada polígono y se enviarán a la subrutina $calcula_fronteras$, el cual se explicará detalladamente en el Apéndice A.

Esbozando el funcionamiento del programa, se traza para cada par de vértices una linea auxiliar y se realizan las siguientes comprobaciones:

1. **DoesCrossPolygon**: Que comprueba si existen cortes de la línea auxiliar con alguna arista que no sean las que conforman el destino y el final de la línea auxiliar. Si se produce un corte bajo estas condiciones, ese camino ya no es factible.
2. **isOutsideArc**: Comprueba, a través de un calculo de productos vectoriales, si ese camino, pasa por dentro o no del polígono.

Para que un camino sea posible es necesario que no cruce ninguna arista que pertenezca al polígono de estudio y que este camino siempre sea externo y nunca cruce por dentro del mismo.

$calcula\_fronteras$ da como resultado un matriz de $nodospol_{polk}$ x $nodospol_{polk}$, los valores iguales a 1 son los posibles caminos que cumplen las dos condiciones citadas anteriormente. Identificando estos puntos (teniendo en cuenta que *naristas* guarda los identificadores numéricos de los *waypoints*), es posible entonces establecer una matriz que proporcione la información contraria, es decir, que caminos son prohibidos.

```
172 %% Caminos prohibidos
173 Caminos_prohibidos=zeros(N,N);
174
175 for jj=1:length(EVG.poligono2nodo(:,1))
176     %Extraemos poligono
177     npol=jj;
178     naristasextend=EVG.poligono2nodo(npol,:);
179     indcero=find(naristasextend==0,1);
180     naristas=naristasextend(1:indcero-1)';
181
182     polygonVertices=WPs(naristas,:);
183     possiblePaths=calcula_fronteras(polygonVertices);
184
185     %Ahora se adapta esto al formato de caminos prohibidos
186     %possiblepaths(a,b) es 1 cuando el camino es posible, por lo que el camino no está prohibido
187
188     % possiblepaths tiene el formato de naristas por naristas
189
190     for kk=1:length(naristas)
191         for ll=kk+1:length(naristas)
192             if possiblePaths(kk,ll)==0 %0 sea si el camino no es posible, está prohibido
193                 a=naristas(kk);
194                 b=naristas(ll);
195                 Caminos_prohibidos(a,b)=1;
196                 Caminos_prohibidos(b,a)=1;
197             end
198         end
199     end
200 end
201
202 % % 1 no permitido, 0 permitido
203 % %Las aristas consecutivas son caminos permitidos.
```

Código 2.7 Identificación de Caminos_prohibidos.

2.2.4 Cálculo de las matrices de ordenación

La siguiente sección entronca directamente con uno de los puntos claves del documento original de referencia de Ghosh & Mount, [6], que se basa en la idea de que la complejidad propuesta del algoritmo es alcanzable siempre y cuando los datos y operaciones que se narrarán a continuación sean calculables con complejidad $O(1)$. Todo esto se fundamenta en el *Extended Visibility Graph*, en la implementación realizada se ha creado una estructura de datos a la que se hace referencia en todas las funciones y subfunciones con las siglas en inglés EVG.

Por ejemplo, $nodo2poligono$ se ha almacenado dentro de la estructura de datos EVG, dado que contiene información de interés para varias funciones.

El diagrama de visibilidad extendido: EVG

Estrictamente hablando en el método originalmente propuesto se plantean dos posibilidades, computar la triangulación del dominio y luego el Grafo Extendido de Visibilidad (EVG), o computarlo todo de forma simultanea. En la implementación realizada se ha tomado la opción de procesarlo todo a la par para evitar almacenar información acerca de la triangulación y luego información extra acerca del propio Diagrama Extendido.

Conceptualmente, una triangulación que respete las obstáculos que se plantean dentro del dominio, podría funcionar como un protodiagrama de visibilidad, $proto-$ en el sentido de que existirían

muchos falsos negativos, pero funcionaría bajo ciertas situaciones concretas. La triangulación propuesta en el artículo original, [6], es una versión generalizada del método de Mehlhorn, [9], preparada para que triangule recintos con huecos en su interior.

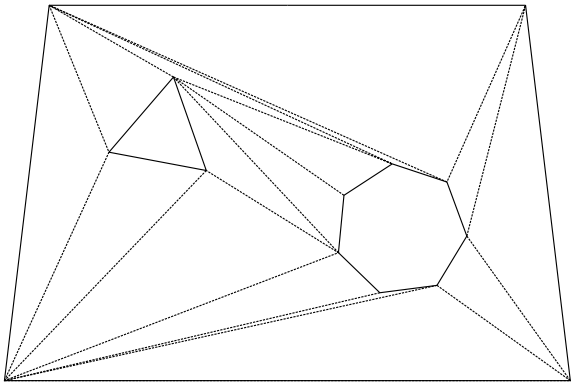


Figura 2.3 Ejemplo práctico de triangulación respetando obstáculos.

Aquí ya podrían plantearse ciertas dudas respecto a la complejidad algorítmica propuesta en el artículo original, [6], donde se plantea que el método concreto es capaz de ejecutarse con una complejidad de $O(E + n\log n)$. Donde E es el número de lineas de visión en el grafo de visibilidad, que en el peor de los casos dado que el susodicho grafo puede ser trabajado como una matriz de dimensiones N x N, siendo N el número de puntos de paso, será una fracción ξ, que dependerá de la geometría de trabajo. Esto podría tender de forma aproximada como:

$$O(\xi N^2) \backsimeq O(N^2). \tag{2.2}$$

Sumado a esto, hay que considerar que la complejidad algorítmica asociada a $O(n\log n)$ está asociada a la computación de la triangulación de un entorno libre de obstáculos, es mencionado en el artículo que la triangulación que se consideró fue fundamentalmente similar a la de Mehlhorn, pero con una *sencilla generalización*, de la que no se aportan más detalles. Triangular un dominio sin agujeros es más sencillo que hacerlo con ellos, dado que se añade la necesidad de revisar si se está respetando adecuadamente la geometría de los susodichos agujeros. Este y otros aspectos serán analizados de forma crítica en la sección de revisión del método e implementación.

El diagrama de visibilidad extendido plantea que las siguientes operaciones puedan ser realizadas en tiempo constante y con complejidad $O(1)$ para un vértice v y una linea de visión (u,v) incidente en v dados:

- Localización de las dos aristas pertenecientes al dominio triangulado incidentes en v. Que en la implementación se ha sacado más partido usando en su lugar $Consec_{aristas}$, que no pertenecen al dominio triangulado sino que conforman las aristas que preceden y suceden a v. Se puede considerar que esta información está registrada en $vectoru$, véase la sección 2.3.2 para una explicación más detallada.
- Los sucesores horarios y antihorarios de u,v.
- Las extensiones horarias y antihorarias de u,v.
- El reverso de (u,v). Que es trivial, dado que solo implicaría permutar las posiciones.

Estas son las operaciones que se requiere que se puedan hacer sobre el grafo de visibilidad extendido, pero existen más datos que se almacenarán en el mismo, las familias de embudos y otras variables de interés precomputadas, que serán añadidas en la implementación en la sección 2.2.5.

Sucesores horarios y antihorarios

Según la sección **7. Data Structure** del artículo original, [6], se plantea que la implementación de la siguientes operaciones debe plantearse empleando una doble lista de adyacencia, tal que las entradas estén organizadas en el orden propuesto. Para simular este efecto, se han empleado estructuras vectoriales ordenadas para un punto dado, registrando así los nodos en un vector fila. Agrupando todos estos vectores fila en orden de punto de paso se obtiene la matriz $Matrizorden$.

El proceso de ordenación se lleva a cabo al analizar cada punto de paso. Por esta razón, la matriz $Matrizorden$ tendrá dimensiones $N\ x\ N-1$, ya que un nodo no puede ordenarse angularmente en relación consigo mismo. Se utiliza un nodo como referencia, que corresponde al *ángulo 0*, ya sea el nodo 1 o 2 en el caso particular del nodo 1, y se calculan los ángulos de ese nodo respecto a la referencia. Este ángulo se determina aplicando las propiedades fundamentales del producto escalar:

$$\vec{a}\cdot\vec{b} = |\vec{a}|\left|\vec{b}\right|\arccos(\theta), \tag{2.3}$$

para evitar las ambigüedades que comúnmente presenta la función inversa del coseno, es a adicionalmente comprobada la tercera componente del producto escalar. Todos estos valores angulares son ordenados usando el comando *sortrows*, dando como resultado el vector de nodos ordenados en sentido horario, que recorriendo de forma inversa el vector se conseguirá la versión en sentido antihorario. El uso de esta matriz para calcular sucesores será desarrollado en mayor detalle en el Apéndice A, sección A.3.2.

Con el objetivo de no usar comandos *find* para buscar posiciones específicas de nodos dentro del vector fila, se usa la matriz $Mat\_ind\_pos\_enMatrizorden$ que contiene la información contraria, es decir en que posición del vector fila se encuentra el nodo de interés.

Esto es interesante para calcular el sucesor horario o antihorario de un nodo en concreto, dado que si se quisiera saber cual es el sucesor dentro de la secuencia, sería necesario localizarlo y consultar el siguiente valor o el anterior en la secuencia. Es por ello que el disponer de esta información de antemano convierte el cálculo de $O(N)$ en $O(1)$.

```
207 %% Cálculo de las matrices de ordenación
208 % Esto es crucial para el desarrollo del algoritmo, se esecifica como dato
209 % del problema en la sección de estructura de datos.
210 %La matriz será NxN, donde el input es por filas, es decir, para una fila k
211 %dada, cada uno de los valores de esa fila son los otros nodos ordenados alrededor de
212 %el en sentido de las agujas del reloj visto desde el nodo k empezando por el primer nodo de
     waypoint (ya sorted).
213 Matrizorden=zeros(N,N-1); %N-1 por que el propio punto no se puede ordenar respecto de si mismo
214 filaaux=zeros(N,1);
215
216 for ii=1:N
217     %Formado elvector auxiliar a recorrer
218     if ii==1
219         filaaux=2:N;
220     elseif ii==N
221         filaaux=1:(N-1);
222     else
223         filaaux=[1:(ii-1),(ii+1):N];
224     end
225
226     vbase=WPs(filaaux(1),:)-WPs(ii,:);
227     modvbase=norm(vbase);%Para calcular el ángulo
228
229     angulovec=zeros(2,N-1);
230     angulovec(2,:)=filaaux;%el primer nodo será el que corresponda al punto de base sobre el que
      medimos
231
232     for jj=2:(N-1)%Se ordenan los N-2 nodos restantes (Es decir, todos menos si mismo y el que
      usamos de referencia)
233         vpunto=WPs(filaaux(jj),:)-WPs(ii,:);
234         modvpunto=norm(vpunto);
235         anguloesc=acos(dot(vpunto,vbase)/(modvpunto*modvbase));%En radianes!
```

```
236
237         %Discriminamos el sector para evitar ambigedades
238         qz=vpunto(2)*vbase(1)-vpunto(1)*vbase(2); % usamos el producto escalar para ver si queda
        a la izq o a la derecha del semiplano y asi discriminar el ángulo
239         %El producto vectorial es el de vbase sobre vpunto
240
241         if qz<0  %No puede ser igual a 0, implicaría que hay tres puntos en linea.
242             anguloesc=2*pi-anguloesc;
243         elseif qz==0
244             warning('Tres puntos son colineares.')
245         end
246         angulovec(1,jj)=anguloesc;
247         [180/pi*anguloesc, qz, filaaux(jj),vbase,vpunto];
248     end
249
250     angulovec=sortrows(angulovec',1)';%Version ordenada de la matriz, la segunda fila son los vé
        rtices ordenados en sentido creciente de los ángulos
251     Matrizorden(ii,:)=angulovec(2,:);
252 end
253
254
255 %Matrizorden: Ordena los nodos en sentido horario partiendo desde el primer
256 %o el segundo nodo.
257 % Pongamos un ejemplo práctico, queremos saber cual es el nodo incidente en
258 % i que se encuentra a continuación de j en sentido horario: Eso se haría tal que Matrizorden(i,j
        +1).
259 % De forma análoga, el nodo incidente en i que se encuentra a continuación
260 % de j en sentido antihorario seria Matrizorden(i,j-1)
261
262 %% Matriz de correlación posición en Matrizorden con nodo
263 % Para evitar llamar varias veces al comando find dentro del algoritmo de
264 % cálculo de los CWs y los CCWs, precomputamos las posiciones de estas
265 % mismas, dado que esto es una cuestión geométrica.
266
267 Mat_ind_pos_enMatrizorden=nan(N,N);
268
269 for ii=1:length(Matrizorden(:,1))
270
271     for jj=1:length(Matrizorden(1,:))
272         valoraux=Matrizorden(ii,jj);%Da el valor del nodo
273         Mat_ind_pos_enMatrizorden(ii,valoraux)=jj;%Sería la operación inversa a matriz orden, en
        vez de decirnos la secuencia que tienen los nodos, dice la posición de la secuencia.
274     end
275 end
276
277 %Ejemplo de uso de Mat_ind_pos_enMatrizorden, queremos saber, que posición en Matrizorden tiene
        el
278 %nodo j para la secuencia de nodos alrededor de i, lo que debería hacerse
279 %es: posicion= Mat_ind_pos_enMatrizorden(i,j)
```

Código 2.8 Cálculo de los sucesores CW/CCW.

Nótese que ese postprocesado una vez más supera las expectativas de complejidad presentadas. Dado que se hacen dos bucles recorriendo el rango completo de N y al final de la segunda capa de bucle, se realiza una ordenación con el comando *sortrows* cuya complejidad algorítmica es, en el mejor de los casos, $n \log n$, es decir, la que correspondería a un *Merging sort*. A continuación se plantea la complejidad algorítmica de este *cuello de botella* presente en el preprocesado:

$$O(N(N+N\log N)) \backsimeq O(N^2 \log N). \tag{2.4}$$

Se adjunta en la figura 2.4, un ejemplo visual de cómo funciona la operación sucesores.

Padres posibles

En esta sección del código se revisa para cada combinación de punto de paso y polígono que nodos son factibles como padres, este concepto de paternidad cobrará sentido cuando se introduzcan los nodos en la sección 2.2.7. La función $test\_de\_paternidad_Main$, cuya programación será desarrollada

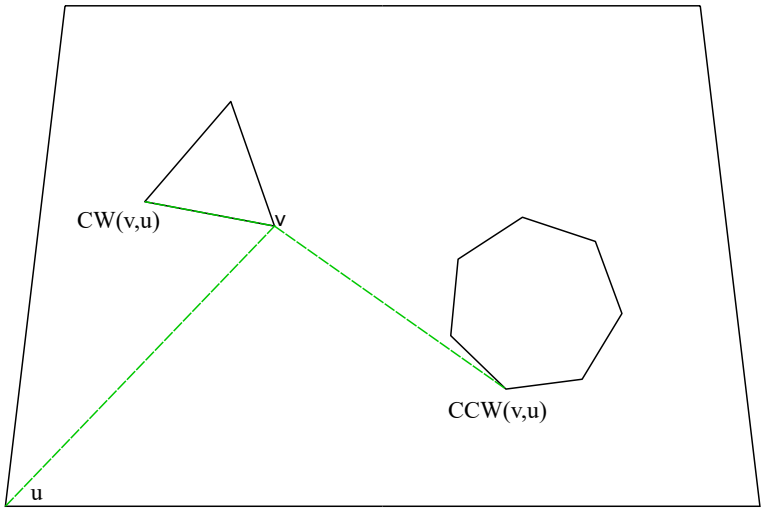


Figura 2.4 Sucesores CW y CCW.

en el Apéndice A, sección A.3.2, revisa tres condiciones geométricas que validan si un nodo *nodopropuesto* perteneciente a un polígono tiene potencial para ser padre para un ancestro dado x:

1. Todos los nodos que pertenecen al polígono al que pertenezca x son potenciales padres. Es por ello que,
2. todos los nodos pueden ser padres para si mismos, dado que ellos son parte de la cadena de la que son ancestros.
3. Si *nodopropuesto* y x pertenecen a diferentes polígonos, solo podrán ser padres los que se encuentren en la cara oculta del polígono de *nodopropuesto* o bien la tangente al polígono ubicada más en el sentido de las agujas del reloj respecto a x.

Todas estas comprobaciones se realizan en el código $test\_de\_paternidad_Main$, en versiones tempranas del código esto se calculaba según se necesitaba disponer de él pero considerando que es una propiedad geométrica que no depende directamente del cálculo del grafo puede precomputarse.

```
281 %% Matriz padres posibles
282 EVG.MATposiblepadre=NaN(N,N);%Primera para nodox, segunda para el nodo a comprobar
283 EVG.WPs=WPs;
284
285 for jj=1:N
286     %Actualizar un nodo cambia todos los del polígono
287
288     for kk=[0 1:length(EVG.poligono2nodo(:,1))]
289         if kk==0
290             nodopropuestopadre=1;
291         else
292             nodopropuestopadre=EVG.poligono2nodo(kk,1);
293         end
294
295         [EVG]=test_de_paternidad_MAIN(jj,nodopropuestopadre,EVG);
296     end
297 end
298
299 EVG.MATposiblepadre(1,1)=1;
300 EVG.MATposiblepadre(2,2)=1;
301 EVG.MATposiblepadre(N-1,N-1)=1;
302 EVG.MATposiblepadre(N,N)=1;
303 EVG.MATposiblepadre(1,2)=1;
```

Código 2.9 Identificación de padres posibles.

Extensiones horarias y antihorarias

Esta operación no ha sido implementada directamente en el código, en la sección 2.2.7, se desarrolla una serie de criterios similares para discernir cuando es factible insertar nodos en familias, pero por resaltar las diferencias con el método original, se realizará la presentación de la operación.

Para calcular la extensión horaria $CX(u,v)$ es necesario seguir los siguientes pasos:

1. Se traza un arco de radio $|\vec{uv}|$ de 180 centrado en v en sentido horario.
2. Si la totalidad de ese arco pasa por el interior del dominio P, que es el conformado por la substracción de los agujeros poligonales al recinto exterior; la extensión será el siguiente segmento visible en sentido horario. Siempre que se mantenga todo el recorrido del arco dentro de P. En caso de que ese arco no estuviera dentro en su totalidad del dominio P, se considera la extensión no definida.

La extensión antihoraria se extrae siguiendo los mismos pasos y restricciones salvo que trazando el arco en sentido antihorario.

Se adjunta la imagen 2.5, con diferentes casos y aplicaciones para mayor claridad. Resaltadas en color verde se encuentran las extensiones factibles, y en roja un ejemplo de una extensión no definida.

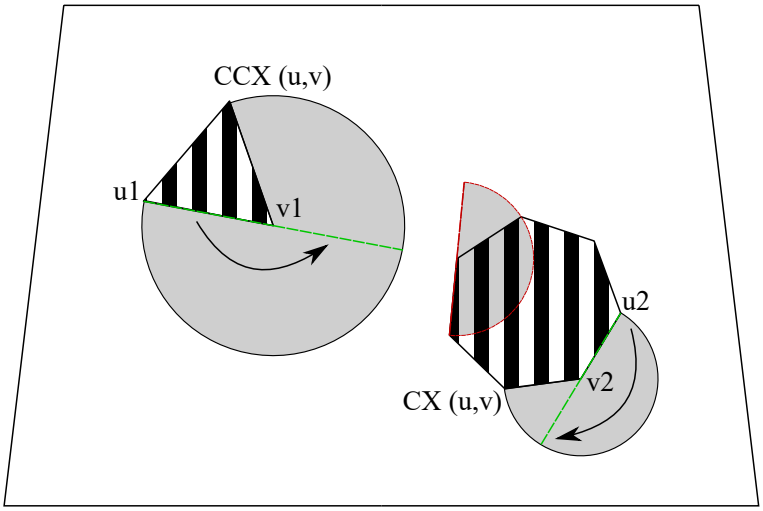


Figura 2.5 Extensiones CX y CCX.

2.2.5 Formando la estructura de datos

Este es todo el preprocesado que será realizado con la información proporcionada por la geometría. A continuación todas las variables computadas y otras que serán detalladas a continuación se añaden a la estructura de datos EVG.

```
304 %% Montando la estructura de datos
305 % Hay muchas variables precomputadas, para evitar entradas a funciones
306 % demasiado largas, las organizaremos todas en el Enhanced Visibility Graph
307
308 % EVG.WPs,Caminos_prohibidos,Matrizorden, Mat_ind_pos_enMatrizorden
309
310 Visibilidad=zeros(N,N);
311
312 EVG.Visibilidad=Visibilidad;
313 EVG.Caminos_prohibidos=Caminos_prohibidos;
314 EVG.Matrizorden=Matrizorden;
315 EVG.Mat_ind_pos_enMatrizorden=Mat_ind_pos_enMatrizorden;
316 EVG.TOT_famxv_glob={};
```

```
317 EVG.TOT_famvy_glob={};
318 EVG.nodosTOT_famvy_glob={};
319 EVG.nodosTOT_famxv_glob={};
```

Código 2.10 Estableciendo el diagrama de visibilidad extendido EVG.

2.2.6 Antes de iniciar el bucle: Presentando la triangulación

Una vez toda la información está preprocesada, es necesario presentar dos conceptos cruciales acerca del flujo de trabajo que tendrá el algoritmo implementado.

La algoritmia entera se basa en los siguientes dos preceptos:

1. **Expansión del dominio procesado mediante triangulación**.
2. **Determinación de visibilidad mediante recorrido de familias de embudos**.

Los detalles de la algoritmia de triangulación serán presentados junto con el código, pero siguiendo lo citado en la sección 2.2.4, se responderá al porqué plantear en un primer lugar un algoritmo de triangulación.

Recordando la figura 2.3, existe cierta sinergia entre la triangulación de un dominio y la visibilidad entre diferentes puntos planteados en el mismo, por lo que es posible sacarle partido para estructurar el funcionamiento del código. El dominio P puede descomponerse en diferentes secciones disjuntas que conformen un todo:

$$P_k = \cup_{i=1}^{k} T_k, P_k = T_1 \cup T_2 \cup T_3 \ldots \cup T_{k-1} \cup T_K. \tag{2.5}$$

Estas secciones T_k son las secciones trianguladas fruto de conectar el punto que se está añadiendo al grafo de visibilidad, que para ser congruentes con la implementación se denominará *ii*, con el dominio P_{k-1} a través de los puntos de su envolvente convexa. Se adjunta la figura 2.6 para explicar el método de progresión aditiva. Se van visitando los diferentes puntos de paso en orden de coordenada x creciente y se recoge esta información.

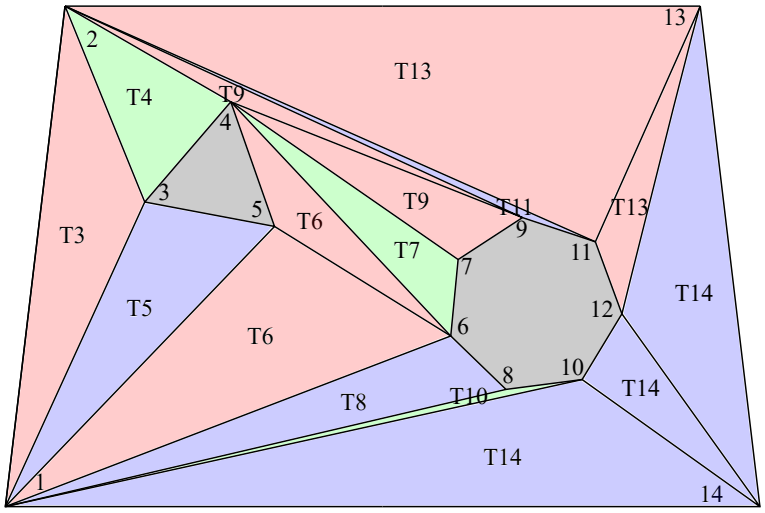


Figura 2.6 Dominio triangulado compuesto.

En la ilustración 2.6, se puede observar que algunas extensiones de dominio T_k pueden ser un espacio nulo, como es el caso de 1 o 2, dado que no puede trazarse un área al añadirse un punto a un espacio nulo, y de manera acorde formar un área con dos puntos. Para el caso del nodo 12,

los únicos puntos de la envolvente con los que puede conectar son, los vértices consecutivos de su polígono, generando que $T_1 2$ sea como T_1 y T_2 un espacio nulo a su vez.

Por otro lado conectar $ii = 6$ a la envolvente de P_5 genera que T_6 esté compuesto por dos subtriángulos, resultado de realizar la substracción $l - 1$, siendo l el número de conexiones posibles, respetando la geometría de los agujeros, entre los nodos de la envolvente de P_5 y $ii = 6$, véase la figura 2.7. Nótese que el hueco delimitado por el triángulo 3,4,5 se encuentra embebido en P_5, pero técnicamente la conexión se hace con el lado externo de la arista que es una entidad perteneciente al dominio P_5, no directamente con el polígono.

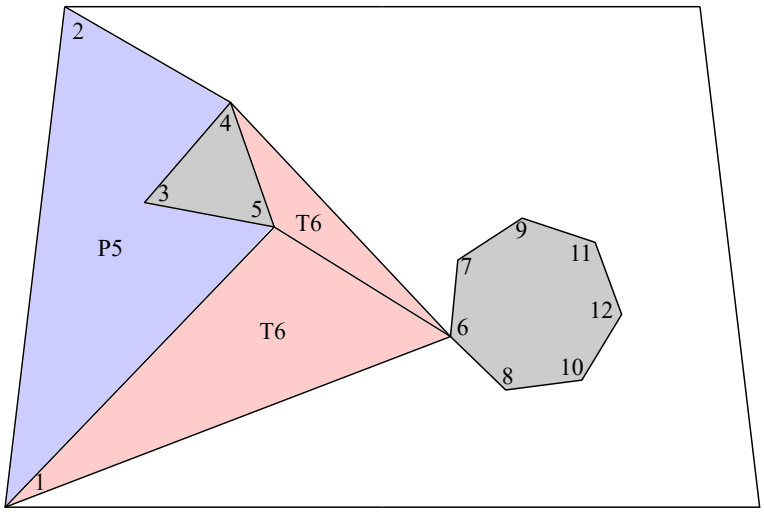


Figura 2.7 Dominio triangulado compuesto: $ii = 6$.

En resumen, el algoritmo de visibilidad está embebido en la construcción de esta triangulación, cuya implementación será desarrollada en la sección 2.3.2. Es por ello que el procesado es aditivo, y plantea un bucle externo en N, el número de diferentes puntos de paso.

2.2.7 Antes de iniciar el bucle: Presentando los embudos

El salto lógico necesario para llegar de la triangulación a la generación del diagrama de visibilidad la proveen los embudos y las familias de los mismos. Estas entidades matemáticas surgen de manera orgánica a la hora de resolver problemas de camino más corto y grafos de visibilidad.

Un embudo en un dominio cualquiera queda definido de forma unívoca por los siguientes grados de libertad:

- **Una recta conformada por dos puntos del dominio que conforman una linea de visión**: que se definirá de forma consistente por los puntos (x,y) y se llamará base, esta nomenclatura también se usará en el código.
- **Un punto externo a esa recta**: que en el código toma el nombre *ii*, que se denominará ápice. Sólo se consideran los nodos que están más allá del semiplano en el que se encuentra el ápice.
- **El nodo de estudio y su padre en el árbol inferior**: a esta construcción de dos puntos se le denomina nodo principal y nodo padre.

En la figura 2.8, pueden observarse dos embudos respecto a los nodos marcados como x,y, y el ápice ii. Todos los embudos tienen un camino que conecta con el nodo de la base x, este camino se denominara árbol inferior, y otro camino que conecta con el nodo de la base y, que conformará el árbol superior. Estos caminos quedan definidos por la regla de la *goma elástica*.

La propiedad más interesante de los embudos, tal y como han quedado definidos, es que el área que en la figura queda marcada por colores rojo y verde para cada embudo, puede demostrarse geométricamente que está vacía de obstáculos.

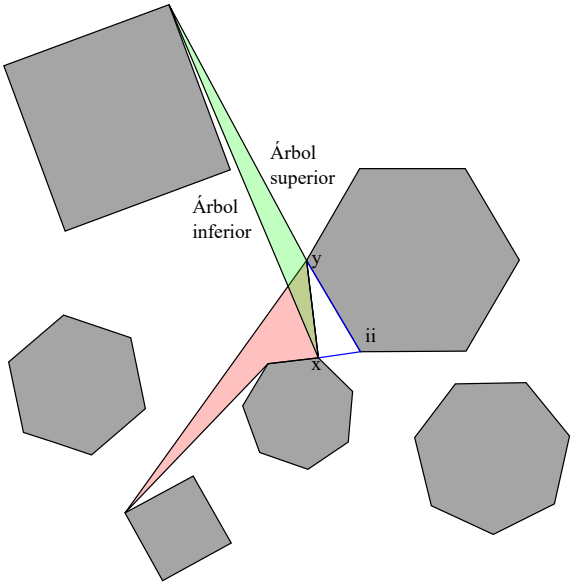


Figura 2.8 Ejemplos de embudo en un dominio cualquiera.

La regla de la *goma elástica* consiste en conectar el nodo principal del embudo a un punto de la base y desplazar, usando de guía la base, hasta alcanzar el nodo de la base elegido, y desde ahí hacer tirante la goma. Este procedimiento se muestra visualmente en la figura 2.9, ejecutada para el árbol superior e inferior de un nodo dado.

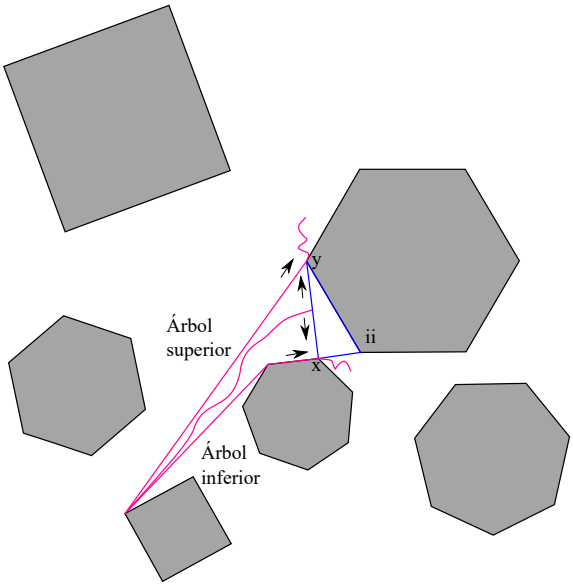


Figura 2.9 Regla de la goma elástica.

Dependiendo de la geometría del dominio esto podría llevar a cierta ambigüedad, como la que se muestra en la figura 2.10, que según de la disposición inicial de la goma pudiera tomar dos caminos posibles, uno más cercano a x y otro que se apoya en el lado derecho del triángulo, ambos señalados en la parte izquierda de la imagen. Esta ambigüedad en este caso puede resolverse gracias a la

aplicación de la regla de la *goma elástica*, y entenderse de forma más sencilla con la ordenación.

La parte derecha de 2.10, muestra diferentes nodos que no deberían ser padres por diversas razones, para poner en práctica la aplicación de las características y propiedades de los embudos se detallarán de forma pormenorizada las razones que impiden la paternidad:

1. **a**: *a* no puede conectar directamente con *x*, al buscar él un padre que si lo haga, o que conectara a la altura de alguna rama del árbol inferior, como la de *padre*1, no se generaría un camino convexo. Por ende, no es válido.
2. **b**: atraviesa una arista perteneciente a una región prohibida.
3. **c**: no cumple la regla de la *goma elástica*, dado que geométricamente ese punto no da soporte en el árbol inferior, pero si en al superior como en la imagen de la izquierda.
4. **d**: igual que *c*, no puede dar soporte en el árbol inferior, salvo que en este caso, tampoco podría servir de soporte en el superior.
5. **e**: por un lado atraviesa aristas pertenecientes a una región prohibida, y además, es imposible que se cumpla la regla de la *goma elástica* en el nodo *y*, que es en el sentido metafórico de los embudos, estéril.
6. **f**: igual que para *d*, *f* no puede ser padre en el árbol superior al no poder cumplirse la ya citada regla.

Una vez repasado esto, se puede inferir que la regla de la *goma elástica* acaba implicando las siguientes restricciones: independientemente de si se busca identificar padres en el árbol superior o inferior, la regla de la goma elástica busca minimizar la longitud de los caminos, para ello sigue puntos de tangencia que mantengan la convexidad del mismo. Es decir, un padre válido es aquel que se encuentra a la vuelta de la esquina. Esto queda matemáticamente descrito en la parte de la implementación de la ordenación de familias, Apéndice A.

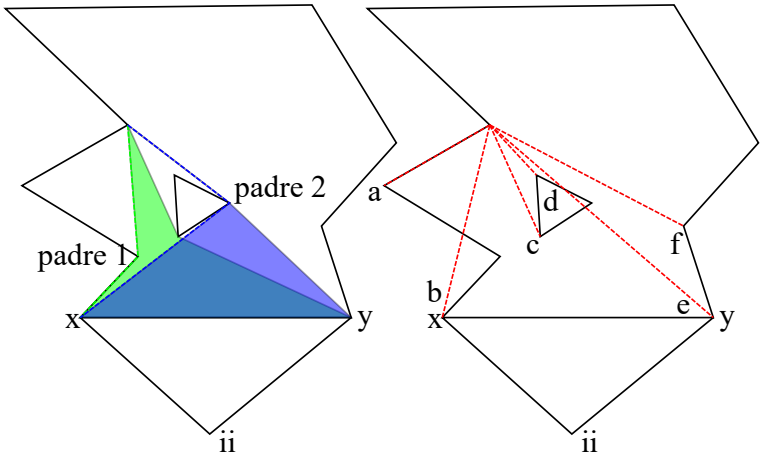


Figura 2.10 Ambigüedad de padres factibles y muestra de padres no válidos.

El concepto de ordenación surge cuando se busca agrupar todos los embudos que pertenecen a una misma base, una vez identificados. Por razones prácticas, es conveniente establecer una manera unívoca de establecer un orden, del que algorítmicamente hablando se pueda sacar partido. El orden propuesto es el recorrido en preorden en sentido horario, los nodos de la figura 2.11, se identifican como lo harían los puntos de paso, en orden creciente de *x*, y se muestran la estructura ya ordenada en . Nótese en la figura 2.12, que el orden de la ecuación 2.6 es el reflejado por las letras en la parte superior izquierda de las esferas.

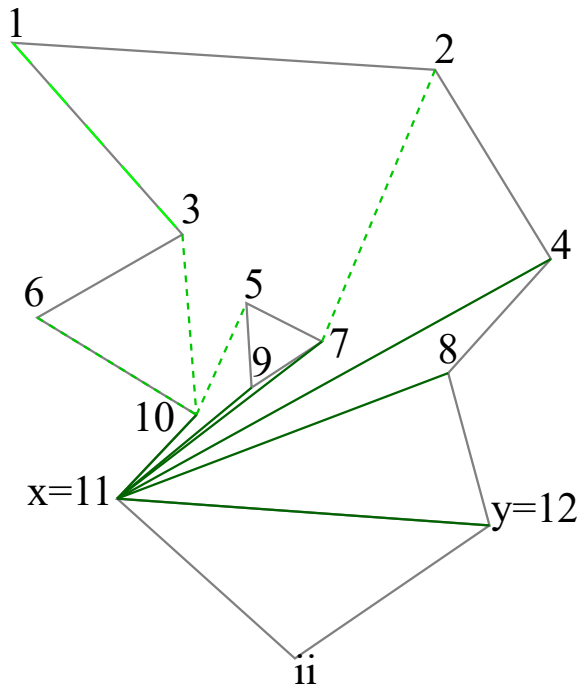


Figura 2.11 Familia de embudos para (x,y).

Construyendo las relaciones parentales entre embudos, siguiendo las normas anteriormente citadas, se parte del nodo x, el ancestro o raíz del árbol inferior, se disponen a continuación todos sus hijos ordenados en sentido horario. Se revisa para cada uno de sus hijos, en sentido horario si estos tienen descendientes directos, se adjuntan en la lista inmediatamente después del padre. Una vez se han revisado todos los padres de segunda generación, la primera generación la conformaba el nodo x, se comprueba si entre los nodos de tercera existe algún padre y así hasta que o se ubican todos los nodos pendientes o se identifica que hay nodos que no presentan padres compatibles. Esos nodos en concreto son aquellos que ya no interesa guardar ya que no existiría una potencial línea de visión entre él y nodos que se encuentren en el otro semiplano delimitado por la base. En la implementación esta ordenación se hace a la par que la asignación de padres.

Estas familias de nodos asociadas a la base (x,y) son identificadas por el código y el método originalmente propuesto como $FNL(x,y)$. Estos a su vez comparten la característica de que ven a los segmentos de la base extendida a ii, qeu se denominarán xv, a la que conecta con x con ii y vy a la que conecta ii con y.

Trabajar de esta manera presenta ciertas ventajas, gracias a esta ordenación, una vez fijado el árbol inferior, si se aplica la operación sucesor antihorario $CCW(\text{padre,hijo})$ para los nodos $k < ii$, puede localizarse el padre en el árbol superior. Gracias a esto, el embudo queda vacío si el padre en el árbol inferior está bien identificado.

La familia de embudos $FNL(x,y)$ se dispondría tal y como se muestra en la ecuación 2.6. La columna de la izquierda representa los nodos en sí, en sentido de orden de embudo, y la segunda, los padres que tendrían asociados a esa base.

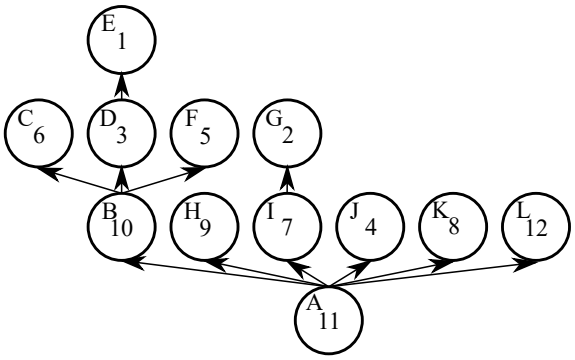


Figura 2.12 Recorrido en preorden en sentido horario de la familia presente en la figura 2.11.

$$\mathbf{FLN}(\mathbf{11},\mathbf{12}) = \begin{bmatrix} 11 & 11 \\ 10 & 11 \\ 6 & 10 \\ 3 & 10 \\ 1 & 3 \\ 5 & 10 \\ 7 & 11 \\ 2 & 7 \\ 4 & 11 \\ 8 & 11 \\ 12 & 11 \end{bmatrix} \tag{2.6}$$

Nótese que el nodo x y el nodo y para $FNL(x,y)$ deben siempre estar presentes, y en las posiciones inicial y final de la familia. Debido a la ordenación, el embudo x, es un embudo degenerado y se referencia a si mismo. Asimismo, el embudo y siempre hará referencia a x. Esto es debido a que la familia solo se plantea cuando la base es una linea de visión. Además, todos los hijos suceden a sus padres, no puede darse una familia en la que o bien el padre está ausente o el hijo lo preceda, dado que inestabilizaría la algoritmia.

Almacenamiento y acceso a las familias

Una de las principales diferencias entre la implementación realizada y el documento original, es que se almacenan de forma activa y sin discriminar, todas las familias. Según la metodología propuesta por Ghosh & Mount, [6], respecto de la cual se proporcionan ciertas indicaciones generales, no guarda pero si modifica de forma activa toda esta información, sobreescribiendo tan pronto se procesan los datos en el algoritmo *SPLIT*. Para ello se indica en el artículo que se empleó una estructura de datos basada en la de Gabow & Tarjan, [10], cuya implementación ha quedado fuera del alcance del proyecto debido a su complejidad. La implementación de una estructura de datos a partir de los conocimientos y experiencia del redactor, siguiendo las directrices mencionadas en el apartado, no ha sido viable, por lo que se ha usado un tratamiento de los datos más generalista.

Los principales retos a la hora de implementar una estructura de datos que permitiera alcanzar los estándares de la originalmente planteada han sido los siguientes:

- Acceso a la información presente en las familias de embudos a través de los datos del EVG. Se plantearon diversas soluciones, pero todas requerían de diferentes niveles de procesado de los datos y no mantenían la complejidad $O(1)$ propuesta.
- Debido a estos problemas, garantizar que no hubiera falsos negativos en cuanto a visibilidad, se torno imposible, dado que cualquier fallo en el procesado de la información se propagaba

a lo largo de toda la ejecución. Es por ello que la algoritmia implementada invierte muchos recursos en evitar que estos fallos tengan lugar.

- Se propone la algoritmia original con familias sobreescribiendose unas encima de otras, planteando situaciones en las que la información disponible en la iteración pudiera no ser suficiente.

Durante el transcurso de este proyecto, se han probado diferentes estrategias de almacenaje y procesado de familias, desde preprocesar todas las combinaciones posibles de familias a partir de las familias dividas tras la triangulación, lo cual daba resultados muy buenos para valores bajos de N. También se probó con una estrategia similar a la presente en el artículo en la que se van almacenando según se ubican los nodos de interés y aprovechando el orden natural computado en la iteración anterior, lo cual no terminaba de resultar consistente, cuando los triángulos que conformaban los conos de visión eran muy obtusos, el orden de embudo para una base dada bajo estas circunstancias cambiaba y forzaba que se desecharan estos ordenes. Al final, la solución que mejor rendimiento ofreció, fue tomar la familia asociada a la base intentando mantener este orden obtenido de la división de las familias y reconstruirlas siguiendo una serie de criterios geométricos. Siendo necesario para ello, almacenar los nodos para una base sin genealogía en una estructura de datos, almacenando la información en el diagrama de visibilidad extendido. Para ello se ha empleado la función $add\_familia2$, que reconstruye las familias bajo demanda de las llamadas a $SPLIT$, minimizando el almacenamiento de estas familias al mínimo, a la par que permitiendo reconstruirlas en cualquier momento de la ejecución sin pérdida de información.

La implementación de los criterios geométricos empleada en $add\_familia2$ será desarrollada en profundidad en el Apéndice A, sección A.3.3, pero sus propiedades serán redactadas a continuación:

1. Las cadenas en el árbol inferior deben mantenerse convexas.

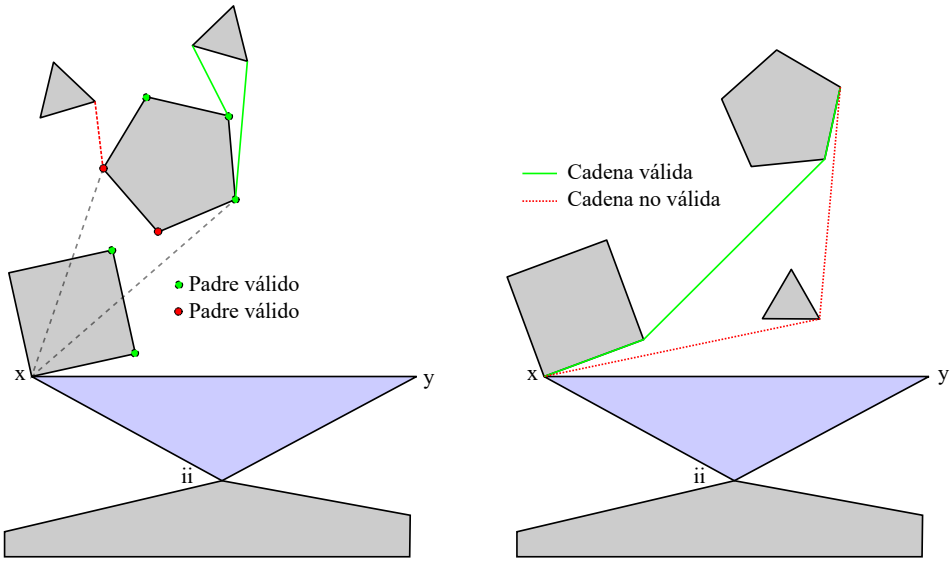
2. No todos los nodos son válidos como padres para un ancestro, *nodox* dado. Los nodos de un polígono del cual *nodox* no tome parte solo serán válidos como padres en el caso de que estén en la cara oculta a *nodox*, salvo en el caso de que sea el último nodo del polígono en orden horario con *nodox* como eje de rotación. Véase la figura 2.13.a.

3. El árbol superior del nodo a introducir debe ser aquel que produzca la cadena de longitud más corta, que no necesariamente es la que menos generaciones presente. Véase la figura 2.13.b.

4. No se pueden introducir nodos en la familia que se encuentren fuera del cono de visión si para ello se conectan a *nodox* desde fuera del mismo. Véase la figura 2.13.c.

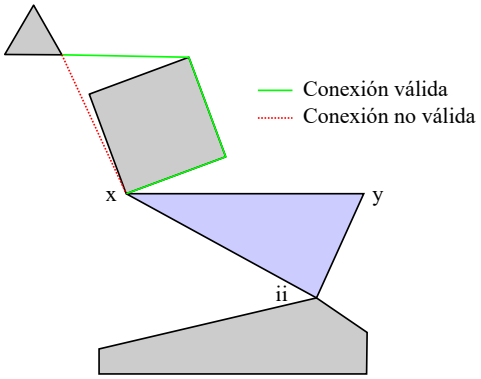
Para poder identificar que nodos son visibles a través del cono de visión, se toman todos los nodos asociados a las familias almacenadas en la base, dependiendo de que punto de paso tenga mayor coordenada x. Estos nodos se guardan al final de cada bucle, una vez concluidas las ejecuciones de $SPLIT$, almacenándose respectivamente en $EVG.nodosTOT\_famvy\_glob\{ii\}$ y $EVG.nodosTOT\_famxv\_glob\{ii\}$.

Una vez obtenidos los nodos, se ordenan respecto a *nodox* empezando por *nodoy* en sentido de las agujas del reloj y se desechan aquellos que quedan en el semiplano del lado ii. Una vez identificados los potenciales nodos, se llama al algoritmo de reconstrucción de familias $add\_familia2$, que aplicará los criterios desarrollados anteriormente y devolverá la familia.Este procedimiento de ordenación y selección de nodos se ilustra en la figura 2.14. Los únicos nodos que se puede garantizar sin ninguna duda que se encontrarán dentro de la familia son *nodox* y *nodoy*, para los cuales se conoce su posición, debiendo abrir y cerrar la familia.



Caso a) Validez de padres

Caso b) Longitud de cadenas mínima



Caso b) Longitud de cadenas mínima

Figura 2.13 Casos ejemplificados de los criterios aplicados.

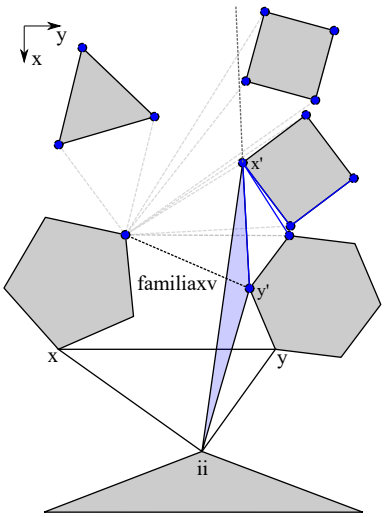


Figura 2.14 Reordenación de la familia partiendo de una base dada.

```
function familiaembudo=calculafamilia(nodox,nodoy,EVG,ii)
%El nodo más alto fue el último en computarse y es el que identifica si la
%familia que se busca es xv o vy. Por ejemplo (15,14) es necesariamente una
%familia computada en la iteración 15, de tipo vy
if nodox>nodoy
    %Consulta a vy
    nodos_TOT=EVG.nodosTOT_famvy_glob{nodox};
else
    %Consulta a xv
    nodos_TOT=EVG.nodosTOT_famxv_glob{nodoy};
end

nodos_TOT(nodos_TOT(:,1)==nodox)=[];
nodos_TOT(nodos_TOT(:,1)==nodoy)=[];

if ~isempty(nodos_TOT)
    posnodoy=EVG.Mat_ind_pos_enMatrizorden(nodox,nodoy);           % Se saca la posición de nodoy
      en la secuencia de nodoorden

    nodoorden=EVG.Matrizorden(nodox,:);                            % Se extrae el vector
     nodoorden que contiene los nodos ordenados en sentido CCW
    nodoorden=[nodoorden(posnodoy+1:end),nodoorden(1:posnodoy)];   % Se ordena en CCW dejando
     primero a nodoy
    nodoorden=nodoorden(end:-1:1);                                 %Ahora y es primero en orden
     CW
    [~, indicefiltrado]= ismember(nodoorden, nodos_TOT(:,1));       % Se filtran los valores
     posibles a partir de los nodos que pertenencian a subfamxv
    nodos_TOT=nodoorden(indicefiltrado>0)';                         % En este caso, se tienen
     todos los nodos ordenados respecto a ii

    %nodos_TOT contiene ahora todos los nodos ordenados en un vector columna

    %Se va a buscar descartar aquellos nodos que no sean factibles porque se
    %encuentren por detrás de la familia

    p2=EVG.WPs(nodox,:)';
    p3=EVG.WPs(nodoy,:)';
    v2=p3-p2;


    for jj=1:length(nodos_TOT)
        nodocheck=nodos_TOT(jj);
        p1=EVG.WPs(nodocheck,:)';

        v1=p1-p2;

        prodvec=v1(1)*v2(2)-v1(2)*v2(1);

        if prodvec<0 %Esto significa que se encuentra en el semiplano válido de los nodos
            if jj~=1
                nodos_TOT=nodos_TOT(jj:end);%si el primer nodo es válido entonces, no hay
     necesidad de recortar los anteriores
            end
            break
        elseif jj==length(nodos_TOT)&&prodvec>0
            nodos_TOT=[];%ningun nodo es válido
        end
    end
    %Se dispone en forma de familia
    nodos_TOT=[nodos_TOT,nodos_TOT];
end

%Se añaden los nodos x e y
nodos_TOT=[nodox,nodox;nodos_TOT;nodoy,nodox];

%Se llama al procesado
[familiaembudo]=add_familia2(EVG,nodos_TOT);
end
```

Código 2.11 Función calculafamilia.

2.3 Bucle en ii

Para realizar la triangulación del dominio de forma aditiva, véase la sección 2.2.6, es una necesidad recorrer todos los puntos de paso, añadiéndolos adecuadamente al grafo de visibilidad extendido. desde aquí, todo el código está metido en un gran bucle for, que recorre todos los puntos de paso y se identifica el nodo que se procesa activamente con ii.

2.3.1 Nodos $ii = 1$ y $ii = 2$

El primer y el segundo punto de paso son los únicos que no pueden procesarse de manera normal, dado que no tienen área y se encuentran mirando enteramente fuera del dominio. Por lo que es necesario extender $FNLextend$ y introducir los datos asociados a las familias a mano en EVG.

Desde la perspectiva del punto de paso 1, como las familias solo pueden incluir puntos de paso iguales o inferiores a ii dado que el proceso es aditivo, la familia está compuesta de un único embudo. Para 2 el único embudo posible, considerando que todos los puntos de paso estarán ubicados en valores superiores de coordenada x, es la familia 1,2, que a su vez solo puede contenerse a si misma.

```
322 %% Loop general
323 % Por comodidad se procesan los primeros nodos por separado
324
325 ii=1;
326 EVG.TOT_famxv_glob{ii}=[1,1];
327 EVG.TOT_famvy_glob{ii}=[1,1];
328 EVG.nodosTOT_famvy_glob{1}=1;
329 EVG.nodosTOT_famxv_glob{1}=1;
330
331 ii=2;
332 EVG.TOT_famxv_glob{ii}=[1,1;2,1];
333 EVG.TOT_famvy_glob{ii}=[2,2;1,2];
334 EVG.nodosTOT_famvy_glob{2}=[1;2];
335 EVG.nodosTOT_famxv_glob{2}=[1;2];
336
337
338 EVG.Visibilidad(1,2)=1;
339 EVG.Visibilidad(2,1)=1;
340
341 %Para comprobar la visibilidad respecto a la envolvente
342 npol=length(EVG.poligono2nodo(:,1));
343 vectorpol=[];%Inicialización de la variable
```

Código 2.12 Procesado manual de los primeros puntos de paso.

Por otro lado se inicializa la variable $vectorpol$, que permitirá ahorrar tiempo de procesado a la hora de determinar los puntos a visitar.

2.3.2 Ejecución normal del bucle: Triangulación

A partir de este momento comienza el proceso de triangulación junto con el de determinación de visibilidad. Para ello se irá desarrollando conforme se muestra en el código, las decisiones que llevaron a la implementacíón final aquí desarrollado. Que se ha interpretado de manera más libre debido a que no se especifica en el documento de referencia [6], como se generaliza el método de Mehlhorn [9], para recintos con agujeros poligonales.

Cabe destacar que es necesario terminar de preparar el terreno para el algoritmo de triangulación definiendo variables auxiliares, para ello se toma el periodo de ejecución del punto de paso 3, del que se deducen sus propiedades siempre y cuando se respeten las directrices señaladas para el recinto externo (sección 2.2).

```
5  %% Bucle en ejecución
6  t1=toc;
7  tic
8  for ii=3:N
9      %% %% Generación vector u
10
11     if ii==3 %No se puede sacar la envolvente de 2 puntos
12
13         %vectoru contiene la envolvente de puntos en la primera columna, vectoruextend en la
       segunda
14         %contiene los signos en formato + o - 1
15         vectoru=[1;2];
16         vectoruextend=[1,1,1;2,1,1];
17         % vectoruextend(ii,3) guarda la información de si es visible o no. 1 si, 0
18         % no
19         %vectorrecorre es el subfragmento a SPLITear
20         vectorrecorre=[1;2];
21
22         %Valores postactualización
23
24         pol=EVG.nodo2poligono(3);
25         vectorpol=[vectorpol;pol];%añadimos a vectorpol el polígono 1, que corresponde por
       definición al que comienza en 3
26         vectoru=[1;3;2]; %Se añade 3 a la secuencia
```

Código 2.13 Inicio de la triangulación.

La variable *vectoru* conforma la envolvente de puntos que ya han sido procesado, nótese que no siempre va a contener todos los puntos procesados, solo el frente de aquellos que tengan los mayores valores de coordenada x. Por otro lado *vectorpol*, considera los polígonos que pudieran hacer pantalla a los puntos de paso que se procesen en futuras iteraciones. Por ejemplo, si el punto 4 está en el mismo polígono, sería necesario revisar que por cuestiones geométricas ninguna arista de ese polígono cortara la visibilidad con alguno de los puntos de la envolvente. Igual sería si 4 perteneciera a otro polígono, dado que sería el primer punto que se estuviera procesando sería imposible que el polígono al que pertenezca entorpeciera la linea de visión de la envolvente. Para ilustrar esta casuística se añade la figura 2.15.

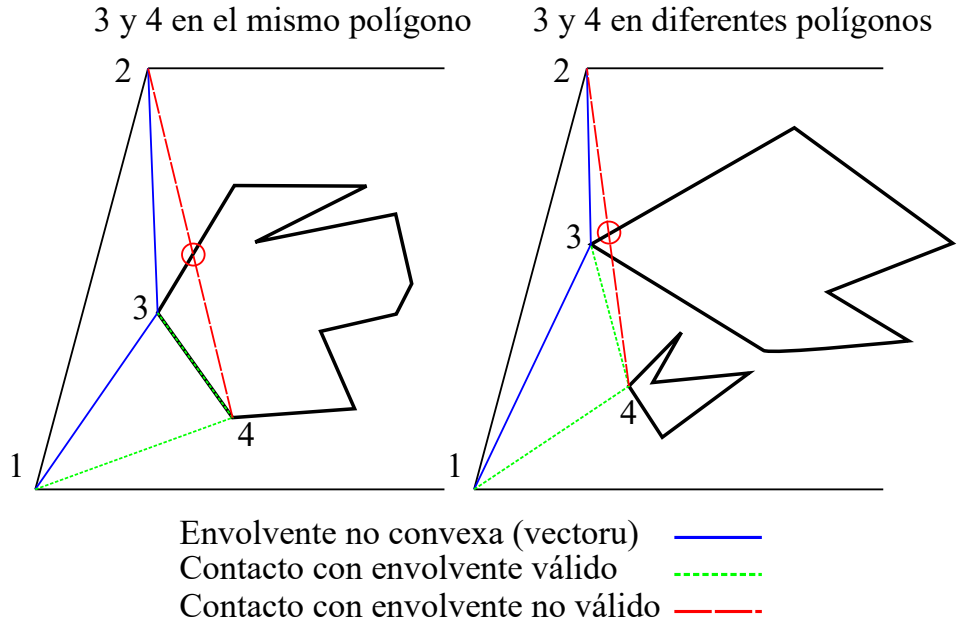


Figura 2.15 Ejemplo de procesamiento de *vectorpol*, bloqueo de visión respecto a la envolvente.

```
368     else
369
370         %Atención: LOS VALORES DE U SE TOMAN DE LA ITERACIÓN ANTERIOR
371         vectoruextend=compruebasigno(vectoru,EVG,ii); %Aquí se le añadirá la segunda columna. Se
        revisan todos los valores
```

Código 2.14 Llamada a compruebasigno desde el algoritmo de triangulación.

En el caso de que el punto de paso sea superior a 3, se llamará a la función *compruebasigno*. Esta función llama a una subrutina que añade una segunda columna con los valores de los signos del cociente de los productos vectoriales que surgen de la siguiente manera: se obtiene el vector que va desde *ii* hasta el punto anterior de *vectoru* respecto al que se está analizando, análogamente se procesa el vector posterior; cada uno de estos vectores se proyecta sobre el vector que que va de *ii* hacia el punto sobre el que se quiere calcular el signo. Este cociente si se normaliza respecto a si mismo, da -1 si el punto de estudio no es tangente y 1 si lo es. Estas proyecciones pueden verse de forma visual en la figura2.16 y su implementación en la caja de código 2.15.

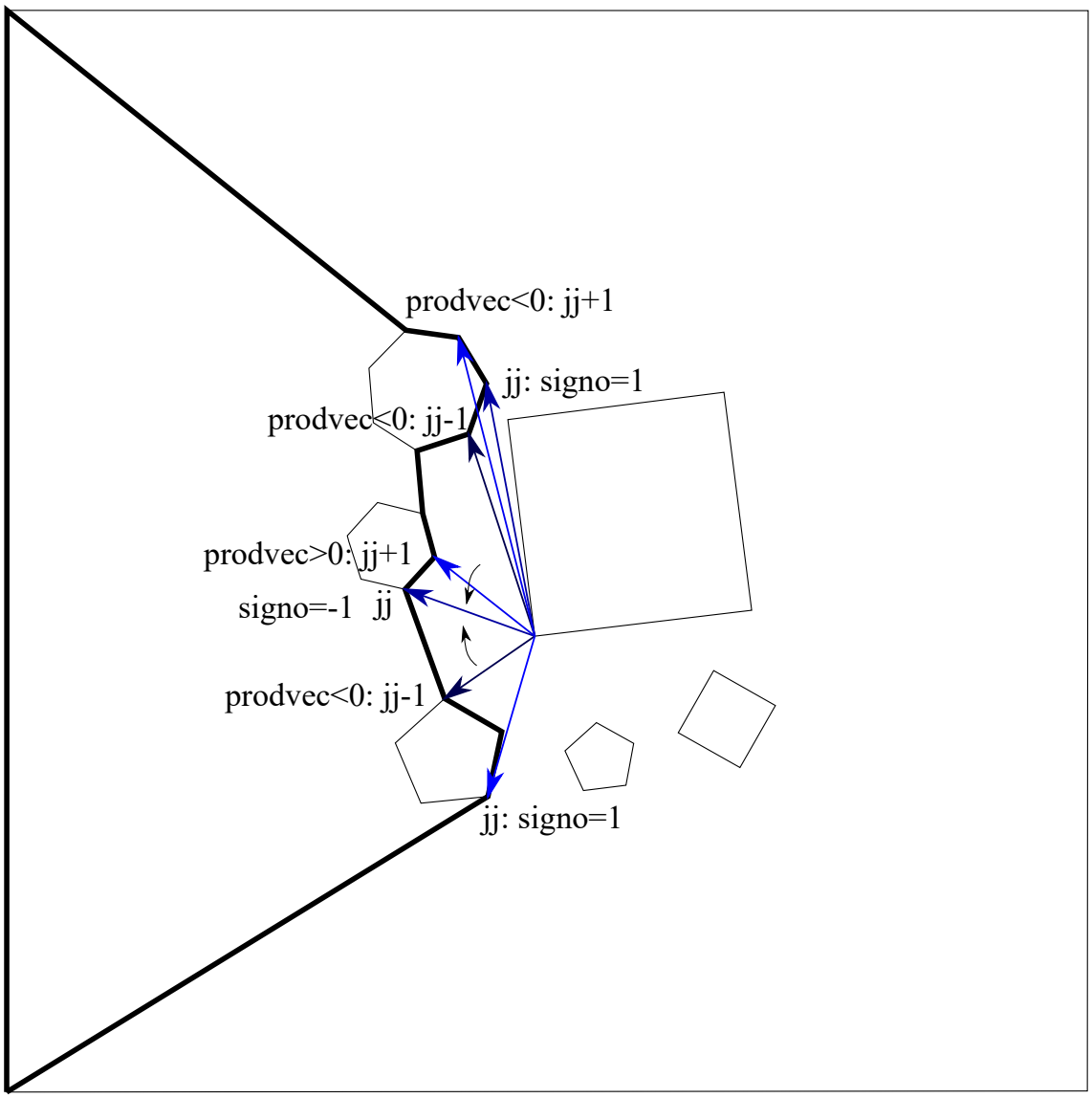


Figura 2.16 Ejemplo de la lógica de *compruebasigno*.

```
%% Función comprueba signo
% Comprueba como son los signos de los diferentes puntos a recorrer

function vectoruextend=compruebasigno(vectoru,EVG,nodoii)
vectoruextend=[vectoru,zeros(length(vectoru),1)];
vectoru=[vectoru(end);vectoru;vectoru(1)];

v1=zeros(2,1);%Vector que lleva de N a p1(punto anterior)
v2=zeros(2,1);%Vector que lleva de N a p2(punto a evaluar)
v3=zeros(2,1);%Vector que lleva de N a p3(punto posterior)

p1=zeros(2,1);%Punto anterior
p2=zeros(2,1);%Punto a evaluar
p3=zeros(2,1);%Punto posterior
p4=EVG.WPs(nodoii,:)';%Punto asociado a N

for jj=2:(length(vectoru)-1)
    %Así evitamos que tener que utilizar casos específicos para los valores
    %de 1 y 2
    p1=EVG.WPs(vectoru(jj-1),:)';
    p2=EVG.WPs(vectoru(jj),:)';
    p3=EVG.WPs(vectoru(jj+1),:)';

    v1=p1-p4;
    v2=p2-p4;
    v3=p3-p4;

    prodveca=v1(1)*v2(2)-v1(2)*v2(1);
    prodvecb=v3(1)*v2(2)-v3(2)*v2(1);

    signo=prodveca*prodvecb/abs(prodveca*prodvecb);

    vectoruextend(jj-1,2)=signo;
end

end
```

Código 2.15 Función compruebasigno.

Todos los puntos de interés estarán ubicados entre dos tangentes, y se evaluará si se procesaran sus familias de embudos en función de sus visibilidad.

Una vez se ha distinguido cuales son los puntos que potencialmente podrían resultar interesantes para analizar según la metodología de Ghosh & Mount, [6], es necesario comprobar si existen interrupciones de la visibilidad con este subconjunto de valores que se encuentran entre dos tangentes.

Para ello se identifica cual es el primer y el último valor igual a 1 en el *vectoru*, usando dos comandos *find*, uno en sentido directo y otro inverso. Gracias a esto se puede evitar consultar el vector entero, debido a que la búsqueda para cuando se encuentra el único nodo de interés.

```
        % Se computa el vector subu, solo  hace falta revisar entre el primer 1(pos) y
        % el último 1 (pos)

        %% Creación vectorsubu

        %El vector subu es el segmento del vector que potencialmente contiene los
        %elementos a recorrer (si no hubiera aristas que cortan la visión vectorsubu sería igual
    a vector recorre)

        ind1=find(vectoruextend(:,2)==1,1);% Primera aparición de 1
        ind2=length(vectoruextend(:,2))-find(vectoruextend(end:-1:1,2)==1,1)+1; %Última aparición
     de 1

        vectorsubu=vectoru(ind1:ind2);
        vectorsubuextend=[vectoruextend(ind1:ind2,:)];

        %% Comprobación visibilidad
        % Estas visibilidades se calculan revisando los conjuntos de polígonos y la
```

```
389         % envolvente
390
391         aristaintersec=NaN(2,1);
392         poligono2nodoextend=[EVG.poligono2nodo;vectoru',zeros(1,N-length(vectoru))];%comprobamos
        sobre la envolvente completa
393         vectorpolextend=[length(poligono2nodoextend(:,1));vectorpol];
```

Código 2.16 Identificación de polígonos de interés en la triangulación.

Para comprobar si existen cortes de visibilidad con los puntos concretos de la envolvente no convexa, se seguirá el siguiente procedimiento:

1. Definición de variables de utilidad en el bucle:

- *aristaintersec* es una variable que almacena la última arista que cortó la visibilidad para el punto anterior, una vez se inicia el bucle esta información todavía no existe.
- *poligono2nodoextend*: es una versión extendida de *polgono2nodo* que contiene además, información acerca del polígono que conforma la envolvente no convexa, que podría causar intersecciones en casos concretos, dado que su perímetro puede desarrollar dientes de sierra.
- *vectorpolextend*: que contiene la misma información que *vectorpol*, acerca de que polígonos son susceptibles de interrumpir la visibilidad, pero añadiendo el polígono $n+1$, que sería el de la envolvente. Como este cambia para cada iteración, es necesario actualizarlo.

2. Se recorren todos los valores de *vectorsubu* buscando cortes, comprobando primero si *aristaintersec* sigue bloqueando la visibilidad. *inter* funciona como una bandera, en el momento en el que se detecta una intersección se levanta, $inter = 1$, y se registra acordemente que no es visible el punto de estudio y se pasa al siguiente. Esta información acerca de la visibilidad se guarda en la respectiva fila de la tercera columna $vectorsubuextend(jj,3)$. Si por un casual, la conexión entre *ii* y el punto de estudio es un *Camino_prohibido*, véase la sección 2.2.3 para más detalle, se interpreta que hay una intersección dado que la visibilidad esta bloqueada pero no por una intersección, sino por producirse este intento de visión dentro de uno de los agujeros.

a) Para cada valor de *vectorsubu* que no se haya podido descartar de forma preliminar, es necesario comprobar si hay intersecciones con los polígonos registrados en *vectorpol*, se recorre cada uno de ellos buscando una intersección.

b) Las aristas a estudiar están registradas de manera conveniente en *poligono2nodoextend*, por lo que eliminados los valores nulos residuales, para cada pareja de vértices consecutivos, que conforman una línea, se busca si existe intersección con la recta que va desde *ii* hasta el nodo de estudio.

c) Este corte con las aristas de los polígonos solo puede darse si se cumplen las siguientes condiciones:

- Al menos uno de los vértices que compone la arista debe tener preceder a *ii*. Si están más allá de *ii* no puede producirse un corte.
- No se considerará ninguna de las aristas que tengan a *ii* o al nodo de *vectorsubu* que se estudie en ese momento. Dado que solo puede existir una intersección entre dos rectas, y ese punto sería o bien *ii* o el nodo de estudio.

d) Se guardan las coordenadas asociadas a los puntos de paso de la arista que se vaya a estudiar *line*1, y la recta que une *ii* y el punto de estudio, *line*2.

e) Se calcula si existe un punto de intersección usando geometría euclidiana básica, que será desarrolla y mostrada en el correspondiente Apéndice A, sección A.3.2.

f) En el caso de existir se corta la revisión de aristas gracias al levantamiento de la variable bandera *inter* y se paran de procesar las aristas gracias a un *break*.

3. Por defecto, no existe corte hasta que se detecta uno, es por ello que $vectorsubuextend(jj,3) = 1$ es establecido al principio del bucle y cuando se levanta la bandera se cambia a o.

```
395         for jj=1:length(vectorsubu)
396             % 'siguiente   u'
397             inter=0;
398             contadorpol=1;
399             vectoraristaextend=zeros(1,length(poligono2nodoextend(1,:)));
400 
401             %Antes de empezar a calcular para cada polígono debería comprobarse si
402             %aristaintersec sigue bloqueando la visión del nuevo punto
403 
404             %% Comprobamos visibilidad con arista intersec
405 
406             if ~isnan(aristaintersec(1,1)) && aristaintersec(1)~=ii && aristaintersec(2)~=ii &&
    aristaintersec(1)~=vectorsubu(jj) && aristaintersec(2)~=vectorsubu(jj)
407                 %Si no es nan calculalo, si no lo es, compruebalo normal
408 
409                 %line1 es la línea que une N con el punto de la secuencia a comprobar
410                 %line2 es la línea con la que se busca que interseque (cualquiera de las aristas
    de los polígonosconsiderados)
411                 line1=[EVG.WPs(ii,1),EVG.WPs(vectorsubu(jj),1);...
412                     EVG.WPs(ii,2),EVG.WPs(vectorsubu(jj),2)];
413                 line2=[EVG.WPs(aristaintersec(1),1),EVG.WPs(aristaintersec(2),1);...
414                     EVG.WPs(aristaintersec(1),2),EVG.WPs(aristaintersec(2),2)];
415 
416                 [inter,~]=cortapuntos(line1,line2);
417 
418                 if inter==1
419                     %si hay intersección se registra y arista intersec
420                     %se queda igual.
421                     vectorsubuextend(jj,3)=0;
422                     % 'CorteAristaintersec'
423                 end
424             end
425 
426             if Caminos_prohibidos(ii,vectorsubu(jj))==1
427                 inter=1;
428                 vectorsubuextend(jj,3)=0;
429                 % 'Camino prohibido'
430             end
431 
432             %% Se comprueban cortes con cada uno de los polígonos candidato
433             while inter==0 && contadorpol<=length(vectorpolextend(:,1))  % En el momento que se
    identifica un corte la línea N-vectorsubu(jj) es no visible
434                 % 'comprobación pol'
435 
436                 poligono=vectorpolextend(contadorpol);
437                 vectoraristaextend=poligono2nodoextend(poligono,:);%Extend incluye para poligono2
    nodo el vectoru, para arista los 0 sobrantes
438                 naristaspol=find(vectoraristaextend(1:1:end)==0,1)-1;
439                 vectorarista=[vectoraristaextend(naristaspol),vectoraristaextend(1:naristaspol),
    vectoraristaextend(1)];
440                 %Procesarlo así hace mucho más sencillo gestionar las parejas a
441                 %procesar
442                 vectorsubuextend(jj,3)=1;
443 
444                 % Para cada polígono se comprueban los cortes con cada arista
445                 for kk=2:(naristaspol+2)
446                     if (vectorarista(kk-1)<ii || vectorarista(kk)<ii) && vectorarista(kk-1)~=ii
    && vectorarista(kk)~=ii && vectorarista(kk-1)~=vectorsubu(jj) && vectorarista(kk)~=
    vectorsubu(jj)%Las dos últimas condiciones sirven para que no se compruebe si hay intersecci
    ón con las aristas que pasan por el punto que vamos a añadir. Esto es así porque daría un
```

```
      falso positivo diciendo que existe intersección entre las aristas que conforman el punto que
       vamos añadir, pero esas no bloquean la visibilidad, serían otras del polígono.
447                       %line1 es la línea que une N con el punto de la secuencia a comprobar
448                       %line2 es la línea con la que se busca que interseque (cualquiera de las
      aristas de los polígonosconsiderados)
449                       %Las lineas vendrán dadas en el formato
450                       % [x1, x2
451                       % ;y1,y2]
452
453                       line1=[EVG.WPs(ii,1),EVG.WPs(vectorsubu(jj),1);...
454                           EVG.WPs(ii,2),EVG.WPs(vectorsubu(jj),2)];
455                       % linea1=[ii,vectorsubu(jj)]
456                       line2=[EVG.WPs(vectorarista(kk-1),1),EVG.WPs(vectorarista(kk),1);...
457                           EVG.WPs(vectorarista(kk-1),2),EVG.WPs(vectorarista(kk),2)];
458                       % linea2=[vectorarista(kk-1),vectorarista(kk)]
459                       [inter,~]=cortapuntos(line1,line2);
460                       if inter==1
461                           % 'Corte'
462                           % Guardamos la información de la arista que cortó
463                           % para empezar ahí la siguiente iteración
464                           aristaintersec=[vectorarista(kk-1),vectorarista(kk)];
465                           vectorsubuextend(jj,3)=0;
466                           break
467                       end
468                   end
469               end
470               contadorpol=contadorpol+1;% cambio de polígono
471           end
472           inter=0;%reseteo para comprobar el siguiente valor de u
473       end
```

Código 2.17 Cálculo de puntos de corte en el área visible de la envolvente.

Una vez obtenida la información acerca de que puntos están entre dos tangentes y además no encuentran su visión obstaculizada por los polígonos potenciales ni la propia envolvente no convexa, se computa el *vectorrecorre*.

```
477         vectorrecorre=vectorsubuextend(vectorsubuextend(:,3)==1);
```

Código 2.18 Identificación de *vectorrecorre*.

Este *vectorrecorre* tiene su contraparte en el documento original de Ghosh & Mount en el Lema 2.1, [6], del que deriva la siguiente casuística:

- *vectorrecorre* contiene dos puntos: Es decir, solo esa línea de visión puede servir como cono para conectar el punto *ii* al dominio triangulado extendido P_k.
- *vectorrecorre* contiene más de dos puntos: El dominio se puede ampliar añadiendo más de un triángulo, pero existen ahora diferentes conos de visión a comprobar. La información de estos conos de visión deberá luego sintetizarse. Este caso y el de dos puntos son los más comunes.
- *vectorrecorre* solo tiene un valor: Esto significa que al no existir cono de visión, el único contacto posible es el de *ii* con el único valor perteneciente al vector. Esto sucede para cuando el punto de paso está *a la vuelta de la esquina*, y su única conexión es otro punto anterior, por lo general del mismo polígono, que bloquea la visibilidad.
- *vectorrecorre* está vacío: Esto sólo sucede en recintos no convexos, cuando existen dos puntos de paso que conectan a un tercero entre los dos, que tiene menor coordenada x que los anteriores. Estas aristas laterales actúan como una pantalla y hacen imposible cualquier tipo de conexión con la envolvente.

Esta casuística puede verse representada para un mismo dominio para diferentes valores de punto de paso en la figura 2.17.

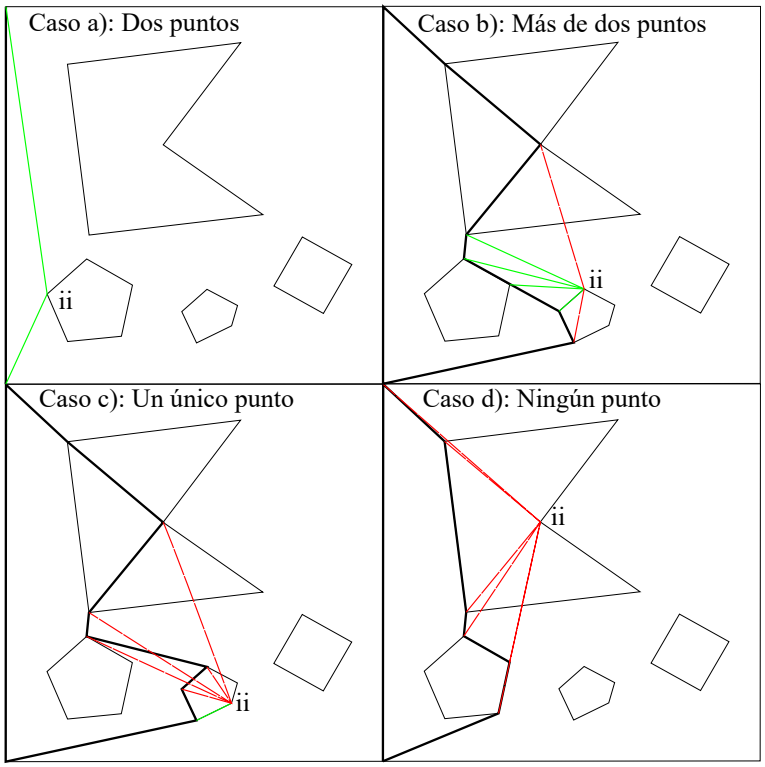


Figura 2.17 Diferentes casos para *vectorrecorre*.

Cada caso influye a su vez con la manera de introducir *ii* en el *vectoru* de cara a la siguiente iteración:

- *vectorrecorre* contiene dos puntos o más: Se localiza en *vectoru* la ubicación de los dos nodos extremos de *vectorrecorre*, se introduce *ii* sobrescribiendo los nodos que hubiera entre medias.
- *vectorrecorre* solo tiene un valor: Dado que no se tiene referencia de entre que valores hay que insertar el nuevo nodo *ii*, debido a que solo hay un valor, es necesario establecer algún tipo de regla. Se comprueba cuales son los nodos consecutivos asociados al nodo de *vectorrecorre* y se insertan en orden de *y* creciente. Distinguiendo dos casos: *xv* si *ii* tiene mayor coordenada *y* y *vy* en caso contrario.
- *vectorrecorre* está vacío: Se recorre el *vectoru*, insertando *ii* cuando se detecte la primera posición en la que su posible predecesor y su posible predecesor tengan menor y mayor coordenada *y* respectivamente. Esto es debido a que se carece completamente de referencias de *vectorrecorre* acerca de donde debería conectar.

```
479         %% Actualización vectoru
480         % Añadiendo valores a vectoru
481
482         ind1=find(vectorsubuextend(:,3)==1,1);
483         if ~isempty(ind1)
484             nodo1=vectorsubuextend(ind1);
485             %Dónde se ubica el valor nodo1 en vectoru?
486             ind1=find(vectoru(:)==nodo1,1);
487
```

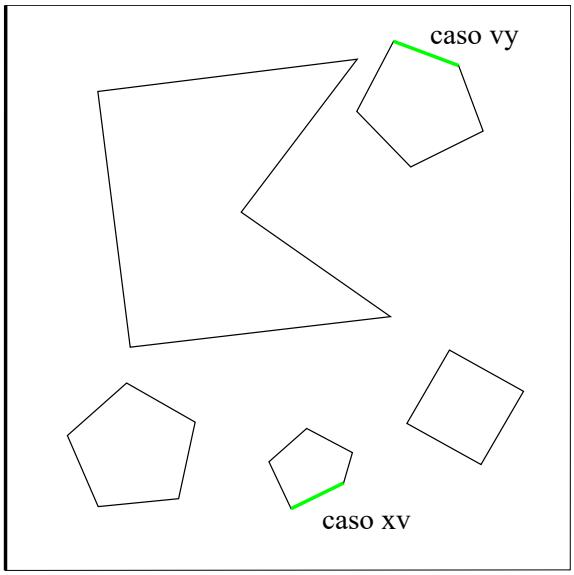


Figura 2.18 *vectorrecorre* contiene solo un nodo: Caso *xv* y caso *vy*.

```
488             ind2=length(vectorsubuextend(end:-1:1,3))+1-find(vectorsubuextend(end:-1:1,3)==1,1);
489             nodo2=vectorsubuextend(ind2);
490             %Identificación de la posición nodo2 en vectoru
491             ind2=find(vectoru(:)==nodo2,1);
492
493             if (ind2-ind1)>=1
494                 vectoru=[vectoru(1:ind1);ii;vectoru(ind2:end)];
495                 %Enfasis en que si el nuevo punto conecta con dos vértices de pk-1, este
496                 %vector se reduce.
497                 caso='';
498             elseif length(vectorrecorre)==1
499                 if EVG.nodo2poligono(ii)==EVG.nodo2poligono(vectorrecorre)
500                     % nodosconsec=find(EVG.Consec_aristas(ii,:)==1);
501                     nodosconsec=EVG.Consec_aristasrec(ii,:);
502
503                     if EVG.WPs(nodosconsec(1),2)<EVG.WPs(nodosconsec(2),2)
504                         %se orden y dejamos arriba el que tenga mayor componente y
505                         nodosconsec=[nodosconsec(2),nodosconsec(1)];
506                     end
507
508                     if nodosconsec(1)==vectorrecorre
509                         caso='vy';
510                         vectoru=[vectoru(1:ind1-1);ii;vectoru(ind1:end)];
511                     else
512                         caso='xv';
513                         vectoru=[vectoru(1:ind1);ii;vectoru(ind1+1:end)];
514                     end
515                 else
516                     if EVG.WPs(vectorrecorre,2)>EVG.WPs(ii,2)
517                         caso='vy';
518                         vectoru=[vectoru(1:ind1-1);ii;vectoru(ind1:end)];
519                     else
520                         caso='xv';
521                         vectoru=[vectoru(1:ind1);ii;vectoru(ind1+1:end)];
522                     end
523                 end
524             else
525                 error('Error on Vectoru iteration, please check sign algorithm.')
526             end
527         else
528             for jj=2:length(vectoru)
529                 if  EVG.WPs(ii,2)<EVG.WPs(vectoru(jj),2) && EVG.WPs(ii,2)>EVG.WPs(vectoru(jj-1)
    ,2)
530                     vectoru=[vectoru(1:jj-1);ii;vectoru(jj:end)];
531                     break
```

```
532                 end
533             end
534         end
```

Código 2.19 Actualización *vectoru* de cara a la siguiente iteración.

La única acción restante entonces es actualizar el *vectorpol*. Aquí es donde entra *limitespoligono*, véase la sección 2.2.2, si *ii* resulta ser el primer o el último punto de paso del polígono al que pertenece, debe añadirse o eliminarse de *vectorpol*. Este *vectorpol* es casi análogo al vector T empleado en el método de Lee, el que registraba las aristas susceptibles de ser intersecadas, véase 1.3.2.

```
536         %% Añadir/quitar polígonos
537         %Actualización vectorpol
538 
539         for jj=1:length(EVG.poligono2nodo(:,1))
540             if ii==EVG.limitespoligono(jj,1)
541                 %Si es el valor más occidental hay que empezar a considerar ese
542                 %polígono en la siguiente iteración
543                 vectorpol=[vectorpol;EVG.nodo2poligono(ii)];
544 
545             elseif ii==EVG.limitespoligono(jj,2)
546                 %Si es el valor más occidental hay que dejar de considerar ese polígono
547                 indaux=find(vectorpol==EVG.nodo2poligono(ii),1);%Buscamos el polígono en la
    secuencia vectorpol que concluye en ii
548                 vectorpol=[vectorpol(1:(indaux-1));vectorpol((indaux+1):end)];%Se elimina el
    valor
549             end
550         end
551         %Fin if ii==3 y de la triangulación
552     end
```

Código 2.20 Actualización de los polígonos de interés.

Una vez obtenido el *vectorrecorre* y actualizadas las variables como se recoge en este párrafo, es el turno de pasar a procesar la visibilidad de los puntos. Nótese que en el caso de que se almacenasen los estados de *vectoru* y *vectorpol*, o *vectorrecorre* en última instancia, para cada valor de *ii*, podría procesarse completamente por separado la visibilidad. Esto es debido a que la triangulación solo depende de la geometría del dominio, hecho que también fue comentado por Ghosh & Mount, [6]. En última instancia, podría incluirse como parte del preprocesado, pero dado que los autores originales incluyeron la triangulación dentro de su estimación de la complejidad algorítmica se ha mantenido como parte de un todo.

La complejidad algoritmica de esta parte, que Ghosh & Mount indicaron que se podría realizar en un entorno libre de obstáculos en $O(n\log n)$, una vez añadidos a la ecuación los mismos esta complejidad deja de ser representativa del proceso, y aunque se generalice de forma sencilla, esta complejidad no puede mantenerse.

Se estima entonces que la complejidad algorítmica real implementada asociada a la sección de triangulación es aproximadamente igual a:

$$O\left(N\left(\sqrt{N}+\frac{\sqrt{N}}{2}\sqrt{N}+\sqrt{N}\right)\right) \backsimeq O(N^2). \tag{2.7}$$

Donde la primera N deriva de la necesidad de ejecutar esto para cada punto de paso. El primer sumando del paréntesis surge de estimar el número de valores que posee *vectoru* para ejecutar *compruebasigno*, que depende de N pero suele tener muchos menos puntos de paso en procesado activo dado que todos van sobreescribiendose entre ellos, por lo que se fija como valor conservador $\sqrt{N}$ en entornos complejos con geometrías complejas y muchos nodos.

En el segundo sumando se ha estimado el recorrido de *vectorsubu* y la búsqueda de cortes de las aristas de los polígonos asociados a *vectorpolextend*, estimando *vectorsubu* como la mitad de *vectoru*, lo que puede ser una estimación acertada pero conservadora para una buena variedad de geometrías. La estimación del número de aristas a comprobar es difícil de definir, dado que existiendo *aristaintersec*, se suelen ahorrar muchas ejecuciones. De forma similar a *vectoru*, dado que no se están procesando las aristas de todos los polígonos, por lo que se hace la estimación a su vez de que sean $\sqrt{N}$ comprobaciones.

El tercer sumando solo aplicaría en el caso de que no hubiera ningún nodo en *vectorrecorre*, una vez más, sería un recorrido parcial en *vectoru*, por lo que se toma también la estimación de $\sqrt{N}$.

Lo que aproximando el contenido del paréntesis cuando N tiende a ∞, queda como $O(N^2)$. Superior a lo propuesto, pero mejora todavía la complejidad del método de Lee, [5].

2.3.3 Ejecución normal del bucle: *SPLIT*

Una vez identificados los puntos a añadir a la triangulación, es necesario recorrerlos y así computar el grafo de visibilidad. El procedimiento que se va a describir a continuación solo puede aplicarse en los casos en los que *vectorrecorre* tenga 2 o más valores, en caso contrario, existen excepciones implementadas que se comentarán por separado en aras de la claridad.

Para cada pareja de valores consecutiva que se pueda extraer de *vectorrecorre* será procesada, aplicando las propiedades de los embudos mencionados anteriormente. Gracias a las familias registradas, se es conocido para linea de visión dada, en este caso la que marca la pareja de valores de *vectorrecorre* según la iteración correspondiente, cuales son los nodos que son potencialmente visibles desde el ápice *ii*, véase 2.2.7. Gracias a esta información se puede calcular de manera eficiente cuales son los nodos visibles para *ii*.

Como subproducto de este proceso de calculo de visibilidades, se obtienen las familias de embudos asociadas a *ii*, que servirán para mantener el algoritmo funcionando. Esta función es la que conforma el núcleo de todo el algoritmo, se ha mantenido la nomenclatura del documento original y se le ha llamado *SPLIT*, dado que parte una familia tomando como referencia un tercer punto. Conceptualmente, el resultado debería ser como el mostrado en la figura 2.19, el punto de partida son los embudos visibles y potencialmente visibles para la línea de visión $x-y$, y la salida, las familias ya formadas.

En la práctica, el método mostrado en el documento de Ghosh & Mount, especifica como se apilan las familias de embudos y hace hipótesis sólidas al respecto, pero adelantando las conclusiones, es necesario realizar operaciones de extensión horaria para todos los valores a añadir al grafo. Hecho que a la hora de almacenar y establecer las posibles subfamilias derivadas de xv y vy, $v \equiv ii$, complica la algoritmia a implementar al no usarse en este caso de estudio la estructura de datos de Gabow & Tarjan, [10].

Se ha señalado en la figura 2.19 ciertos embudos que son cruciales para el método, y permiten esquematizar las diferentes operaciones de reparto y comprobación de visibilidad que se realizan dentro de la función:

- **u**: Es el último nodo que bloquea la visibilidad de embudos que se encuentran en el ala izquierda de la familia.
- **u'**: Es el último hijo, en el sentido de las agujas del reloj, de *u*, cuya visión con *ii* está bloqueada por *u*.
- **q**: Es el último nodo del ala izquierda.
- **r**: Es el último nodo del ala derecha, es el sucesor de *q* en orden de embudo.
- **t**: Es el último nodo que bloquea la visibilidad de embudos que se encuentran en el ala derecha de la familia. Es análogo a *t*.

- **s**: Es el padre de *q* en el árbol superior.

Presentados los diferentes nodos de interés, se provee un pseudocódigo con el objetivo de aclarar las diferentes secciones de la función *SPLIT*:

Pseudocódigo 3.1 Cálculo del grafo de Visibilidad mediante algoritmo *SPLIT*

[*familiaxv*,*familiavy*,**EVG**] =SPLIT(**EVG**,*ii*,*nodox*,*nodoy*)
1: Extracción y cálculo de la familia de **EVG**
2: Búsqueda *u* + Recurrencia **SPLIT** en los subconos del ala izquierda
3: **si** $u \neq y$ **entonces**
4: Localización en la familia de *r* y *q*.
5: Adición a *familiavy* de los nodos entre *u* y *q*
6: Recorrido desde *nodoy* en sentido inverso buscando *nodot*
7: Adición a *familiaxv* de los nodos entre *r* y *t*
8: Recurrencia *SPLIT* en los subconos de visión del ala derecha.
9: Adición de *ii* a las familias.
10: **devolver** *familiaxv*,*familiavy*,**EVG**

Durante la búsqueda de los nodos de interés se determina la existencia de las líneas de visibilidad, dado que existen comprobaciones intermedias de si existen obstáculos o no que interrumpan la linea de visión de directa ente *ii* y el nodo que se analice.

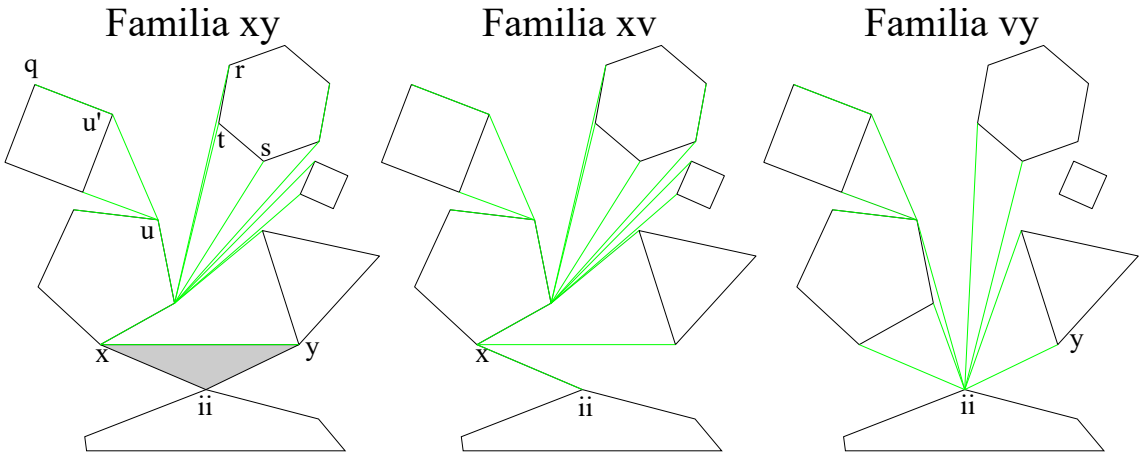


Figura 2.19 Ejemplo de aplicación de *SPLIT*.

Una vez aclarada a grandes rasgos la función *SPLIT*, pasamos dentro de la implementación a su llamada, donde se puede observar inputs y outputs esperables bajo las premisas establecidas, salvo las variables *loop* y *'externo'*. *loop* es una necesidad de diseño, dado que se espera que *SPLIT* sea recursiva, es necesario poner un número máximo de capas, para evitar tiempos de computación elevados que puedan llevar a rendimientos decrecientes en tiempo de computación. Por otro lado, *'externo'* es una variable *string* que se usa dentro del código con fines de facilitar la depuración. *'externo'* devuelve información de interés acerca de como se está realizando el procesado e *interno* evita que se muestre. En la versión del código adjunta se han eliminado las líneas asociadas a la depuración.

```
554     %% Bucle valores a SPLITear
555     if length(vectorrecorre)>1
556         TOT_famxv=[];
557         TOT_famvy=[];
558         for jj=2:length(vectorrecorre)
```

```
559
560             nodox=vectorrecorre(jj-1);
561             nodoy=vectorrecorre(jj);
562
563             %% Llamada Split
564             % clc
565             loop=0;
566
567             [EVG,familiaxv,familiavy]=SPLIT(nodox,nodoy,EVG,ii,loop,'interno');
```

Código 2.21 Llamada a la función *SPLIT* desde *vectorrecorre*.

A partir de este momento, las líneas de código que se muestren pertenecerán a la función *SPLIT* salvo que se indique lo contrario, esto podrá notarse rápidamente mirando los indicadores de línea de código. esta función se encuentra en el mismo script que $Main_v13$ que es el que gestiona el postprocesado y la triangulación, por lo que se continua usando la numeración de lineas.

```
718 function [EVG,familiaxv,familiavy]=SPLIT(nodox, nodoy, EVG,ii,loop,modo)
719 loop=loop+1;
720 maxloop=2;
721
722 %% Declaración de variables
723 %Se extrae de EVG la familia de nodos
724
725 seguimientoSPLIT=[ii, nodox,nodoy];
726
727 if condicion_embudo_mod(EVG,nodox,nodoy,ii)%SEG
728     familiaembudo=[nodox,nodox;nodoy,nodox];
729 else
730     familiaembudo=calculafamilia(nodox,nodoy,EVG,ii);
731 end
732
733 familiaxv=[];
734 familiavy=[ii,ii];
735
736 nodou=zeros(1);
737 indiceu=zeros(1);
```

Código 2.22 Función *SPLIT*: Arranque.

El arranque de la función es, a su vez, una preparación para disponer datos necesarios para el funcionamiento del algoritmo de forma estructurada. Cabe destacar los siguientes eventos:

1. Aumento de la variable de seguimiento de recursividad de bucle *loop*.
2. Establecimiento de la variable de la máxima recursividad permitida (podría haberse introducido en *EVG*, pero al ser información referente a la implementación se ha preferido mantener aparte).
3. *seguimientoSPLIT* es una variable muy útil en la depuración.
4. *condicion_embudo_mod* es un seguro para comprobar que ciertas familias sólo presenten los embudos x e y. Esto resulta de interés en los siguientes casos:
   - **a)** Los dos vértices sean consecutivos y su familia de embudos mire hacia el interior del polígono, véase el desarrollo presente en 2.2.1.
   - **b)** Realmente no sea una linea de visión. Esto no debería ocurrir pero se ha añadido esto para hacer la algoritmia *fail-safe*.
   - **c)** Si tres puntos pertenecen a un mismo polígono y este es un triangulo, esta familia debería estar vacía. Si solo hubiera polígonos convexos esta condición podría extraerse a polígonos de más lados.

5. Si se identifica que la familia puede encontrarse completa debido a que las condiciones anteriormente no se cumplen se realiza la llamada a $calculafamilia$.

6. Se plantean las variables que contendrán las familias partidas $familiaxv$ y $familiavy$. Nótese el detalle de que $familiavy$ ya tiene introducido por defecto su *nodox* correspondiente, que es el nodo degenerado *ii*.

7. Se plantean los *placeholders* de *nodou* e *indiceu*. Esto será una constante para todos los nodos de interés, para minimizar el número de llamadas a $find$, se almacenará el índice que ocupan en la $familiaembudo$ original.

```
function SOL=condicion_embudo_mod(EVG,nodox,nodoy,ii)

polnodox=EVG.nodo2poligono(nodox);
polnodoy=EVG.nodo2poligono(nodoy);
polnodoii=EVG.nodo2poligono(ii);

if (polnodox==polnodoii &&polnodoy==polnodoii)
    naristas=EVG.nodo2poligono(EVG.nodo2poligono(:)==polnodoii);
    if length(naristas)==3
        condtriang=true;
    else
        condtriang=false;
    end
else
    condtriang=false;
end

if EVG.Consec_aristas2(nodox,nodoy) || EVG.Caminos_prohibidos(nodox,nodoy)==1 || condtriang
    SOL=true;
else
    SOL=false;
end

end
```

Código 2.23 Función *condicion_embudo_mod*.

Puntualización sobre la definición de visibilidad

Una constante a lo largo de todo el documento es que no se especifica en ningún momento ningún tipo de criterio para identificar cuando dos nodos son visibles. Se podría inferir que, aprovechando las propiedades de los embudos, en especial la norma que trata acerca de que se encuentran vacíos, se pueden extraer ciertas reglas que resultan de ayuda. Si dentro del cono que forma el padre en el árbol superior y el árbol inferior se encuentra *ii*, ese punto sería por definición visible. Esto puede verse en la figura 2.20, con un ejemplo práctico.

Por ejemplo, se puede inferir, que siempre *ii* es visible respecto a *x* e *y*, esta información está doblemente confirmada gracias al algoritmo de triangulación. De manera homóloga, se puede deducir, que cualquier embudo cuyo padre en el árbol inferior sea *x* y en el superior sea *y*, también es necesariamente visible. Aplicando estas reglas y geometría euclidiana, en especial las propiedades de los productos vectoriales, detectar si un vector se encuentra dentro de un cono es trivial. Nótese que el padre en el árbol superior se identifica usando la operación sucesores antihorarios, véase 2.2.4.

Se distinguen diferentes casos para poder identificar de forma más sencilla el nodo *u*. Los nodos del ala izquierda no visibles se identifican como $'fueraconoi1'$, cualquier nodo que sea visible para el ala derecha forzará que se corte la búsqueda de *u*.

En la versión actual del código se aprovechan parcialmente estas características para descartar rápidamente la visibilidad de ciertos embudos para los siguientes casos:

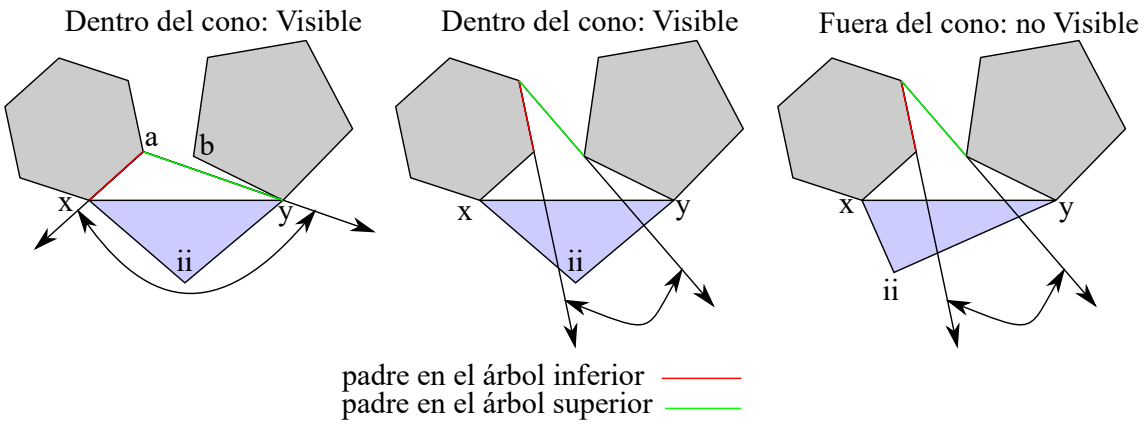


Figura 2.20 Cálculo de la visibilidad mediante el empleo de las características de los embudos.

1. Embudos que se encuentran fuera del cono de visión formado por $x - ii$ e $ii - y$.
2. Embudos cuya visibilidad está bloqueada por un tercer nodo en el ala izquierda (aprovechando el conocimiento acerca del padre en el árbol inferior).
3. Embudos cuya visibilidad está bloqueada por un tercer nodo en el ala derecha (aprovechando el conocimiento acerca del padre en el árbol superior).

En teoría, esto debería funcionar, y de forma general lo hace con resultados satisfactorios, el problema radica en que existen ciertas configuraciones en las que lo anteriormente explicado no aplica. Este hecho se ha descubierto durante las labores de prueba, que en el caso de que un embudo pudiera tener dos padres potenciales, existirían dos conos posibles, y probar ambas alternativas no sería factible dado que solo se puede identificar un padre. El ejemplo de la figura 2.21 lo muestra sin lugar a dudas, dos configuraciones diferentes de embudos dan lugar a dos rangos de visibilidad muy diferentes. Dado que en el documento original no se dan indicaciones acerca de cómo seleccionar o reorganizar embudos al cambiar de familias, se ha optado por resolver el problema por fuerza bruta y comprobar si existen cortes con las aristas de los polígonos asociados a los nodos presentes en la familia, y en última instancia a todos los nodos asociados a los polígonos que se visitan. Todo esto queda desarrollado en el Apéndice A, sección A.3.2.

Nótese además, que por como están configurados los *SPLIT*, al basarse estos en un proceso de triangulación incremental, las visibilidades en el grafo se añaden desde el punto de estudio *ii* a todos aquellos que tengan menor coordenada x. Esto se hacía de forma similar en el método de Lee, que computaba los semiplanos superiores al punto de estudio, salvo que en este caso son los semiplanos izquierdos. La visibilidad hacia valores mayores de x se computará cuando se alcancen estos puntos y se detecte su visibilidad, dado que el grafo de visibilidad es simétrico.

Cálculo de u

Dentro del documento de Ghosh & Mount[6], se especifica que u es un nodo visible desde *ii* y que precede a otro nodo, sin dar detalles de su cálculo. Con tal de trazar cierta simetría en el problema se han tomado propiedades y características de t para poder usar de forma similar sus propiedades en el árbol izquierdo.

Tampoco especifica directamente como identificar el *nodou*, y parte de la base de que ya está localizado. Con las propiedades asignadas de t, esto puede lograrse.

- Se recorre de uno en uno cada embudo de la familia, computando si son visibles para *ii* o no. Y en el caso de no serlo, se revisa la forma en la que no son visibles, gracias a *modosalida*. A partir de aquí existen dos posibilidades: que sea visible o que no.

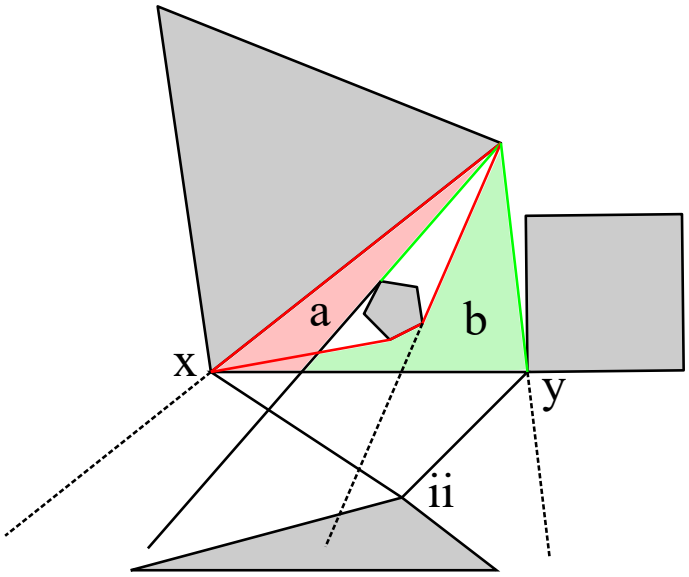


Figura 2.21 Ejemplo de fallo en la demostración geométrica de la visibilidad con las propiedades de los embudos.

- Si el embudo es visible: Si las condiciones son apropiadas, se busca aplicar *SPLIT* al embudo actual de la operación y el anterior embudo que fuera visible, se almacenan los embudos obtenidos.
- Si no es visible: En el caso de que no sea visible por $'fueraconoi'$, es decir, por un corte de visibilidad, producido por un nodo del ala izquierda, se almacena el embudo en $familiavy$.
- Se comprueba si el bucle debería parar, ya sea si es a causa de que se ha alcanzado *nodoy* o si es porque el embudo de estudio no cumple con la condición $'fueraconoi'$.

```
740 %% Calculo u
741 % u es el último nodo que puede bloquear la visibilidad de nodos que se encuentren en el ala
        izquierda de la familia
742 % 'Calculo u'
743
744
745 indiceaux=1;
746 nodoaux=familiaembudo(indiceaux,1);
747
748 EVG.Visibilidad(nodoaux,ii)=1;
749 EVG.Visibilidad(ii,nodoaux)=1;
750
751 familiaxv=[familiaxv;familiaembudo(indiceaux,:)];
752 familiavy=[familiavy;familiaembudo(indiceaux,1),ii]; %Al partir el embudo queda la parte de xv
        igual y la de vy, se convierte en un nodo cuya base es N
753
754 modosalidaextend=strings(length(familiaembudo(:,1)),1);
755 modosalidaextend(1)='VIS';
756
757 %El primer nodo est añadido por defecto
758 flag=0;
759 indiceaux=2;
```

Código 2.24 Implementación del cálculo de u: Arranque.

Antes de empezar el bucle es necesario realizar cierta preparación, se añade $x-ii$ al diagrama de visibilidad como visible y el correspondiente embudo a $familiaxv$y $familiavy$. Se define *modosalidaextend*, que guarda los *outputs* de texto del algoritmo de detección de visibilidad,

$visible_D EF$, en el vector para realizar de forma sencilla operaciones dentro del bucle, como pararlo o realizar $SPLIT$ de forma recursiva. Se define la bandera $flag = 0$ que mantendrá el bucle ejecutándose hasta alcanzar la condición de parada, ya sea alcanzar el final de la lista, o encontrar el último embudo del ala izquierda que bloquea la visibilidad de los siguientes, es decir, el *nodou*.

```
761 while flag==0
762     %Arrancamos el bucle
763     [VIS,modosalida,EVG]=visible_DEF(EVG,familiaembudo,indiceaux,ii,nodox,nodoy);
764     modosalidaextend(indiceaux)=modosalida;
765 
766     if VIS==true
767         nodoaux=familiaembudo(indiceaux,1);
768         EVG.Visibilidad(nodoaux,ii)=1;
769         EVG.Visibilidad(ii,nodoaux)=1;
770         %Si es visible se comprueba que se pueda splitear
771 
772         for jj=(indiceaux-1):-1:1
773             if strcmp('VIS',modosalidaextend(jj))
774                 nodoa=familiaembudo(jj,1);
775                 break
776             end
777         end
778 
779         nodob=familiaembudo(indiceaux,1);
780 
781         %Una vez se tiene la base de SPLIT, se comprueba que esta sea
782         %congruente
783 
784         %Se hace una necesidad técnica el comprobar que la forma del SPLIT
785         %es adecuada, nodob debe preceder a nodoa CW
786         p1=EVG.WPs(nodoa,:)';
787         p2=EVG.WPs(nodob,:)';
788         p3=EVG.WPs(ii,:)';
789         v1=p1-p3;
790         v2=p2-p3;
791         prodvec=v1(1)*v2(2)-v1(2)*v2(1);
792 
793         if prodvec<0
794             llamadaCW=true;
795         else
796             llamadaCW=false;
797         end
798 
799         % CONDVACIO
800         %Para evitar que nodos que se postcedan en la familia no estén
801         %entorpeciendo y cumplir la condición de que los nodos deban
802         %encontrarse vacíos, introducimos una condición adicional (4,6,caso1; check nodo34)
803 
804         %para ello se organizan todos los nodos alrededor de ii partiendo
805         %desde nodoy, se revisan si entre los dos nodos propuestos hay uno o
806         %más nodos
807 
808         posnodoy=EVG.Mat_ind_pos_enMatrizorden(ii,nodoy);              % Se saca la posición de
    N en la secuencia de nodoorden
809         nodoorden=EVG.Matrizorden(ii,:);                               % Se extrae el vector
    nodoorden que contiene los nodos ordenados en sentido CCW
810         nodoorden=[nodoorden(posnodoy+1:end),nodoorden(1:posnodoy)];    % Se ordena en CCW
    dejando último a N
811         nodoorden=nodoorden(end:-1:1);                              %A hora N es primero en orden
     CW
812 
813         [~, indicefiltrado]= ismember(nodoorden, familiaembudo(:,1));    % Se filtran los valores
    posibles a partir de los nodos que pertenencian a subfamxv
814         familiaord=nodoorden(indicefiltrado>0);                        % En este caso, se tienen
    todos los nodos ordenados respecto a ii
815 
816         ind1=find(familiaord==nodoa);
817         ind2=find(familiaord==nodob);
818 
819         condvacio=[];
```

```
820         if ind2-ind1>1
821             for jj=(ind1+1):(ind2-1)
822                 %Se comprueba si se encuentran DENTRO del cono
823                 nodocheck=familiaord(jj);
824                 isInside = checkIfInsideTriangle(ii, nodoa, nodob, nodocheck, EVG);
825                 if isInside==true
826                     condvacio=false;
827                     break
828                 end
829             end
830             if isempty(condvacio)
831                 condvacio=true;
832             end
833
834         else
835             condvacio=true;
836         end
837
838
839
840         if ~(nodox==nodoa && nodoy==nodob) && loop<=maxloop && llamadaCW && condvacio
841             %Para evitar recursividades, es necesario meter esta
842             %condición, tmbn es necesario integrar en familiaguia estás familias de embudos, si
      no se pierden las posibles visibilidades
843
844             [EVG,familiaxv_sub,familiavy_sub]=SPLIT(nodoa,nodob,EVG,ii,loop,'interno');
845
846             %El primer valor ya está computado en familiaxv, el ultimo es
847             %ii por lo que no se añade
848             vecaux=familiaxv_sub(2:end-1,:);
849
850             [familiaxv]=add_familiaxv(EVG,familiaembudo,familiaxv,vecaux,nodox);
851
852             vecaux=familiavy_sub(3:end,:); % el primer valor es ii, no se añade, el segundo es el
       nodoy del anterior, por lo que tampoco
853             [familiavy]=add_familiavy(EVG,familiaembudo,familiavy,vecaux,ii);
854         else
855             [familiaxv]=add_familiaxv(EVG,familiaembudo,familiaxv,familiaembudo(indiceaux,:),
      nodox);
856             [familiavy]=add_familiavy(EVG,familiaembudo,familiavy,familiaembudo(indiceaux,:),ii);
857         end
858     end
```

Código 2.25 Implementación del cálculo de *u*: *nodoaux* es visible.

Cómo se indicó en el resumen inicial, lo primero que se hace es revisar la visibilidad y almacenar el *modosalida*. Dentro del if que comprueba si la variable VIS contiene información acerca de si es visible o no, se guarda en el diagrama de visibilidad. A continuación se recorre hacia atrás los nodos ya visitados buscando el último nodo que fue visible. Ese nodo se identificará como el *nodox* de la siguiente generación, que se llamará para evitar confusiones *nodoa*, de forma análoga, el nodo de estudio que sería el *nodoy* en la recursividad se llamará *nodob*. Se adjunta la imagen 2.22, donde se puede observar como a lo largo de la búsqueda de *nodou* se van visitando diferentes nodos del ala izquierda y se realizan operaciones de $SPLIT$ acorde, esto actúa como un seguro en caso de que, por ejemplo, no se hayan guardado debidamente embudos en la familia, y pudiera resultar altamente beneficioso en el caso del $SPLIT3$ de la imagen.

Antes de ejecutar este $SPLIT$ es necesario garantizar que *nodoa* preceda a *nodob* tanto en orden de embudo, cosa que está inherente al concepto de familia, como en sentido horario pivotando desde *ii*. Esto es algo que no se desarrolla en el documento original, pero existen casos concretos en los que puedan darse dos nodos visibles atravesando túneles formados entre polígonos, que provocarían fallos. Esto además actúa como un seguro en el caso de que un embudo tuviera algún fallo en su asignación en la familia. Todo esto se recoge en la definición de la variable $llamadaCW$.

Además se comprueba si entre *nodoa* y *nodob* pudiera haber algún nodo dentro del cono, que provocaría falsos positivos, dado que se trabaja suponiendo que estos conos de visión están vacíos.

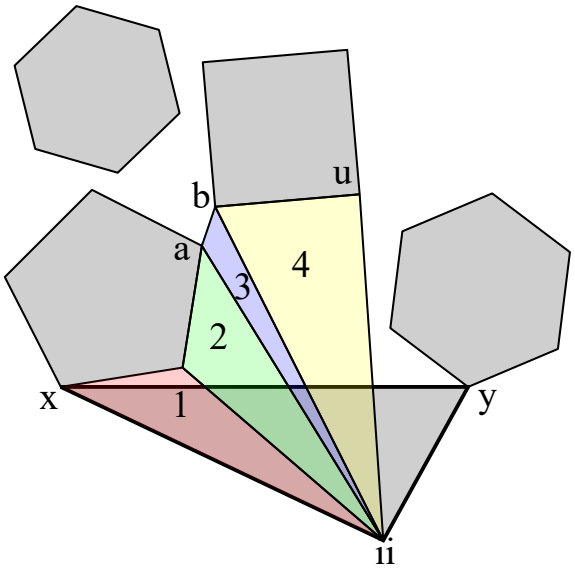


Figura 2.22 Uso de recursividad *SPLIT* para detección de embudos.

Para ello se extrae de *Matrizorden*, véase 2.2.4, se busca la secuencia de nodos pivotados en torno a *ii* a partir de *nodoy* en el sentido de las agujas del reloj, y se verifica que entre los dos no existan nodos, y de existir, se comprueba si quedan dentro del cono de visión gracias a la función *checkIfInsideTriangle*. Esta lo que hace es una operación de cálculo de áreas y comprueba como se distinguen entre estas áreas entre sí, si el total de las secciones suma igual o menor que el cono, el punto se encuentra dentro. Una explicación esquemática puede encontrarse en 2.23. Si ningún nodo se encuentra dentro, no se ha recurrido más allá del número permitido máximo de bucles *maxloop* y no se está intentando partir el mismo *SPLIT* que el actual, se procede a ejercer la recursividad.

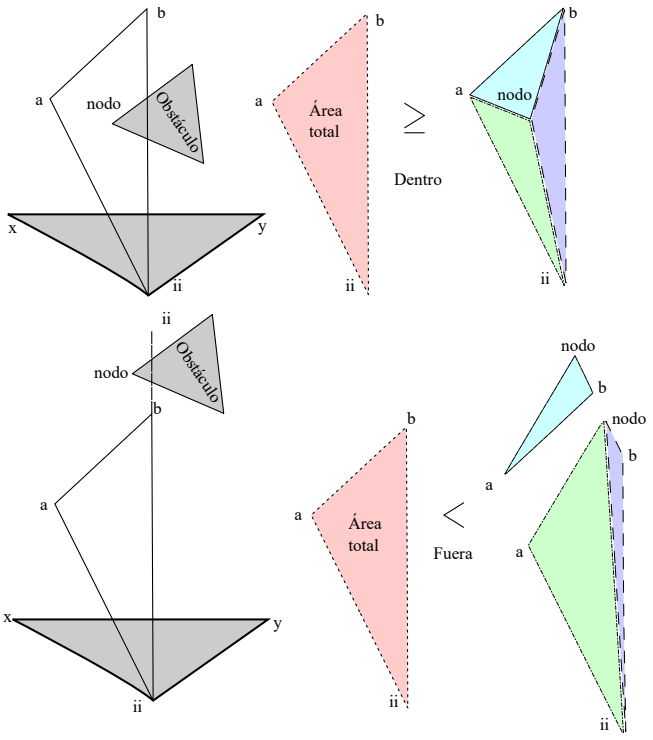


Figura 2.23 Uso de *checkIfInsideTriangle* para comprobar la posición relativa de un nodo externo.

Si se llega a realizar la llamada a *SPLIT* se extrae la familia partida, como se esperaba, y esas familias se añaden a $familiaxv$ y a $familiavy$ usando las funciones análogas $add\_familiaxv$ y $add\_familiavy$. En el código 2.26, se puede identificar una estructura sencilla consistente en identificar nodos repetidos en *vecaux* que estén a su vez en $familiaxv$, a partir de ahí los nodos no repetidos se añaden. Se organizarán fuera de la llamada de *SPLIT* principal. Las funciones para $familiaxv$ y $familiavy$ son completamente iguales por lo que por compacidad solo se añade una.

En el caso de que no se cumplan todas las condiciones que permitan hacer la llamada a *SPLIT* solo se añaden los nodos *a* y *b* a las correspondientes familias usando las ya mencionadas funciones.

```
1  function [familiaxv]=add_familiaxv(EVG,familiaembudo,familiaxv,vecaux,nodox)
2
3  % Obtener los índices de los elementos de vecaux(:,1) que NO están en familiaxv(:,1)
4  [ismem, ~] = ismember(vecaux(:,1), familiaxv(:,1));
5
6  % Invertir el valor lógico para obtener los índices NO repetidos
7  indicesnorep = find(~ismem);  % Aquí mantenemos el orden de vecaux
8
9  if ~isempty(indicesnorep)
10     % Mantener solo los elementos de vecaux que no están en familiaxv
11     vecaux = vecaux(indicesnorep, :);
12 else
13     vecaux = [];
14 end
15
16 % Mantener solo los valores únicos en familiaxv respetando el orden original
17
18 % 'Inicio'
19 if ~isempty(vecaux)
20     familiaxv=[familiaxv;vecaux];
21 end
22
23 end
```

Código 2.26 Implementación de $add\_familiaxv$.

La última parte del código presenta una comprobación del modo de salida, para ubicar correctamente el nodo en cuestión en la correspondiente familia. En el caso $'fueraconoi1'$, es necesario añadir el embudo a la familia *vy*, si no fue visible para *ii* por un obstáculo que interrumpía su visión por la izquierda, tampoco lo será para la base *xv*. Esto queda reflejado en la figura 2.24, dónde se puede observar la estrategia de determinación de visibilidad mostrada en la figura 2.20, quedando constancia visual de lo anteriormente explicado. Es por ello que hace una revisión rápida del caso y se añade si procede.

En el caso de que el modo de salida fuera otro, no sería necesario añadirlo ya que en etapas posteriores esto será contemplado, por lo que no se perderá esta información.

Para decidir si continuar con el bucle o no se hace de nuevo una distinción de casos:

1. Nodo de estudio visible o $'fueraconoi1'$: Se revisa si se ha alcanzado el último embudo de la familia, si es el caso se corta el bucle. Si no, se incrementa *indiceaux* y se continua con el escrutinio. Se levanta una bandera llamada $resueltafamilia$, para indicarle al resto del código si es necesario seguir aplicando la metodología implementada.
2. Nodo de estudio no visible en casos diferentes a $'fueraconoi1'$: El bucle se corta, el último nodo registrado como visible se identifica como *nodou*. La familia no está entera resuelta, por lo que es necesario continuar el procesado.

```
860      if strcmp('fueraconoi1',modosalida)
861          [familiavy]=add_familiavy(EVG,familiaembudo,familiavy,familiaembudo(indiceaux,:),ii);
862      end
863
```

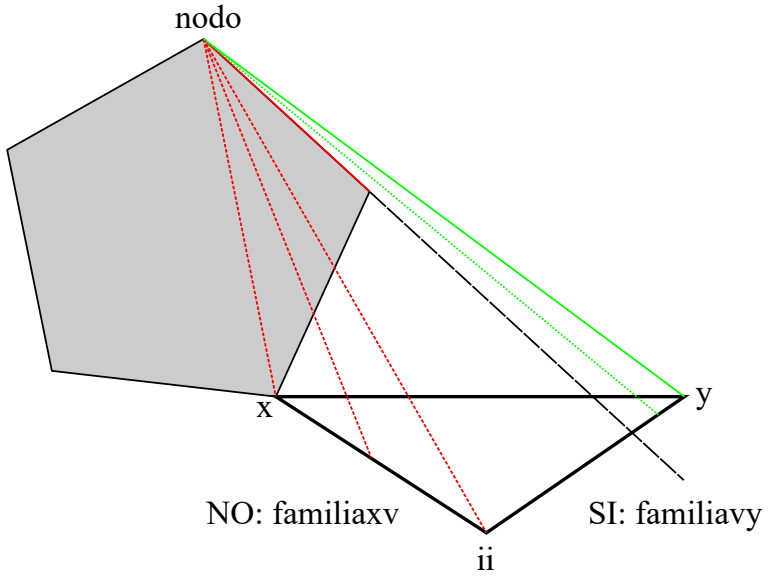


Figura 2.24 Justificación de añadir a $familiavy$ pero no a $familiaxv$: Caso $'fueraconoi1'$.

```
864
865     if strcmp('VIS',modosalida) || strcmp('fueraconoi1',modosalida)
866         %Cualquiera de esos 2 casos continua corriendo el bucle en
867         %búsqueda de $u
868         if indiceaux<length(familiaembudo(:,1))
869             indiceaux=indiceaux+1;
870             nodoaux=familiaembudo(indiceaux,1);
871         else
872             flag=1; %Con esto lo que se hace es salir del bucle una vez se han computado todos
      los nodos
873             indicesol=indiceaux;
874         end
875     else
876         flag=1;%se rompe el bucle
877         for jj=(indiceaux-1):-1:1
878             if strcmp(modosalidaextend(jj),'VIS')
879                 indicesol=jj;
880                 break
881             end
882         end
883     end
884 end
885
886 nodou=familiaembudo(indicesol,1);
887 indiceu=indicesol;
888
889 resueltafamilia=false;
890
891 if indiceu==length(familiaembudo(:,2)) % este es el caso particular de que se han computado en el
       calculo de u todos los embudos
892     resueltafamilia=true;
893 end
```

Código 2.27 Implementación del cálculo de u: $nodoaux$ no visible y cierre.

Calculo de u'

Una vez identificado u, si la familia no está resuelta, el código se bifurca gracias a una sentencia if en base a la variable $resueltafamilia$. El objetivo de esta parte del código es identificar u', el hijo de u ubicado más en el sentido de las agujas del reloj cuya visibilidad de ii está interrumpida por u, para que sirva como ayuda para localizar r, el primer nodo en orden de embudo del ala derecha,

siendo el más lejano de esta ala. Tampoco se proporcionan indicaciones en el artículo original acerca de su cálculo pero gracias a la claridad de la definición su localización es sencilla.

En la primeras primeras líneas se observa una vez más la preparación para el arranque del bucle, y el inicio de la sentencia *if* que realiza la derivación conforme a la naturaleza de *familiaresuelta*. Se recorren desde *nodoy* hasta *nodou* todos los nodos, aumentando *contadorindice* y restándolo al índice auxiliar que realiza el seguimiento de la posición en el bucle *indicecandidatouprima*. Si el padre es *nodou*, comprobación trivial si se comprueba la segunda componente del vector fila correspondiente que guarda el embudo; se comprueba si la componente *qz* resultante del cálculo de $\vec{v}_1 \times \vec{v}_2$, donde $\vec{v}_1$ es el vector de origen *ii* y destino *nodou*, mientras que $\vec{v}_2$ comparte origen pero su destino es el nodo candidato a ser *uprima*, es mayor o igual a 0. En el caso de ser mayor o igual a 0, *nodou* se encontraría más adelantado en el sentido de las agujas del reloj, por lo que bloquearía su visión, y por ende, dado que se está recorriendo la familia de fin a principio, se obtendría el primer nodo cuyo padre es *nodou* y su línea de visión con *ii* se ve interrumpida por este, figura 2.25.

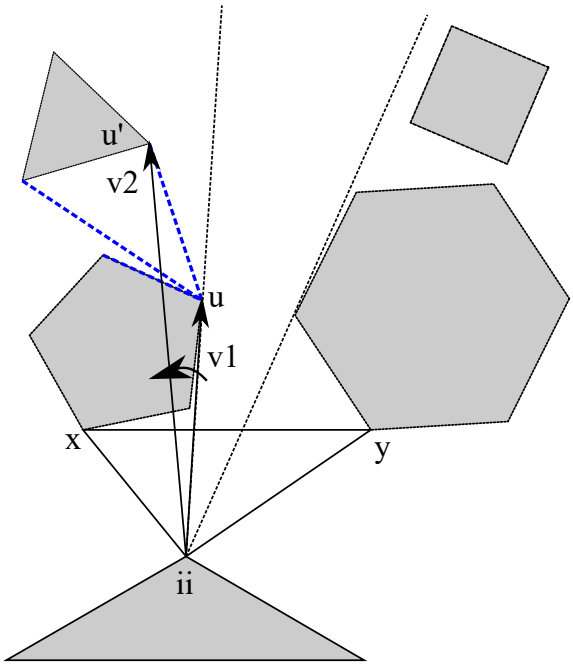


Figura 2.25 Ejemplo de identificación *uprima*.

```
895 %% Cálculo u'
896 % 'Cálculo uprima'
897 % uprima es el hijo de u más clockwise (por definición de u, no es visible uprima desde v), puede
        no existir
898 % Se recorrerá usando un bucle que vaya desde y hasta u, si no encuentra un
899 % solo hijo de u entonces saldremos del bucle
900
901 contadorindice=0;
902
903 indicecandidatouprima=(length(familiaembudo(:,2))-contadorindice);
904 nodouprima=NaN;
905 indiceuprima=NaN;
906
907 if resueltafamilia==false
908
909
910     while indicecandidatouprima>indiceu
911         % Solo realizamos este bucle cuando u sea diferente de y
912         if familiaembudo(indicecandidatouprima,2)==nodou
913             p1=EVG.WPs(familiaembudo(indiceu,1),:);
914             p2=EVG.WPs(familiaembudo(indicecandidatouprima,1),:);
```

```
915             p3=EVG.WPs(ii,:);
916
917             v1=p1-p3;
918             v2=p2-p3;
919
920             prodvec=v1(1)*v2(2)-v1(2)*v2(1);
921             if prodvec>0
922                 %El padre del candidato nodouprima es u por lo que no es candidato, es
923                 %el nodo
924                 indiceuprima=indicecandidatouprima;
925                 nodouprima=familiaembudo(indiceuprima,1);
926                 break
927             end
928         end
929         contadorindice=contadorindice+1;
930         indicecandidatouprima=(length(familiaembudo(:,2))-contadorindice);
931     end
```

Código 2.28 Cálculo de u'.

Cálculo de q y r

La identificación de estos nodos si se encuentra documentada en el documento original de Ghosh & Mount, [6], y para ello es necesario distinguir dos casos:

1. *nodouprima* no existe: No hay entonces en la familia ningún embudo cuya visibilidad es bloqueada por *nodou*, lo que implica que escondido por *nodou* para *ii* no hay ningún nodo, por lo que la definición de q (el último nodo del ala izquierda) queda invalidada, y se realiza la siguiente asignación $nodoq = nodou$. El siguiente nodo entonces en orden de embudo es *nodor* (el último nodo del ala derecha).
2. *nodouprima* existe: Que será el caso que se explicará con más detalle a continuación, dado que es necesario aplicar las propiedades de la familia de embudos para encontrarla.

En el caso de existir *uprima*, hay que considerar que este mismo embudo, todos sus hermanos *CCW*, es decir, los hijos de *nodou* que se encuentran posicionados más en el sentido de las agujas del reloj respecto a *nodouprima*, y los descendientes de todos los nodos anteriormente citados, debido a la convexidad de las familias de embudos (regla de la *goma elástica*, véase sección 2.2.7), no son visibles para *ii* por la presencia de *u*. El embudo posterior al último descendiente de *uprima*, se encontrará en el ala derecha y será *nodor*, y que será hijo o bien de *nodou* o bien de cualquiera de sus ascendientes en el árbol superior.

Es por ello que se usará la siguiente estrategia:

1. Se parte de *uprima* y se busca si tiene algún hermano posterior a este en la familia de embudos. Si lo tiene, ese hermano se identifica como *nodor* y el nodo que precede al mismo se identifica como *nodoq*.
2. Si *uprima* no tiene hermanos *CW*, se busca si su ancestro directo tiene hermanos entre *nodouprima* e *y*. En caso afirmativo se para el bucle y se realiza la identificación ya mencionada de *nodor* y *nodoq*.
3. En caso contrario, se repite la búsqueda entre los antecesores hasta llegar a *nodox*, el embudo primigenio.

Se adjunta para dos casos diferentes ejemplos del procedimiento, figura 2.26, en el caso A) se puede observar que *nodouprima* si tiene un hermano *CW*, por lo que la identificación de *nodor*, y consecuentemente la de *nodoq* es directa. Mientras que en el caso B), es necesario remontar hacia el padre de *u* para encontrar un nodo válido.

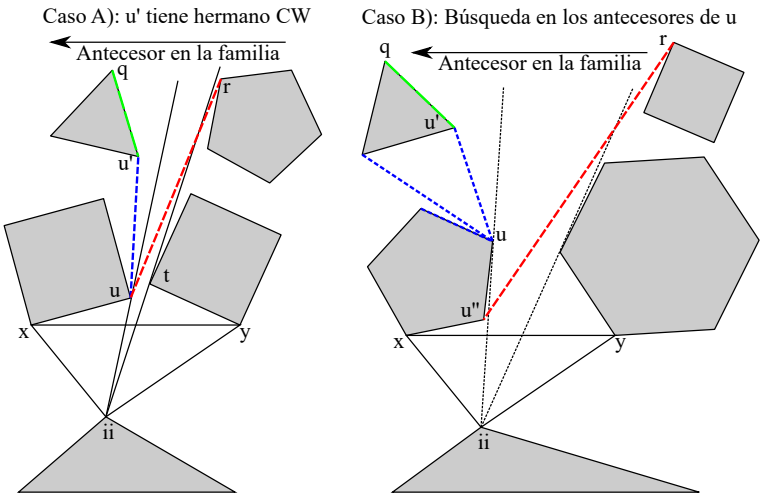


Figura 2.26 Ejemplo de identificación *nodor* y *nodoq*.

En el desarrollo teórico del documento original, [6], se plantea la deducción de estas propiedades a partir de lo que los autores denominan el reloj de arena,que consiste en la forma que presentarían los nodos de la familia si se representaran para una familia dada $x-u-q-r-t-y$, siempre y cuando *nodoq* y *nodou* sean embudos diferenciados. Tal y como se presenta la familia de embudos y sus propiedades, existe un cuello estrecho vacío de nodos conformado por *nodou* y *nodot*; de forma adicional justifica que *nodoq* y *nodor* sean consecutivos en orden de nodo. Esto de forma teórica es muy interesante, pero en la práctica ciertas configuraciones pudieran no ser reducibles a estos parámetros, especialmente cuando existen pasillos estrechos después de haber procesado muchos polígonos, dándose varios relojes de arena. Se presentan los relojes de arena asociados a las familias propuestas para el cálculo de *nodor* en la figura 2.27.

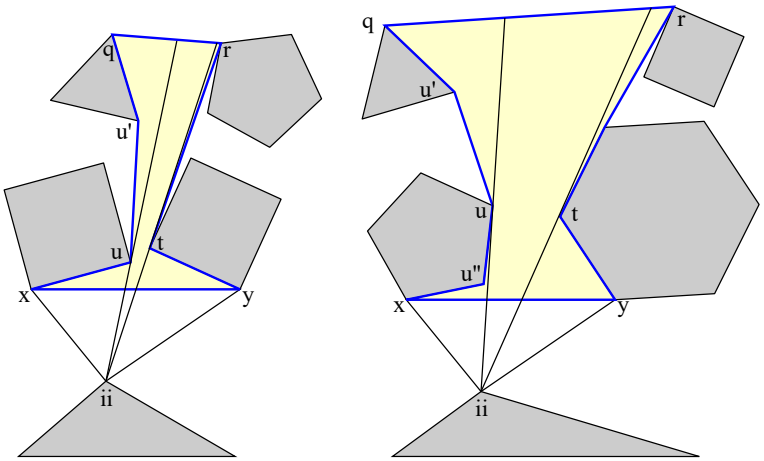


Figura 2.27 Identificación de relojes de arena para los casos de 2.26.

Nótese que al principio del código se considera el caso que el *nodouprima* no existe, por lo que el código genera dos ramificaciones que se vuelven a unir al final de la sección, con ambos valores de *nodor* y *nodoq* ya identificados según el caso.

```
933     %% Calculo q y r + Adición nodos solo visibles para (v,y)
934     %Se irá desde u hasta x viendo si hay hermanos
935     % 'Calculo q y r + Adición nodos solo visibles para (v,y)'
936     nodoq=NaN;
937     nodor=NaN;
```

```
938
939     indiceq=NaN;
940     indicer=NaN;
941
942
943     if isnan(nodouprima) || nodouprima==nodou || nodouprima==nodox
944         %Si uprima no existe, entonces u es una hoja, ello implica lo siguiente
945         % 'Caso1'
946         nodoq=nodou;
947         indiceq=indiceu;
948         indicer=indiceu+1;
949         nodor=familiaembudo(indicer,1);
950     else
951         % 'Caso2'
952         %Si uprima existe hay que buscar el nodo r
953         %Para ello partiremos del padre de u y veremos si tiene hermanos,
954         %si no tiene, iremos hasta  el siguiente ancestro de u y veremos si
955         %tiene hermanos hasta llegar a la raiz, como u~=y debería haber un
956         %hermano
957
958         flag=0;
959
960         nodopadre=familiaembudo(indiceuprima,2);%Nodo padre, de uprima
961         indicepadre=find(familiaembudo(indiceu:-1:1,1)==nodopadre,1);% Probar con sentido
     creciente para ver la variación de rendimiento
962         indicepadre=length(familiaembudo(indiceu:-1:1,1))-indicepadre+1;
963
964         nodohermano=NaN(1);
965         indicehermano=NaN(1);
966         contador=0;
967         while flag==0
968             %Se busca el siguiente nodo en orden de embudo, posterior a u, que
969             %sea hermano de u, es decir, que compartan padre
970
971             indicehermano=find(familiaembudo((indiceuprima+1):end,2)==nodopadre,1);%Hay que
     meterle una corrección qe va después del isempty
972             if isempty(indicehermano)
973                 % 'Dentroempty'
974                 %Si no tiene hermano comprobamos si el ancestro anterior tiene más hijos después
     de u en orden de embudo
975                 nodopadre=familiaembudo(indicepadre,2);
976                 indicepadre=find(familiaembudo(indiceu:-1:1,1)==nodopadre,1);
977                 indicepadre=length(familiaembudo(indiceu:-1:1,1))+1-indicepadre;
978             else
979                 indicehermano=indicehermano+length(1:indiceuprima);
980                 %Si ha encontrado un hermano más clockwise que uprima, el siguiente nodo a uprima
      en orden de embudo será r
981                 % nodohermano=familiaembudo(indicehermano,1)
982
983                 if indicehermano>indiceuprima
984                     indicer=indicehermano;
985                     nodor=familiaembudo(indicer,1);
986                     %q es el predecesor de r
987                     indiceq=indicer-1;
988                     nodoq=familiaembudo(indiceq,1);
989                 else
990                     indicer=indiceuprima+1;
991                     nodor=familiaembudo(indicer,1);
992                     indiceq=indiceuprima;
993                     nodoq=familiaembudo(indiceq,1);
994                 end
995                 flag=1;
996
997             end
998             contador=contador+1;
999             if contador>10
1000                familiaembudo
1001                seguimientoSPLIT
1002                nodouprima
1003                nodou
1004                error('Bucle infinito: Revisar geometría_nodoy')
```

```
            end
        end

        %Todos los nodos entre u+1 y q son solo visibles desde v,y, por lo que
        %se añaden a familiavy

        [familiavy]=add_familiavy(EVG,familiaembudo,familiavy,familiaembudo(indiceu+1:indiceq,:),
    ii);

        %EVG.Visibilidad No hay que añadirlo dado que estos puntos no son
        %visibles desde ii
    end
```

Código 2.29 Cálculo de q y r.

Como ocurría con los nodos no visibles en $'fueraconoi1'$ durante el cálculo de *nodou*, figura 2.24, todos los embudos desde $nodou+1$ hasta *nodoq* están ocultos para *ii* y por ende para la base *xv*, por lo que se añaden por separado a $familiavy$.

Con esto concluye el procesado de los nodos del ala izquierda.

Cálculo de t

Una vez identificados los nodos de interés del ala izquierda, se buscarán los nodos de interés del ala derecha. En este caso, el método propone una búsqueda algo más elaborada de t, consistente en partir de s, nodo posterior a t del ala derecha, que a su vez es el padre de q en el árbol superior, recorrer a través del árbol superior de s hasta y, usando extensiones CCW, y desde y se recorrerían los nodos del ala derecha ubicados más en el sentido de las agujas del reloj hasta llegar a t. Este nodo es el homólogo de *nodou* en el ala derecha de la familia, bloquea la visión de todos los nodos restantes del ala derecha, hasta llegar a *nodor*.

Como están planteadas las familias de embudos en esta implementación, no genera una ventaja la utilización del *nodos*, por lo que no se ha considerado su uso. Se recorrerán todos los nodos desde y hasta r, guardando la información acerca de los ángulos y apuntando cada vez que haya un nodo con mayor ángulo de los que se hayan registrado anteriormente, estos conformarán el camino a $SPLIT$ más adelante. Esto cumpliría la función de las extensiones de una manera algo menos eficiente. La función sucesores centrada en *ii* podría técnicamente identificar a t, si se hiciera sucesor CW con pivote en t partiendo desde u, pero no tendría información acerca de los puntos a visitar.

El bucle comienza recorriendo desde y hasta r todos los embudos existentes en sentido contrario al orden de embudo. Una vez identificado el nodo a partir de su índice, se revisa su visibilidad y el ángulo que forma respecto al *nodoy* tomando *ii* de pivote. Para evitar problemas con la medida del ángulo se emplea el producto vectorial para evitar las ambigüedades. Toda esta información se registra en *vectorangulo* que guarda la siguiente información:

1. columna: *nodoaux*, que es el nodo de trabajo en cuestión.
2. columna: *angulo*, el ángulo en radianes.
3. columna: VIS,la variable que recoge si el nodo es visible.
4. columna: 0, es una variable bandera, con al finalidad de marcar si un embudo presenta un ángulo mayor al anteriormente propuesto, en el caso de que esto sea así se levantará.
5. columna: *indiceaux*, el índice del nodo de trabajo.

```
    %% Recorriendo desde y hasta r
    indices=length(familiaembudo(:,1));
    nodos=familiaembudo(indices,1);

    %% Calculo de t
```

```
1022      % Suposición: los nodos entre y y s son todos visibles desde v (podríamos comprobarlo por có
          digo de todos)
1023      tkind=(indices+1):length(familiaembudo(:,1));
1024      indiceaux=indices;
1025
1026      vectorangulo=[];
1027      maxangulo=[];
1028
1029      while indicer<=indiceaux
1030          nodoaux=familiaembudo(indiceaux);
1031          [VIS,~,EVG]=visible_DEF(EVG,familiaembudo,indiceaux,ii,nodox,nodoy);
1032
1033          if VIS==true
1034              EVG.Visibilidad(ii,nodoaux)=1;
1035              EVG.Visibilidad(nodoaux,ii)=1;
1036          end
1037
1038          %Se calcula el ángulo respecto a vy (el mayor será el que sea t)
1039          p1=EVG.WPs(nodoaux,:);%nodo a revisar visibilidad
1040          p2=EVG.WPs(nodoy,:);%Nodoy
1041          p3=EVG.WPs(ii,:);
1042
1043          v1=p1-p3;
1044          v2=p2-p3;
1045
1046          angulo = calcularAngulo(v1, v2);
1047          % NOTA: si el ángulo es positivo. v1 rota hacia v2 en sentido CCW.
1048          %si el ángulo es negativo, v1 rota hacia v2 en sentido CW
1049          %es por ello, que va a hacer falta usar el menos ángulo para filtrar
1050          %por valor de máximo valor de ángulo = t
1051
1052          angulo=-angulo;
1053
1054          vectorangulo=[nodoaux,angulo,VIS,0,indiceaux;vectorangulo];
```

Código 2.30 Cálculo de t: Parte 1.

A continuación si el nodo en cuestión fue visible, y el ángulo es igual a mayor al último registrado se guarda en *vectorangulo* y se levanta la bandera para la fila asociada a ese nodo.

Una vez todos los nodos han sido computados, *nodot* puede identificarse como el nodo con el mayor ángulo presentado, que será el que bloquee la visibilidad de los que se encuentren por detrás en el ala derecha, hasta *nodor*. Una vez identificado el nodo se busca su índice. Todos los nodos entre *nodor* y $nodot-1$ son visibles solo para la base de $familiaxv$, de la misma manera que los nodos entre $nodou+1$ y *nodoq*, solo eran visibles para la base de $familiavy$, véase figura 2.24.

El uso de *maxangulo* es necesario, dado que este seguimiento es el que permite identificar que nodos son aquellos que están disponibles de cara a realizarles un *SPLIT*, de forma análoga a como se hacía en el ala izquierda durante la búsqueda de *nodou*. Por ejemplo, se ve en la figura 2.28, entre a y b hay nodos intermedios que no conforman una base válida para un *SPLIT*, estos son $b-1$, $b-2$ y $b-3$. Con esta manera de proceder pueden identificarse de forma sencilla con la información computada en *vectorangulo*.

Se añaden también a $familiavy$ aquellos nodos que se hayan detectado como visibles pero no hayan dado como positivo por *maxangulo*, de esa manera se evita que falten embudos en las familias.

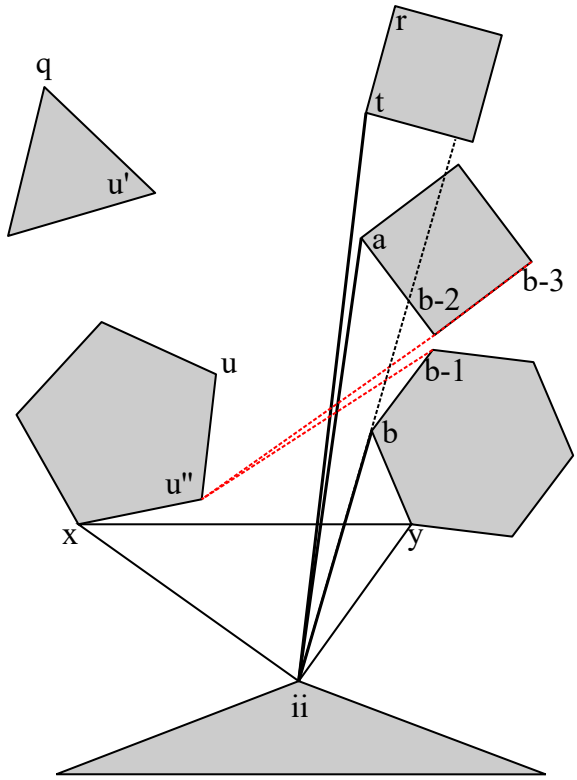


Figura 2.28 Relevancia del uso de *maxangulo*.

```
1056          if ~isempty(maxangulo)
1057              if angulo>maxangulo && VIS==true
1058                  maxangulo=angulo;
1059                  vectorangulo(1,4)=1;
1060              end
1061          else
1062              maxangulo=angulo;
1063              vectorangulo(1,4)=1;
1064          end
1065          indiceaux=indiceaux-1;
1066      end
1067      nodot=vectorangulo(vectorangulo(:,2)==maxangulo,1);
1068
1069      locindicet=find(vectorangulo(:,1)==nodot);
1070      indicet=find(familiaembudo(:,1)==nodot);
1071
1072      [familiaxv]=add_familiaxv(EVG,familiaembudo,familiaxv,familiaembudo(indicer:(indicet-1),:),
          nodox);
1073
1074      nodosVIShastat=vectorangulo(vectorangulo(:,3)==true,1);
1075
1076      [familiavy]=add_familiavy(EVG,familiaembudo,familiavy,nodosVIShastat,ii);
1077
1078      tkindrec=vectorangulo(vectorangulo(:,4)==1,5);
```

Código 2.31 Cálculo de t: Parte 2.

Vector t_k

Con la cadena de puntos ya definida, solo es necesario recorrerla de forma análoga a como se hizo con *vectorrecorre*, véase la sección 2.3.2, pero realizando las comprobaciones previas que se tomaron para los *SPLIT* de *nodou*, véase la sección 2.3.3. En el método original solo se plantea recurrencia de *SPLIT* de forma explicita en el ala derecha, aunque esto podría haberse dado por

entendido al mencionar que todos los nodos entre x y u debían estar previamente resueltos y añadidos al grafo de visibilidad extendido.

La cadena de nodos t_k, representada por un vector de embudos organizados en orden de embudo, se encuentra dentro de la variable $tkindrec$, que contiene los índices de los embudos en $familiaembudo$. La cadena t_k, tiene la propiedad adicional de ser el árbol superior de t, tal y como se ha planteado su identificación.

Como se ha mencionado anteriormente, esta sección del cálculo es análoga a la presentada en el cálculo del *nodou*, solo cambia ligeramente la estrategia de inserción de nodos en la familia, que se muestra en la segunda parte del código 2.33. El curso de acción una vez más es el siguiente:

1. A diferencia de para *nodou*, se tienen todos los nodos a los que se les debe realizar *SPLIT* de antemano. Por lo que es necesario identificar las parejas de puntos a visitar de forma externa. La estrategia de identificar a x e y en la recurrencia es la misma, renombrándose en la capa superior de código como a y b.
2. Se aplican también las condiciones $llamadaCW$ y $condvacio$, pero no la condición para evitar que se recurra a si mismo, es decir $x = a \& y = b$, no puede darse en el ala derecha, dado que ninguno de los nodos posteriores o iguales a t puede ser x.
3. En vez de extraer introducir los valores directamente en el *else* de las condiciones, donde se introducía el nodo auxiliar, se generan las familias procesadas para introducirlas en las familias.

```
1080    %% Bucle Tk (Recursividad)
1081    % tkindrec
1082    %Se recorren por parejas los conjuntos de familias de nodos
1083    auxind=NaN(1);
1084    auxlength=NaN(1);
1085
1086    if length(tkindrec)>=2
1087        for kk=2:length(tkindrec)
1088
1089            %Se hace una necesidad técnica el comprobar que la forma del SPLIT
1090            %es adecuada, nodob debe preceder a nodoa CW
1091            nodoa=familiaembudo(tkindrec(kk-1),1);
1092            nodob=familiaembudo(tkindrec(kk),1);
1093
1094            p1=EVG.WPs(nodoa,:)';
1095            p2=EVG.WPs(nodob,:)';
1096            p3=EVG.WPs(ii,:)';
1097            v1=p1-p3;
1098            v2=p2-p3;
1099            prodvec=v1(1)*v2(2)-v1(2)*v2(1);
1100
1101            if prodvec<0
1102                llamadaCW=true;
1103            else
1104                llamadaCW=false;
1105            end
1106
1107            % CONDVACIO
1108            %Para evitar que nodos que se postcedan en la familia no estén
1109            %entorpeciendo y cumplir la condición de qu elos nodos deban
1110            %encontrarse vacíos, introducimos una condición adicional (4,6,caso1; check nodo34)
1111
1112            %para ello organizamos todos los nodos alrededor de ii partiendo
1113            %desde nodoy, revisamos si entre los dos nodos propuestos hay uno o
1114            %más nodos
1115
1116            posnodoy=EVG.Mat_ind_pos_enMatrizorden(ii,nodoy);              % Se saca la posición
      de N en la secuencia de nodoorden
1117
1118            nodoorden=EVG.Matrizorden(ii,:);                           % Se extrae el vector
     nodoorden que contiene los nodos ordenados en sentido CCW
```

```
1119            nodoorden=[nodoorden(posnodoy+1:end),nodoorden(1:posnodoy)];        % Se ordena en
    CCW dejando último a N
1120            nodoorden=nodoorden(end:-1:1);                                  %A hora N es primero en
    orden CW
1121
1122            [~, indicefiltrado]= ismember(nodoorden, familiaembudo(:,1));    % Se filtran los
    valores posibles a partir de los nodos que pertenencian a subfamxv
1123            familiaord=nodoorden(indicefiltrado>0);                          % En este caso, se
    tienen todos los nodos ordenados respecto a ii
1124            % Atención, esto quitaría los fueraconoi, pero técnicamente se
1125            % recuperarían en la recursividad
1126
1127            ind1=find(familiaord==nodoa);
1128            ind2=find(familiaord==nodob);
1129
1130            condvacio=[];
1131            if ind2-ind1>1
1132                for jj=(ind1+1):(ind2-1)
1133                    %Se comprueban si se encuentran DENTRO del cono
1134                    nodocheck=familiaord(jj);
1135                    isInside = checkIfInsideTriangle(ii, nodoa, nodob, nodocheck, EVG);
1136                    if isInside==true
1137                        condvacio=false;
1138                        break
1139                    end
1140                end
1141
1142                if isempty(condvacio)
1143                    condvacio=true;
1144                end
1145
1146            else
1147                condvacio=true;
1148            end
1149
1150            if loop<=maxloop && llamadaCW && condvacio
1151                [EVG,familiaxv_sub,familiavy_sub]=SPLIT(familiaembudo(tkindrec(kk-1),1),
    familiaembudo(tkindrec(kk),1),EVG,ii,loop,'interno');
1152            else
1153                familiaxv_sub=[familiaembudo(tkindrec(kk-1),1),familiaembudo(tkindrec(kk-1),1);
    familiaembudo(tkindrec(kk),1),familiaembudo(tkindrec(kk-1),1);ii,familiaembudo(tkindrec(kk-1)
    ,1)];
1154                familiavy_sub=[ii,ii;familiaembudo(tkindrec(kk-1),1),ii;familiaembudo(tkindrec(kk
    ),1),ii];
1155            end
```

Código 2.32 Recurrencia t_k : *Parte*1.

De forma similar a como se añadían los valores en el cálculo de *nodou*, pero con un diferente enfoque; cada ejecución de ese cálculo añadía todos nodos a la familia, de forma que, si la ejecución paraba en la siguiente iteración porque se detectaba un nodo no visible de manera no $'fueraconoi'$, pudiera parar directamente, aquí el enfoque es distinto. Todas las ejecuciones se solapan, por lo que se añaden una vez se conocen las familias, todos los nodos obtenidos entre a y $b-1$, dado que b será añadido como el a de la siguiente iteración de indices de *tkindrec*, o bien a la salida del bucle.

Se ha cubierto el caso de que pudiera existir un único nodo en *tkindrec*, eso se daría en el caso de que y actuara a su vez como t. En ese caso ese único nodo se añade a ambas familias.

```
1157            %Se añaden valores a familiaxv y familiavy
1158            indiceaux=tkindrec(kk-1);
1159            nodoaux=familiaembudo(indiceaux,1);
1160
1161            %Los embudos extraidos de iteraciones de SPLIT deben ir
1162            %acumulandose
1163
1164            EVG.Visibilidad(nodoaux,ii)=1;
1165            EVG.Visibilidad(ii,nodoaux)=1;
1166
```

```
1167            %Se busca el primer nodo de familiaxv_sub que se encuentre
1168            %en familia xv, -2 porque los últimos valores son nodoy e ii.
1169
1170            vecaux=familiaxv_sub(1:end-2,:);
1171
1172            %Este sigue el nodo propuesto como padre
1173
1174            [familiaxv]=add_familiaxv(EVG,familiaembudo,familiaxv,vecaux,nodox);
1175
1176            %El primer nodo es ii, el último es nodob, se añadirá a la salida del
1177            %bucle si es la última ejecución, o como nodoa en la siguiente
1178            vecaux=familiavy_sub(2:end-1,:);
1179
1180            [familiavy]=add_familiavy(EVG,familiaembudo,familiavy,vecaux,ii);
1181
1182            indiceaux=tkindrec(kk);
1183            nodoaux=familiaembudo(indiceaux,1);
1184
1185            EVG.Visibilidad(nodoaux,ii)=1;
1186            EVG.Visibilidad(ii,nodoaux)=1;
1187
1188            if kk==length(tkindrec)
1189                if familiaembudo(indiceaux,1)==familiaembudo(indiceaux,2)
1190                    familiaxv=[familiaxv;familiaembudo(indiceaux,1),familiaxv(1,1)];
1191                    familiavy=[familiavy;familiaembudo(indiceaux,1),ii];
1192                else
1193                    [familiaxv]=add_familiaxv(EVG,familiaembudo,familiaxv,familiaembudo(indiceaux
    ,:),nodox);
1194                    [familiavy]=add_familiavy(EVG,familiaembudo,familiavy,familiaembudo(indiceaux
    ,:),ii);
1195                end
1196            end
1197            %Fin bucle jj
1198        end
1199    else
1200
1201        indiceaux=tkindrec(1);
1202        nodoaux=familiaembudo(indiceaux,1);
1203
1204        EVG.Visibilidad(nodoaux,ii)=1;
1205        EVG.Visibilidad(ii,nodoaux)=1;
1206
1207        % El único punto que hay, es necesario añadirlo
1208
1209        if familiaembudo(indiceaux,1)==familiaembudo(indiceaux,2)
1210            familiaxv=[familiaxv;familiaembudo(indiceaux,1),familiaxv(1,1)];
1211            familiavy=[familiavy;familiaembudo(indiceaux,1),ii];
1212        else
1213            [familiaxv]=add_familiaxv(EVG,familiaembudo,familiaxv,familiaembudo(indiceaux,:),
    nodox);
1214            [familiavy]=add_familiavy(EVG,familiaembudo,familiavy,familiaembudo(indiceaux,:),ii);
1215        end
1216    end
```

Código 2.33 Recurrencia t_k : *Parte*2.

Últimas comprobaciones

Esta sección actúa como cajón de sastre, guardando ciertas acciones que se realizan en el código para garantizar ciertos estándares en la salida.

La primera de estas acciones es comprobar que se estén conservando todas las familias que había en *familiaembudo*, en el caso de que la concurrencia no hubiera podido tener lugar. Para ello se revisan para las parejas de nodos de *tkindrec*, todos los nodos intermedios en *vectorangulo*, y se añaden con el padre que tenían asociado a *familiaembudo*.

```
1218    %% Se añaden nodos 0 entre t+1 e y
1219
```

```
1220      nodoscheckrecorrido=familiaembudo(tkindrec,1);
1221
1222      %Si por la razón que fuera no se hubieran añadido todos los nodos, se
1223      %buscan entre las parejas de tkindrec, los embudos que no estén y se
1224      %añaden. Se filtrarán y ordenarán fuera del bucle.
1225
1226      for jj=1:length(nodoscheckrecorrido)-1
1227          nodoseg=nodoscheckrecorrido(jj);
1228          nodosegpost=nodoscheckrecorrido(jj+1);
1229
1230          indnodoseg_vectorangulo=find(vectorangulo(:,1)==nodoseg);
1231          indnodosegpost_vectorangulo=find(vectorangulo(:,1)==nodosegpost);
1232
1233          vectorvalores_no1=vectorangulo((indnodoseg_vectorangulo+1):(indnodosegpost_vectorangulo
       -1),1);
1234
1235          if ~isempty(vectorvalores_no1)
1236              for kk=1:length(vectorvalores_no1)
1237                  nodo1=vectorvalores_no1(kk);
1238                  padrefamiliaembudo=familiaembudo(familiaembudo(:,1)==nodo1,2);
1239                  vectorvalores_no1(kk,2)=padrefamiliaembudo;
1240              end
1241              indnodosegpost_famx=find(familiaxv(:,1)==nodosegpost,1);
1242              familiaxv=[familiaxv(1:(indnodosegpost_famx-1),:);vectorvalores_no1;familiaxv(
       indnodosegpost_famx:end,:)];
1243          end
1244      end
1245      %Fin del if de familia resuelta
1246  end
```

Código 2.34 Manteniendo consistencia de familias.

Es necesario hacer notar que esto podría romper el orden de embudo de las familias, que antes podría haberse mantenido considerando la metodología de introducción, aunque con padres no actualizados, pero eso ya no sucede así. De todas maneras, no vale la pena realizar un esfuerzo adicional de implementación en esta parte del bucle debido a que una vez terminadas las instancias más externas de *SPLIT* se realizará un refactorizado completo, en el caso de que se hayan añadido.

Desde el final de la última instancia de código presentada, se vuelven a considerar ahora aquellas ejecuciones en los que durante el cálculo de u, se computaron todos los nodos de la familia, es decir, pasó el end del if de $resuelta familia == true$.

En la siguiente instancia de código se usará el comando *unique* en modo $'stable'$ para eliminar embudos que pudieran estar duplicados, tanto para $familiaxv$ como para $familiavy$. De forma adicional, se revisa si *nodoy* está en la última posición de $familiavy$ dado que si no lo estuviera el algoritmo de reordenación de familias no funcionaría.

```
1248  %% Reasignación familiaxv,vy
1249
1250  [~,indiceunico]=unique(familiaxv(:,1),'stable');
1251  familiaxv=familiaxv(indiceunico,:);
1252
1253  [~,indiceunico]=unique(familiavy(:,1),'stable');
1254  familiavy=familiavy(indiceunico,:);
1255
1256
1257  indy=find(familiavy(:,1)==nodoy,1);
1258  if indy~=length(familiavy(:,1))
1259      familiavy(indy,:)=[];
1260      familiavy=[familiavy;nodoy,ii];
1261  end
```

Código 2.35 Limpieza de nodos duplicados.

Para cerrar la función es necesario añadir a $familiaxv$ el nodo ii, que dado que la base de $familiaxv$ es la conformada por $x - ii$, sería el último de la familia. Se añade con su padre siendo $nodox$ y con esto concluye la ejecución de *SPLIT*.

```
%% Completando familiaxv y familiavy
%Hay que añadir ii a familia xv y ii se añadió a familiavy al principio de
%la ejecución de SPLIT

familiaxv=[familiaxv;ii,nodox];
%FIN SPLIT
end
```

Código 2.36 Fin de SPLIT.

2.3.4 Procesado de las familias

Una vez se han dividido las familias y acumulados todos los nodos visibles para las bases xv e vy, se recuerda que $v \equiv ii$, se realiza un proceso de almacenamiento en dos matrices llamadas TOT_famxv y TOT_famvy que guardan la información de las familias de par en par de columnas, una para los nodos hijos y otra para los padres. El número de pares de columnas es igual a las parejas de valores consecutivos que se encuentran en *vectorrecorre*, que conforman las diferentes bases de estas familias divididas, es decir, la longitud del mencionado *vectorrecorre* menos uno.

Para poder introducir estas familias se comprueba si el número de filas de la susodicha es menor, igual o mayor a la que tiene la matriz. En caso de que sea menor, se amplia el número de filas utilizando $zeros(l,2)$ y si es mayor se amplia el resto de la matriz con matrices de ceros para igualar. Estas familias se almacenan en una estructura de datos almacenada en el grafo de visibilidad extendido.

```
            [EVG,familiaxv,familiavy]=SPLIT(nodox,nodoy,EVG,ii,loop,'interno');

            if jj==2
                TOT_famxv=familiaxv;
                TOT_famvy=familiavy;
            else
                if length(familiaxv(:,1))>=length(TOT_famxv(:,1))
                    TOT_famxv=[[TOT_famxv;zeros(-length(TOT_famxv(:,1))+length(familiaxv(:,1)),
    length(TOT_famxv(1,:)))],familiaxv]   ;
                else
                    TOT_famxv=[TOT_famxv,[familiaxv;zeros(+length(TOT_famxv(:,1))-length(
    familiaxv(:,1)),2)]];
                end

                if length(familiavy(:,1))>=length(TOT_famvy(:,1))
                    TOT_famvy=[[TOT_famvy;zeros(-length(TOT_famvy(:,1))+length(familiavy(:,1)),
    length(TOT_famvy(1,:)))],familiavy]   ;
                else
                    TOT_famvy=[TOT_famvy,[familiavy;zeros(+length(TOT_famvy(:,1))-length(
    familiavy(:,1)),2)]];
                end
            end
        end
        EVG.TOT_famxv_glob{ii}=TOT_famxv;
        EVG.TOT_famvy_glob{ii}=TOT_famvy;
```

Código 2.37 Almacenaje de los resultados obtenidos.

En versiones anteriores del código sólo se trabajaba con las familias que resultaban de forma directa del *SPLIT*, lo que implicaba que solo las familias a las que se podían acceder era a las conformadas por $x_{jj} - v_{jj}$ y $v_{jj} - y_{jj}$, siendo jj el índice que recorre las parejas de *vectorrecorre*.

En geometrías sencillas funcionaba bien, pero al aumentar el número de obstáculos y la complejidad de los mismos faltaba información por lo que el grafo quedaba incompleto.

Procedimiento: Longitud de *vectorrecorre* es inferior a 2

Cuando el *vectorrecorre* solo tiene un valor, no se ha podido partir ninguna familia, por lo que se crean dos familias triviales, es decir, que solo contienen los correspondientes *nodox* y *nodoy*. Por otro lado se añade la línea de visión entre el valor presente de *vectorrecorre*, que se le llamará a por comodidad para representar a continuación la familia de embudos, e ii.

Las familias que se añaden al diagrama de visibilidad extendida son las siguientes:

$$familia_{a,ii} = \begin{bmatrix} a & a \\ ii & a \end{bmatrix}, \quad familiavy = \begin{bmatrix} ii & ii \\ a & ii \end{bmatrix} \tag{2.8}$$

Cabe destacar que una de las familias apunta hacia dentro del polígono que comparten, y la otra hacia el lado externo. Pero ambas sólo contienen a su base, dado que las familias guardan información acerca de lo que se dirige hacia los valores de coordenadas x menores.

```
588     elseif length(vectorrecorre)==1
589
590         EVG.TOT_famxv_glob{ii}=[vectorrecorre,vectorrecorre;ii,vectorrecorre];
591         EVG.TOT_famvy_glob{ii}=[ii,ii;vectorrecorre,ii];
592
593         EVG.Visibilidad(ii,vectorrecorre)=1;
594         EVG.Visibilidad(vectorrecorre,ii)=1;
```

Código 2.38 Caso *vectorrecorre* contiene solo un valor.

En el caso de que *vectorrecorre* esté vacío, situación que se da en recintos cóncavos, dado que en estos existe la posibilidad de que para tres puntos consecutivos del perímetro de un polígono, los extremos tengan mayor coordenada x que el de en medio. No es necesario añadir nada al grafo de visibilidad dado que no existe visibilidad ninguna de cara a valores con menor coordenada x, por lo que se añade un embudo trivial.

```
595     else
596         %Es necesario introducir esta información, esta debe ser
597         %visible para ser consultada. No conectar con ningún valor es un caso
598         %que puede darse, y se da, en recintos cóncavos
599         EVG.TOT_famvy_glob{ii}=[ii,ii];
600         EVG.TOT_famxv_glob{ii}=[ii,ii];
601
602     end
```

Código 2.39 Caso *vectorrecorre* no contiene ningún valor.

Adición de los nodos asociados a las familias al diagrama de visibilidad extendido

Para poder computar las familias empleando $add\_familia2$, es necesario identificar todos los nodos asociados a las familias de tipo xv y de tipo vy obtenidas para ii.

Para ello se emplean las estructuras de datos $EVG.nodosTOT\_famxv\_glob\{ii\}$ y $EVG.nodosTOT\_famvy\_glob\{ii\}$, a las que se le añade el vector columna con los nodos únicos asociados a las familias, según si tienen orientación xv o vy.

```
604     %De esta manera se evita el cálculo repetido de estos valores
605     EVG.nodosTOT_famvy_glob{ii}=extraenodos(EVG.TOT_famvy_glob{ii});
606     EVG.nodosTOT_famxv_glob{ii}=extraenodos(EVG.TOT_famxv_glob{ii});
607
608 end
```

Código 2.40 Extracción de los nodos de las familias asociadas a la base completa.

Por su parte *extraenodos* es una subrutina que agrupa todos los nodos hijos asociados a la ejecución, que se encuentran en las columnas impares de $EVG.TOT\_famvy\_glob\{ii\}$ y $EVG.TOT\_famxv\_glob\{ii\}$, y se les aplica el comando *unique*, dado que al integrar diferentes familias podrían darse duplicados.

```
function nodos_TOT=extraenodos(TOT_fam)
if length(TOT_fam(1,:))==2
    %Esta información ya ha pasado por un unique
    nodos_TOT=[TOT_fam(:,1)];
else
    [~,d]=size(TOT_fam);
    nodos_TOT=[];

    for jj=1:d/2%Siempre será par, por lo que no hay problema
        subTOT=TOT_fam(:,2*jj-1);
        indcero=find(subTOT==0,1);

        if ~isempty(indcero)
            subTOT=subTOT(1:indcero-1);
        end
        nodos_TOT=[nodos_TOT;subTOT];
    end
    [~,indiceunico]=unique(nodos_TOT(:,1),'stable');
    nodos_TOT=nodos_TOT(indiceunico,:);
end
%Fin Bucle ii
end
```

Código 2.41 Función extraenodos.

Dependiendo de las preferencias del usuario, pueden comentarse o descomentarse las dos siguientes llamadas a figuras. Siendo la primera la que se emplea para pintar el diagrama al completo, y la segunda la que muestra de forma escalonada para cada nodo, como conecta con los anteriores, hasta llegar a N.

```
%% Postprocesado + Representación resultados RES
Descomentar si se quiere pintar la gráfica de el grafo de visibilidad al
completo
%%%%%%%%%%%%%%%%%%
fig=figure(2)
hold on
mapshow(V(:,1),V(:,2),'LineWidth',1)
for i = 1:size(WPs, 1)
    % Identificador basado en el orden
    identificador = num2str(i);
    % Mostrar el identificador junto al punto en el gráfico
    text(WPs(i, 1), WPs(i, 2), identificador, ...
        'VerticalAlignment', 'bottom', 'HorizontalAlignment', 'right', ...
        'FontSize', 6, 'Color', 'r');
end

% Personalizar el gráfico
xlim([min(V(:,1)),max(V(:,1))])
ylim([min(V(:,2)),max(V(:,2))])
ylabel('Latitud');
xlabel('Longitud');
title('Diagrama de Visibilidad');
grid on;
axis equal;

% for i = 5:N-4
for i = 1:N
    % for j = i+1:N-4
    for j = i+1:N
    % for j = i+1:271

        if EVG.Visibilidad(i, j) == 1
```

```
643             plot([WPs(i, 1), WPs(j, 1)], [WPs(i, 2), WPs(j, 2)], 'k--');
644         end
645     end
646
647 end
648
649 set(fig, 'WindowState', 'maximized');
650
651 Descomentar si se quiere usar el modo manual para comprobar punto a punto
652 grafo de visibilidad.
653
654 %%%%%%%%%%%%%%%%%%%%
655 figure(2)
656 hold on
657 mapshow(V(:,1),V(:,2),'LineWidth',1)
658
659 for i = 1:size(WPs, 1)
660     % Identificador basado en el orden
661     identificador = num2str(i);
662     % Mostrar el identificador junto al punto en el gráfico
663     text(WPs(i, 1), WPs(i, 2), identificador, ...
664         'VerticalAlignment', 'bottom', 'HorizontalAlignment', 'right', ...
665         'FontSize', 8, 'Color', 'r');
666 end
667 % Personalizar el gráfico
668 xlim([min(V(:,1)), max(V(:,1))])
669 ylim([min(V(:,2)), max(V(:,2))])
670 ylabel('Latitud');
671 xlabel('Longitud');
672 title('Diagrama de Visibilidad');
673 grid on;
674 axis equal;
675
676 % Dibujar las aristas de los polígonos primero
677 hPolygons = mapshow(V(:,1), V(:,2), 'LineWidth', 1);
678
679 % Array para almacenar los handles de las líneas dibujadas
680 visible_lines = [];
681 invisible_lines = [];
682
683 for i = 1:N
684     for j = 1:i
685         if i ~= j  % Evitar dibujar líneas de un punto a sí mismo
686             if EVG.Visibilidad(i, j) == 1
687                 % Dibujar la línea visible y almacenar el handle
688                 visible_lines(end+1) = plot([WPs(i, 1), WPs(j, 1)], [WPs(i, 2), WPs(j, 2)], 'k--',
      'LineWidth', 1.5);
689             else
690                 % Dibujar la línea no visible y almacenar el handle
691                 invisible_lines(end+1) = plot([WPs(i, 1), WPs(j, 1)], [WPs(i, 2), WPs(j, 2)], 'r
     --', 'LineWidth', 1.5);
692             end
693         end
694     end
695
696     % Pausar
697     pause;
698
699     % Eliminar todas las líneas dibujadas en esta iteración
700     if ~isempty(visible_lines)
701         delete(visible_lines);
702         visible_lines = []; % Reiniciar el array de handles
703     end
704
705     if ~isempty(invisible_lines)
706         delete(invisible_lines);
707         invisible_lines = []; % Reiniciar el array de handles
708     end
709 end
```

Código 2.42 Final *Main_v*13.

Con esto finaliza el bucle en *ii*, el grafo de visibilidad queda completado y solo queda extraerlo del diagrama de visibilidad extendido y pasarlo como salida de la función, junto con el tiempo computado tras el *toc*.

```
711 t2=toc;
712 % Guardar la visibilidad final
713 Visibilidad = EVG.Visibilidad;
```

Código 2.43 Final *Main_v*13.

Esta es la versión final implementada del código, la teoría que la sustenta y las diferencias presentes con el método originalmente planteado. En el siguiente capítulo se procederá a mostrar los resultados obtenidos con el mismo y el rendimiento que proporciona.

3 Estudio de resultados

Una vez desarrollado el algoritmo implementado, se va a proceder a ponerlo a prueba con otros métodos y comprobar empíricamente la complejidad algorítmica real.
Con este propósito se proponen los siguientes puntos de análisis:

1. Comparativa con el método de Lee: Para casos de tormentas reales, se procederá a trazar una comparativa para probar los resultados ofrecidos por la implementación.
2. Análisis de recintos complejos: Simulaciones en diferentes entornos repetibles, tiempo necesario para ello y prueba empírica de la complejidad algorítmica del método.
3. *Profile analysis*: Se comprobará para un caso concreto de ejecución en que operaciones se ha invertido más tiempo, propuestas de desarrollo y limitaciones.

3.1 Comprobación del caso de estudio: Método de Lee en tormenta realista

Debido a las diferencias en el planteamiento de ambos problemas, el que resuelve *SPLIT* y el método de Lee, resulta delicado medir ambos programas en igualdad de condiciones. Por un lado el método de Lee trabaja con un destino y un origen en un entorno abierto, mientras que el método *SPLIT* necesita un recinto cerrado pero no tiene la necesidad de plantear un origen y un destino, pero por otro lado no permite que dos puntos compartan coordenada x y no permite la existencia de 3 puntos colineares.

Esto se ha tenido en mente desde el principio del proyecto, procurando hacer que ambos algoritmos puedan trabajar, si bien no con los mismos *inputs*, con cierto preprocesado. Gracias a esto pueden trazarse comparativas aproximadas entre ambos métodos. Para ello se tomará el caso de estudio utilizado en el análisis de resultados desarrollado por Narciso Valverde, [5], se explicarán qué cambios se han tenido que realizar para poder realizar esta adaptación y se mostrarán los resultados obtenidos, comprobando que el número de conexiones del grafo es igual para ambos casos.

Para poder ejecutar un mapa de coordenadas preparado para ser usado con el método de Lee se plantea la siguiente metodología:

1. Se verifica que los polígonos estén dados en sentido contrario a las agujas del reloj en torno a su centroide, en caso contrario se reordenan los nodos asociados a los polígonos en la matriz de coordenadas.
2. Se identifican los valores menores y mayores para las coordenadas x e y. Si hay algún punto fuera del cuadrante x positivo e y positivo se desplazan todos los puntos y después se aplica un escalado.

3. Finalmente, tras esta normalización, se añaden dos triángulos auxiliares: uno con un vértice en el punto de origen y otro con un vértice en el punto de destino. En este aspecto, el método *SPLIT* es más inflexible.

El caso contrario, pasar de mapa preparado para ser usado con el método *SPLIT* a método de Lee, es bastante más sencillo, consistiendo en eliminar el recinto exterior e introducir dos coordenadas, origen y destino, al final de la matriz de coordenadas.

Se muestran a continuación los resultados obtenidos de ejecutar el mapa de estudio del trabajo de Narciso Valverde, [5], con el método *SPLIT*, el grafo de visibilidad obtenido es el mismo una vez se descartan las lineas de visión asociadas al recinto exterior y a los puntos que no son de interés de los triángulos auxiliares de destino y origen.

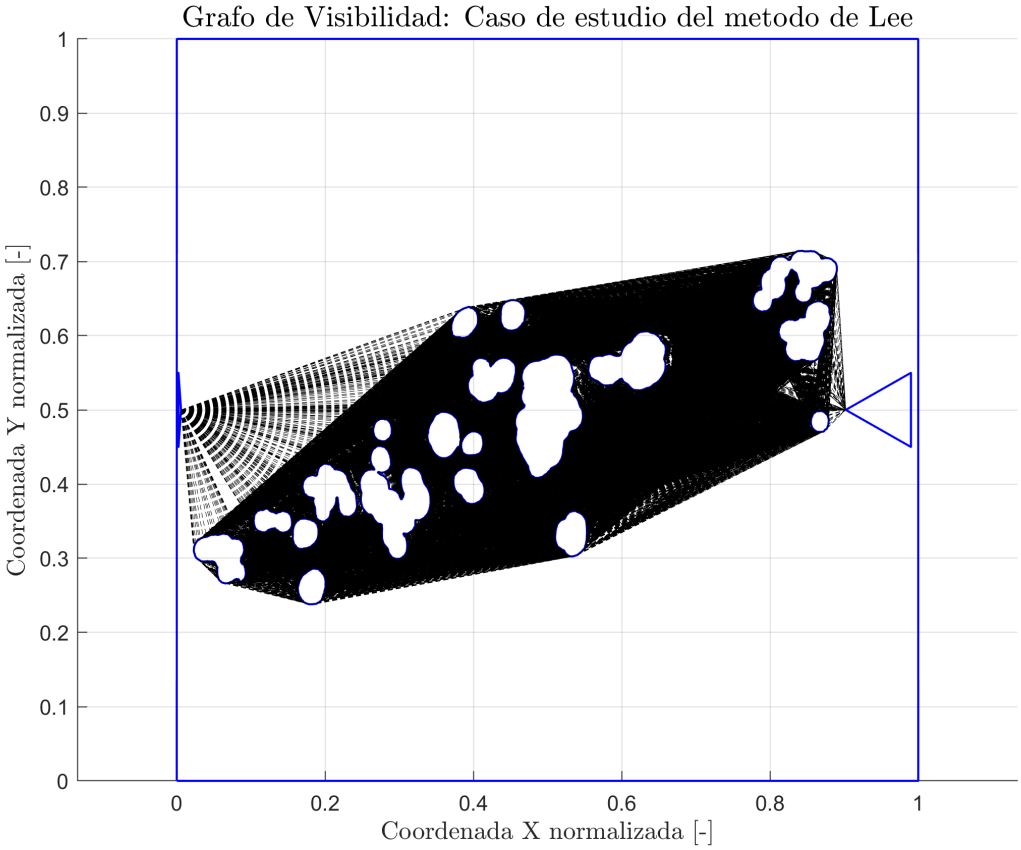


Figura 3.1 Ejemplo de región delimitada por meteorología adversa.

El tiempo de ejecución de esta simulación es de $t = 210{,}57\,s$ en el modo de ejecución de *SPLIT* adaptado a este tipo de geometrías, presentando un rendimiento menor al del método de Lee, véase la tabla 3.1. Por otro lado el número de puntos de paso de la simulación asciende a $N = 1206$, 8 puntos más que para el método de Lee debido a la necesidad técnica de añadir el recinto exterior, 4 puntos, y los nodos auxiliares no utilizados de los triángulos, otros 4 puntos.

Cabe destacar que la metodología empleada en el trabajo de Narciso Valverde [5] fue diferente: partiendo de un caso específico, se realizaron varias simulaciones para calcular el tiempo promedio de ejecución en distintas versiones reducidas del mismo caso, utilizando el algoritmo de Ramer-Douglas-Peucker. Este algoritmo permite reducir el número de puntos de los polígonos de un recinto dado, manteniendo aproximadamente sus geometrías para las tolerancias establecidas. Siguiendo esta metodología, Narciso Valverde, en su estudio [5], aplicó diferentes reducciones en el número

de nodos del recinto mostrado en esta sección, registrando los resultados correspondientes para cada valor de N generado por la simplificación.

Comparando estos valores con las ejecuciones presentes en [5], mostrados en la tabla 3.1, los tiempos son superiores a los obtenidos mediante el método de Lee pero inferiores a los obtenidos mediante el método Ingenuo avanzado que se planteo en su investigación. Estos, además, se encuentran en orden de magnitud más cercanos a los obtenidos mediante el método Ingenuo, debido a las limitaciones inherentes de la implementación.

Tabla 3.1 Comparación de tiempos de ejecución: Algoritmo basado en el método Ingenuo, método *SPLIT* y el método de Lee. Obtenida de [5].

N (-)	**Promedio de $t_{Ing,v2}$ (s)**	**Promedio de t_{SPLIT} (s)**	**Promedio de t_{Lee} (s)**
1198	454.5	210.57	22.7

Se puede observar en la imagen 3.1 que *SPLIT* respeta de forma adecuada las conexiones del grafo de visibilidad en los puntos ubicados en las concavidades de los obstáculos, lo cual se considera un objetivo cumplido. Además las envolventes se encuentran correctamente computadas y las zonas de alta densidad de grafos corresponden con las zonas con mayor densidad de puntos expuestos a otros obstáculos.

Más allá de esta comprobación cualitativa, se han comparado los grafos de visibilidad propuestos por los método Ingenuo y de Lee, arrojando el mismo número de conexiones, validando así el modelo.

3.2 Ejecución en casos propuestos: Recintos convexos

Durante el transcurso de la implementación se han establecido procedimientos y herramientas para poder depurar la algoritmia, todos estos estarán en sus respectivas secciones del Apéndice A. Entre las herramientas más útiles generadas, de cara a encontrar fallos de forma eficaz, es lo que se ha llamado un generador de recintos determinista.

Para un mismo número de puntos, la ejecución puede variar mucho dependiendo de diversos factores, como la posición relativa entre los obstáculos y el tamaño de los mismos. Es por ello que, buscando estas situaciones límite, se han creado dos generadores de recintos, uno de entornos complejos y otro de entornos sencillos, con los que se ha ejecutado el algoritmo en busca de fallos para poder depurarlos.

Se ha tomado la decisión de emplear recintos convexos y no cóncavos debido a que, para un mismo número de puntos de paso, los recintos cóncavos requieren un menor esfuerzo computacional. Esto es causado por la naturaleza dependiente de las conexiones en el grafo. Dado que se conecta un punto con otros que sean visibles, los recintos cóncavos tienen regiones en las que sólo pueden ver su entorno, lo cual reduce significativamente el esfuerzo respecto a un caso convexo, en el que la mayor parte de los nodos se encuentran expuestos a muchos otros polígonos, siempre y cuando no haya obstáculos muy voluminosos.

La matemática detrás del generador de recintos es sencilla, para poder generar entornos repetibles y comprobar de manera sencilla que las soluciones programadas tienen un impacto positivo en la implementación, se entrega una semilla *seed* para que todo lo generado aleatoriamente, usando los comandos *rand*, sea repetible consistentemente. Para poder definir la complejidad del recinto, se ha tomado la solución de indicar el número de polígonos que se necesitan al programa. De esta manera cada polígono se añade sobre una base ya computada, lo que evita que un entorno con una semilla concreta no varíe sustancialmente al querer añadir un nuevo punto.

Dado que el valor de las coordenadas de los puntos no tiene efecto en la disposición relativa de los mismos, se ha optado por normalizar las coordenadas haciendo que los recintos exteriores tengan forma de trapezoide isósceles, y sus coordenadas se ubicarán entre $x \in [0,100]\, y \in [0,100]$. Se ha considerado esa normalización entre 0 y 1 debido a que si las coordenadas están muy cercanas relativamente, la diferencia entre sus valores sería muy pequeña y se ha preferido evitar posibles problemas de cancelación. En todo caso, sólo se usan las coordenadas, que no las posiciones relativas, para el cálculo de productos vectoriales y ángulos con el fin de realizar ordenaciones, por lo que en el ámbito de evaluación con la premisa planteada, no ha existido problema alguno.

El código en detalle será recogido en el Apéndice A, pero su funcionamiento general es el siguiente:

1. Se define el recinto externo como un trapezoide siguiendo el método de definición de puntos empleado en la nomenclatura V, es decir, con valor inicial repetido y ordenados en sentido contrario a las agujas del reloj respecto a su centroide.

2. Se define un máximo número de aristas por polígonos, se ha optado al final fijarlo en 6. Esto facilitaba la depuración debido a que encontrar visualmente líneas ausentes y dibujar a mano las familias con el fin de encontrar que fallaba era más sencillo y claro visualmente. En todo caso, este número puede aumentarse dado que las soluciones implementadas son escalables. Asimismo se añaden unos márgenes respecto a las aristas que componen el trapecio exterior, para evitar intersecciones no deseadas.

3. Se realiza un barrido para cada valor entero de $npol$. En este barrido se generan aleatoriamente la posición del centroide del polígono y un radio. Se comprueba si se respeta una distancia de seguridad definida tanto con los margenes como con el resto de polígonos ya registrados. De no ser así, una nueva combinación de centroide y radio es propuesto.

4. Una vez definido centroide y radio se desplaza el origen de coordenadas al centroide, se traza una línea radial que al cortar con la circunferencia circunscrita definirá el primer vértice. Usando de forma iterativa para número de lados propuesto una matriz de rotación de ángulo $\frac{pi}{\text{nlados}}$ sobre el vértice anteriormente computado, se obtienen todas las coordenadas asociadas al polígono. Una vez cerrado el polígono, se reubican sus coordenadas al origen de coordenadas global y se añaden al vector de coordenadas no procesado V.

5. Estos polígonos podrían hacerse de forma sencilla cóncavos manipulando los radios mientras se produce el giro.

El proceso explicado anteriormente se muestra, para mayor claridad visual, en la figura 3.2, dónde se puede observar como se intenta añadir un nuevo polígono sobre un dominio en el que ya existe uno.

Estas son las bases de *calcula_recinto*, que es el generador de recintos complejos, y *calcula_recinto2*, que es el de recintos sencillos. Se ha realizado esta distinción debido a que cuando el algoritmo estaba en proceso de maduración, fallaba en casos sencillos, con el propósito de facilitar el depurado en etapas tempranas se programó una versión que generaba recintos con polígonos grandes y ubicados en el centro. El caso de recinto complejo contempla además la posibilidad de que existan obstáculos con un tamaño muy inferior al del resto, como se puede observar en la figura 3.3, con el objetivo de ver como interactuaban entre ellos con el algoritmo.

Para validar el método, aparte de la comparación de resultados con otros ya implementados, se han realizado representaciones del grafo de visibilidad para cada punto de paso, revisando que todas las líneas de visión existan, como se hace en la figura 3.4.

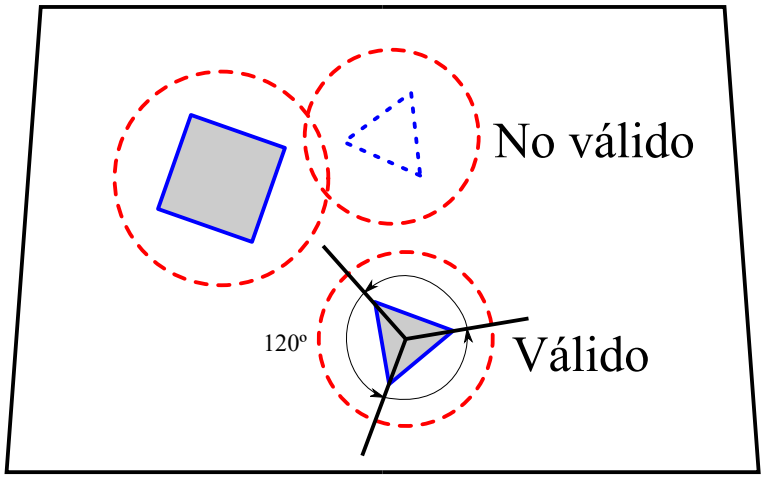


Figura 3.2 Ejemplo esquemático de creación de recintos.

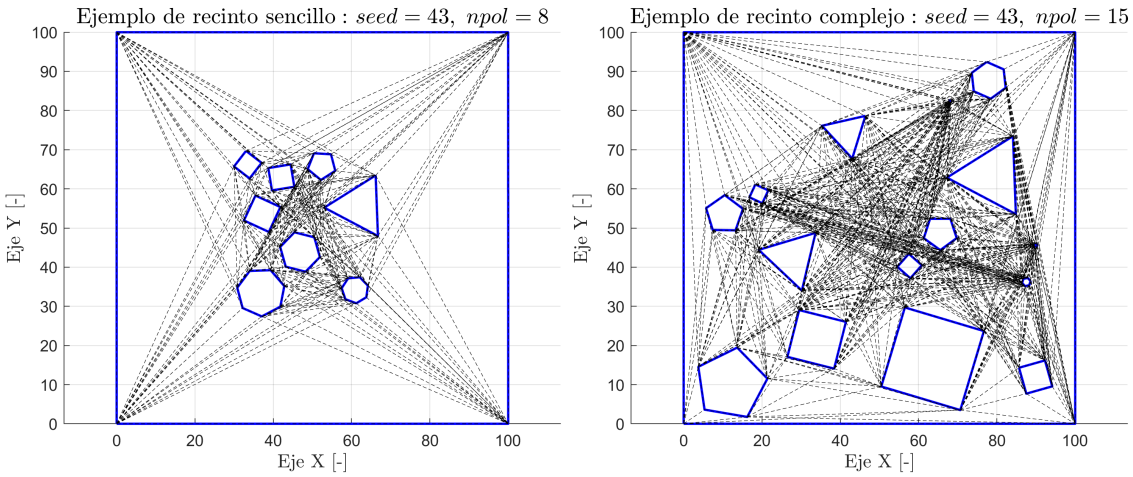


Figura 3.3 Ejemplos de recintos complejos y sencillos.

3.3 Resultados de la ejecución

Una vez planteado como se crean los recintos, es posible establecer una metodología de barrido para poder comprobar empíricamente cuál es la complejidad algorítmica del método. Para ello se plantea un barrido en *seed* y en *npol*, se revisarán 50 semillas, suficientes para tener una muestra significativa, y se realizará un barrido usando *logspace*, que irá desde 10 polígonos, que conforman aproximadamente unos 50 puntos de paso, hasta 265 polígonos, que conformarían aproximadamente 1200 puntos. Este barrido se ha realizado para los dos métodos planteados en el trabajo de Narciso Valverde, [5], y para el método *SPLIT*.

Para acelerar el cálculo de estos casos se ha hecho uso del módulo de *Parallel Computing*, ejecutando varias semillas a la vez, representando cada semilla una ejecución propia del método *SPLIT*, Lee o Ingenuo, permitiendo realizar el estudio paramétrico en márgenes de tiempo menores. Todos estos datos han sido registrados en un archivo *.xlsx* y procesados mediante técnicas de regresión lineal, que se detallarán a continuación.

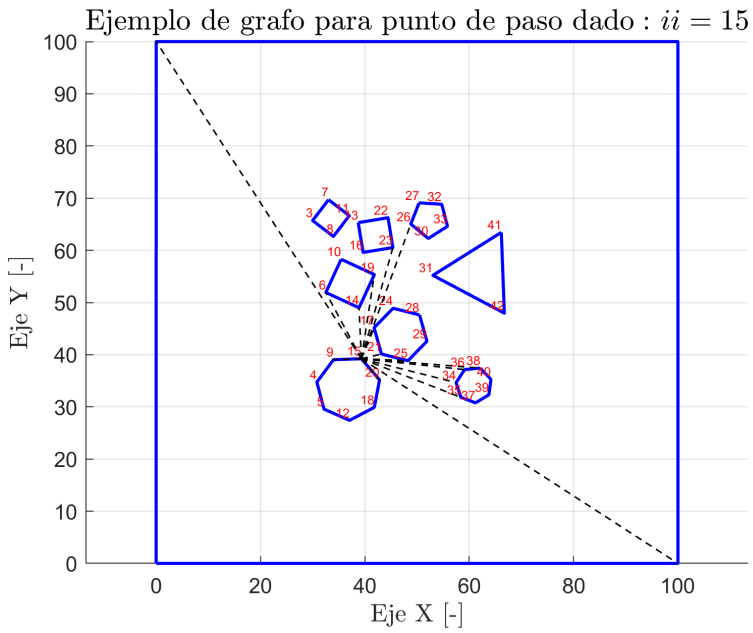


Figura 3.4 Ejemplo de representación de grafo respecto a un solo punto de paso con enfoque en la depuración.

Tabla 3.2 Tiempos de ejecución: Algoritmo basado en *SPLIT*, Ingenuo y Lee.

npol (-)	Promedio de N (-)	Promedio de E (-)	Promedio de $t2_{SPLIT}$ (s)	Promedio de $t2_{Ingenuo}$ (s)	Promedio de $t2_{Lee}$ (s)
10	49	781	0.86	0.37	0.11
15	72	1384	1.71	0.94	0.23
22	102	2296	3.63	2.14	0.43
32	147	3778	7.55	4.93	0.79
46	210	6048	14.88	10.96	1.44
68	310	9808	29.33	25.92	2.74
100	455	15554	56.52	60.10	5.13
147	668	24352	102.96	138.70	9.56
215	974	37673	184.95	314.28	17.58
316	1427	58543	389.97	717.91	32.52

Puede observarse en la tabla 3.2, que para valores de N reducidos, el algoritmo Ingenuo llega a presentar mejores resultados que el *SPLIT*. Esto es debido a que el problema que se resuelve para los casos del método de Lee e Ingenuo es una adaptación del que se plantea para *SPLIT*. De forma consistente, los recintos evaluados en *SPLIT* tienen ocho valores más de puntos de paso, debido a que es necesario añadir el recinto exterior y cuatro puntos más, triángulos auxiliares que contengan el origen y el destino empleados en los métodos de Lee e Ingenuo. Esto para valores de N menores a cien es un aumento necesario de casi el 10 % del número de nodos para poder probar la misma geometría en igualdad de condiciones. Además existe cierto *bias*, debido a que los puntos del polígono-destino adicionales ubicados más a la derecha para el método *SPLIT* contienen un alto número de conexiones en el grafo de visibilidad, lo que aumenta los tiempos de ejecución de esas

iteraciones. Es por ello los valores obtenidos por los métodos planteados en el trabajo de Narciso Valverde, [5], deben tomarse únicamente como una referencia aproximada.

3.3.1 Complejidad algorítimica real

Para estimar la complejidad algorítmica real, se han tomado los datos del doble barrido realizado en el apartado anterior y se les obtenido la recta de regresión. Primero se ha realizado un preanálisis en *Excel*, que es una herramienta más cómoda y directa para trabajar con los datos, comprobando así cual de todos los modelos propuestos de regresión lineal era el más efectivo. Aquel que presentó un mayor coeficiente de correlación fue el modelo potencial:

$$y = Ax^B, \tag{3.1}$$

en especial para valores de B cercanos a 2. Similar a un ajuste polinómico de grado 2, que fue el segundo candidato con mejor ajuste.

Para hacer el análisis en Matlab se tomaron los datos de las diferentes ejecuciones y se realizó un ajuste polinomial sobre los ejes ahora establecidos en base logarítmica:

$$\log y = \log A + B \log x. \tag{3.2}$$

```
%% Análisis de datos
% Lee los datos del archivo Excel en una matriz
data = readmatrix('datos_ejecucion_FINAL.xlsx');

% Muestra los datos leídos en la consola
% disp(data)
%'seed', 'npol', 'N', 't2','E','version'

x=data(:,3);%Fila N
y=data(:,4);%Fila T
E=data(:,5);%Fila E

% Transforma los datos a escala logarítmica
log_x = log(x);
log_y = log(y);

% Ajuste lineal en el espacio log-log
p = polyfit(log_x, log_y, 1);
b = p(1); % Exponente
a = exp(p(2)); % Coeficiente

% Genera valores ajustados para la línea de mejor ajuste
x_fit = linspace(min(x), max(x), 100);
y_fit = a * x_fit.^b;

% Calcula el R^2 para el ajuste no lineal
y_pred = a * x.^b;
SS_res = sum((y - y_pred).^2);
SS_tot = sum((y - mean(y)).^2);
R2 = 1 - (SS_res / SS_tot);
```

Código 3.1 Regresión lineal con *polyfit*.

Los resultados obtenidos para el ajuste propuesto son los siguientes:

$$t = 0.000814N^{1.8060}, \tag{3.3}$$

esto implica conceptualmente que el algoritmo tiene una complejidad algorítmica tal que:

$$t \approx 0.000814N^{1.8060}, O(N^{1.8060}). \tag{3.4}$$

El coeficiente de correlación R^2 es bastante elevado siendo este igual a $R^2 = 0.9321$.

Este resultado, sin llegar a ser el originalmente planteado de $O(N \log N + E)$, aunque tampoco se conocía una tendencia como tal de E con N para empezar, presenta un comportamiento asintótico no tan eficiente como el planteado en la implementación de Narciso Valverde del método de Lee y el algoritmo Ingenuo, siendo las pendientes obtenidas a partir de los resultados del estudio paramétrico $m_{Ing,v2} = 2.15$ y $m_{Lee} = 1.63$, similares a las documentadas en su estudio, [5]. Esto corresponde con los datos de tiempos de ejecución obtenidos para valores altos de N, siendo el más eficiente, la implementación de Lee, a continuación el modelo *SPLIT* y después el algoritmo Ingenuo. Nótese que este cambio de forma de recinto ha tenido un efecto de disminución de rendimiento, que se ve reflejado en un aumento ligero de la pendiente obtenida. Una variación relativa tan pequeña puede verse debida al ajuste, aún así los coeficientes de correlación obtenidos $R^2_{Lee} = 0.9426$ y $R^2_{Ingenuo\,v2} = 0.9172$ siguen siendo aceptables.

Tabla 3.3 Coeficientes de las rectas de ajuste: Método de Lee, *SPLIT* e $Ingenuo_{v,2}$.

Métodos	**Lee**	*SPLIT*	$Ingenuo_{v,2}$
B	1.63	1.81	2.15
R^2	0.9321	0.9426	0.9172

Para determinar matemáticamente la pendiente teórica del modelo, sería necesario conocer en primer lugar la relación matemática entre N y E, se puede estimar que tiene una cota superior de N^2, pero el valor real que se obtenga dependerá de la geometría. Es por ello que se ha realizado un segundo ajuste.

En este segundo ajuste se ha tomado la fórmula de la complejidad algorítmica propuesta, y se ha evaluado para uno de los valores conocidos de tiempo, N y E (en concreto para $seed = 3$ y $npol = 10$):

$$t_{teo}(N,E) = A(N \log N + E), A = \frac{t_{3,10}}{N_{3,10} \log N_{3,10} + E_{3,10}}. \tag{3.5}$$

Una vez conocida la constante, sobre la ecuación planteada anteriormente, se evalúan todas las combinaciones de N,E computadas. Se hace entonces la suposición de que E es dependiente de N, y se realiza un ajuste de los tiempos teóricos frente a N, obteniéndose los siguientes resultados:

$$t_{teo} = 0.010386 N^{1.2568}, \tag{3.6}$$

siendo 1.2568 una escalabilidad increíblemente buena que no se ha llegado a alcanzar debido a las limitaciones propias del método y de la implementación.

Se adjunta la figura 3.5, mostrando las rectas de regresión en ejes doblemente logarítmicos, junto con los casos de estudio resultado de emplear el método *SPLIT*.

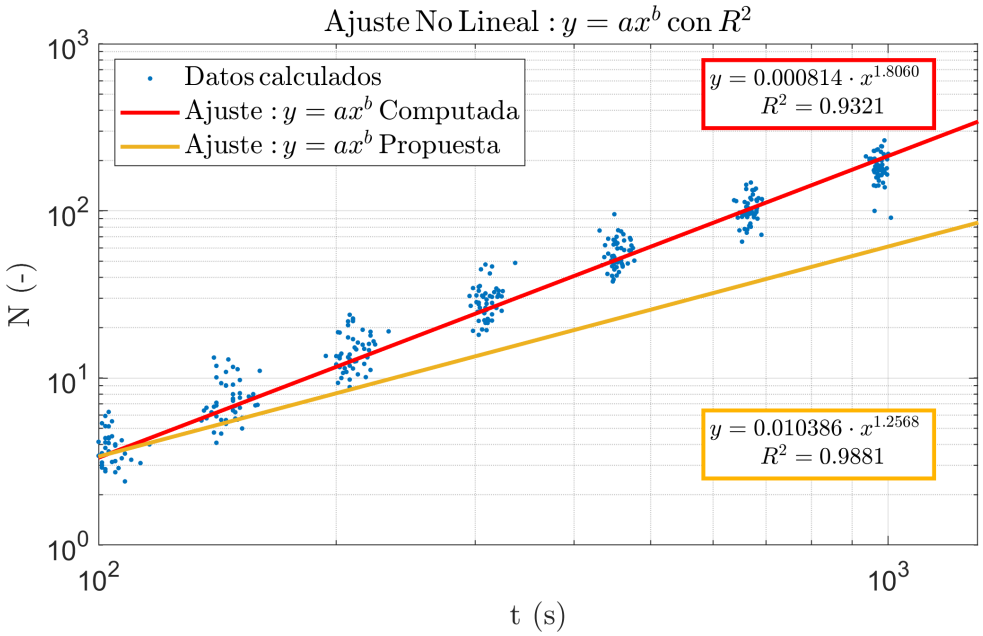


Figura 3.5 Ajustes realizados para la complejidad algorítmica teórica obtenida comparada con la propuesta.

3.3.2 Estimación matemática de E

Aprovechando que se han computado y registrado todos los casos anteriormente citados, se adjunta como ejercicio la estimación del número de conexiones del diagrama de visibilidad, E, frente al número de puntos de paso, N.

Para ello se ha empleado directamente las herramientas de ajustes de Excel y los resultados obtenidos para este subconjunto de recintos calculados son los siguientes:

$$E = f(N) \approx 6.168422N^{1.2724}. \tag{3.7}$$

Esto es interesante, dado que se puede observar cierta similitud entre la pendiente en escala logarítmica del ajuste potencial de la complejidad propuesta con el valor de la pendiente del ajuste de E. Esto podría confirmar la hipótesis establecidas de que asintóticamente, el factor dominante en la expresión:

$$O(N\log N + E), \tag{3.8}$$

pudiera ser E. Esto reafirma la consideración, de que si lo que se busca es un método estable en lo referente a tiempos y recursos de computación necesarios, podría ser más interesante emplear estrategias que no dependan del número de conexiones del grafo de visibilidad.

Se adjunta la figura 3.6 que muestra los datos calculados anteriormente explicados.

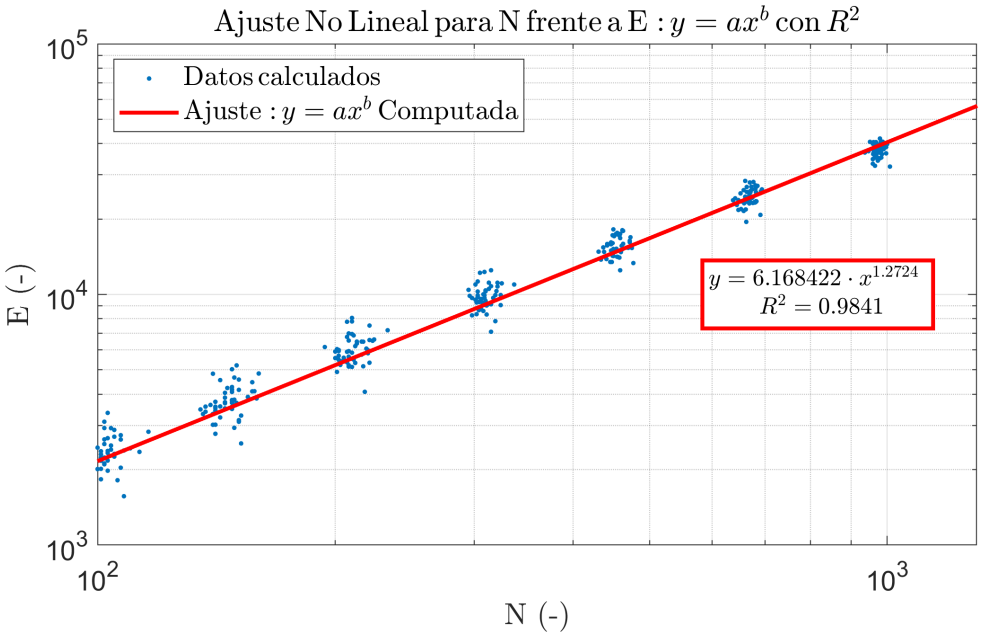


Figura 3.6 Estudio del comportamiento del número de conexiones en el grafo, E, respecto al número de puntos de paso N.

3.3.3 Desglose de tiempos utilizando *profiler*

Para ver como de eficiente es el algoritmo internamente, más allá de la complejidad algorítmica mostrada, se va a hacer uso de la herramienta de *profiler*. Esta permite para un intervalo de código concreto, al igual que se haría con un *tic-toc*, comprobar cuanto tiempo se ha invertido en cada sección del código pudiendo así optimizar de manera más eficiente.

Esta herramienta ha sido ampliamente usada a lo largo del desarrollo. Dado que se estaba realizando una implementación no fiel a la estructura de datos originalmente propuesta, para conseguir resultados aceptables en cuanto a tiempos de ejecución, se ha estado vigilando las partes internas del programa. Buscando diversas estrategias de optimización minimizando el número de cálculos y maximizando el número de casos que pueden ser procesados con garantía de validez de resultados.

```
1 profile clear
2 profile on
3
4 [t2,visibilidad]=Main_v13(V);
5
6 profile off
7 profile viewer
```

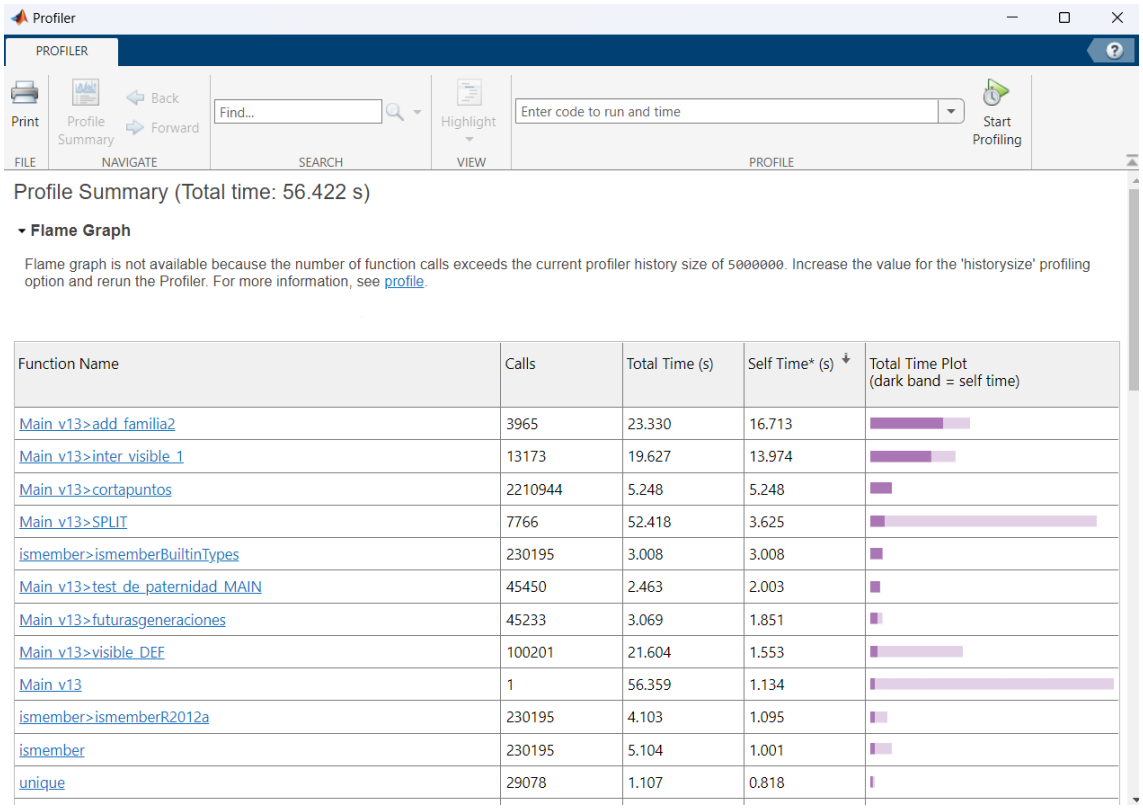
Código 3.2 Ejemplo de uso de *profiler*.

Los argumentos utilizados para sacar el máximo partido a la herramienta son los siguientes:

1. *clear*: Limpia el historial registrado.
2. *on*: Indica a Matlab que empiece a registrar los datos de la ejecución.
3. *off*: Análogo a *on* pero para que se detenga.
4. *viewer*: Abre el lector, que contiene la información de interés.

Cabe destacar que el empleo de la misma empeora los tiempos de rendimiento, dado que se registran de forma activa los tiempos de ejecución y llamadas a las funciones. Por suerte, esto no resulta un gran problema, dado que es una herramienta de depuración.

Para un caso de estudio como el de los recintos convexos planteado anteriormente, $seed = 1$ y $npol = 100$, se pueden observar a continuación los tiempos de computación obtenidos:



Function Name	Calls	Total Time (s)	Self Time* (s)	Total Time Plot (dark band = self time)
Main_v13>add_familia2	3965	23.330	16.713	
Main_v13>inter_visible_1	13173	19.627	13.974	
Main_v13>cortapuntos	2210944	5.248	5.248	
Main_v13>SPLIT	7766	52.418	3.625	
ismember>ismemberBuiltinTypes	230195	3.008	3.008	
Main_v13>test_de_paternidad_MAIN	45450	2.463	2.003	
Main_v13>futurasgeneraciones	45233	3.069	1.851	
Main_v13>visible_DEF	100201	21.604	1.553	
Main_v13	1	56.359	1.134	
ismember>ismemberR2012a	230195	4.103	1.095	
ismember	230195	5.104	1.001	
unique	29078	1.107	0.818	

Figura 3.7 Resultados obtenidos del *profiler* para $seed = 1$ y $npol = 100$.

Los dos principales sumideros de carga computacional durante la ejecución son, precisamente, los derivados de las características de la implementación, $inter\_visible1$ que calcula si existen algún corte entre las aristas vinculadas a los nodos visibles para la base de la familia de embudos que se esté $SPLIT$eando, con la línea de visión entre ii y el nodo en cuestión. $corta\_puntos$ es una función hija de $inter\_visible$ que calcula si dos segmentos de recta intersecan entre sí. Por otro lado $add\_familia2$ que es la función que calcula la familia de embudos que se solicite ordenando los nodos asociados a la base de la familia.

$add\_familia$ e $inter\_visible1$ ocupan aproximadamente un 90 % del tiempo total de ejecución. La complejidad algorítmica propuesta por Ghosh & Mount resultaría factible entonces, de implementarse con los requisitos mencionados, pudiendo mejorarse así la complejidad algorítmica propuesta por el método de Lee, $O(N^2 \log N)$, [5].

Ambas ejecuciones son críticas y afectan en gran manera al resultado final obtenido, por ejemplo, si la familia obtenida está mal ordenada porque existiera algún nodo que se ha insertado en el padre equivocado, todo el fundamento teórico en el que se basa $SPLIT$ queda completamente invalidado, pudiendo provocarse falsos positivos y falsos negativos, que además irían propagándose durante el resto de la ejecución completa del bucle debido a a naturaleza aditiva del método.

Continuando esta línea de pensamiento, si debido a que la información de las familias no se encuentra completa a causa de un problema con la ordenación eficaz de los nodos, podrían darse falsos positivos al no encontrarse aristas que produzcan un corte. Aunque es posible, que debido a

la robustez artificial proporcionada al método, con varias capas de comprobaciones y condiciones, estos fallos no lleguen a propagarse, pero la complejidad computacional se ha visto lastrada por esto. Una de estas comprobaciones es, por ejemplo, el añadir a la lista de nodos de la familia de embudos, los vértices de los polígonos que están en la familia. Esto es necesario para ejemplos como el de la figura 3.8. En polígonos altamente complejos las estrategias de visibilidad necesitan ser adaptadas, es por ello que existe una versión *a* y *b* del código, cada una implementando estrategias de visibilidad distintas con este propósito. Todas estas técnicas, al final, se sirven como refuerzo al concepto de los embudos y sus familias. En última instancia ha sido necesario establecer un equilibrio entre garantizar familias completas y realizar comprobaciones de visibilidad mediante cortes.

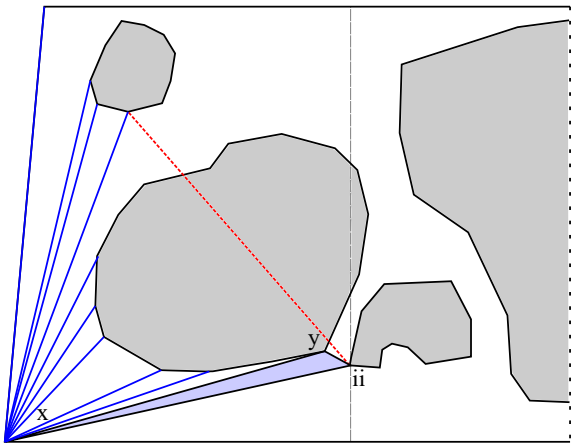


Figura 3.8 Caso de intersección no detectada sin comprobaciones adicionales de todos los vértices asociados a los polígonos de la familia de embudos.

El método de cálculo de visibilidad presentado en la sección 2.3.3, la basada en la definición de los embudos, llegó a funcionar de forma sobresaliente en etapas tempranas de la implementación, pero acabó siendo dejada en segundo plano en aras de garantizar en mejor medida resultados fiables, debido a la casuística comentada en la sección mencionada anteriormente.

El hecho de que la escalabilidad de la complejidad algorítmica calculada empíricamente presente resultados tan notables, aunque los tiempos de computación sean elevados, no es si no un indicador del potencial que tiene este método. Aunque se realizará una disertación en las conclusiones acerca de si la relación costo-riesgo-beneficio de trabajar con algoritmos así de complejos pudiera resultar a la larga una desventaja.

Finalmente, los resultados obtenidos son mejores que los del algoritmo Ingenuo debido a que aunque se pueda llegar a hacer un número de comprobaciones considerables de cortes de segmentos, se hace siguiendo la estrategia que proporcionan las familias. Esto representa en sí, un éxito.

4 Conclusiones y futuras líneas de trabajo

Presentado el método, la implementación y los resultados obtenidos se desarrollarán las lecciones aprendidas, diversas puntualizaciones que pudieran resultar de interés para el lector y diferentes propuestas de desarrollo.

El primer punto a tratar será la complejidad de la implementación del algoritmo, ¿es eficiente desde un punto de vista ingenieril considerando el entorno que rodea a la resolución del problema? El principal atractivo de implementar un algoritmo así de complejo, pero que escala de manera tan eficiente, es poder computar de forma muy precisa rutas que tanto directamente como indirectamente, reportan un beneficio a diferentes escalas, tal y como se planteó en la introducción.

Fuera del entorno teórico existen varias limitaciones y restricciones que podrían echar por tierra estos beneficios obtenidos. ¿Es necesario realmente considerar cada obstáculo por separado y respetando sus concavidades? Podría plantearse el caso de que considerar estas concavidades es imprescindible si, por ejemplo, se quisiera aterrizar de forma segura en un aeropuerto que se encontrara ubicada en una; pero el análisis realizado se ha hecho para un único nivel de vuelo, y lo que es más crítico, considerando los obstáculos fijos. Por supuesto, existen en la vida real obstáculos fijos, como áreas de vuelo restringidas delimitadas geográficamente, pero las rutas o sugerencias de ruta de un vuelo comercial están trazadas de forma que estas condiciones de contorno estén evitadas por defecto, dado que todos los actores que toman parte del mundo aeroespacial deben ser consciente de estas limitaciones a la hora de establecer rutas de acuerdo con los estándares de seguridad operacional y física esperadas en el sector.

Es por ello, que a la hora de la verdad, los agentes meteorológicos juegan un papel muy importante a la hora de tomar decisiones en el día a día. No se plantean de la misma manera una operación de aterrizaje o despegue con condiciones óptimas de visibilidad y/o viento alineado con la pista, que días con niebla, nieve, lluvia intensa o fuertes vientos cruzados. En condiciones extremas, se encuentran contemplados procedimientos para aterrizar en aeropuertos cercanos e incluso que la torre de control no proporcione autorización para despegar. Las tormentas se desarrollan con el tiempo en la atmósfera, pudiendo cambiar su posición y forma en ordenes de magnitud temporales inferiores a la duración estándar de un vuelo comercial. Entonces cabe plantearse lo siguiente, ¿resulta interesante usar algoritmos muy complejos que puedan detectar caminos y pasadizos angostos en un contexto aeronáutico? ¿Existen garantías de que esa ruta exista una vez comenzada?

Asimismo, el cálculo desde tierra de tormentas para un nivel de vuelo dado no es tan preciso como el algoritmo que se ha planteado, [11]. Todos estos factores, acumulados, desde el punto de vista del redactor, hacen más interesante de cara a líneas de desarrollo futuras, optar por estrategias que trabajen con geometrías más sencillas y entornos dinámicos, estableciendo metodologías de decisión de ruta basados en el control en bucle cerrado con diferentes horizontes temporales.

Por otro lado, la implementación, aunque haya satisfecho las expectativas propuestas, plantea una serie de dudas. Los resultados son satisfactorios en cuanto a la capacidad de escalado del método con N, pero la implementación es tan robusta como sensible. Están programados dentro del código secciones enteras de comprobaciones adicionales que han tenido que desestimarse en aras de mantener la complejidad en rangos aceptables, lo cual no ha representado un problema en la extensa variedad de casos de estudio depurados pero cabe la posibilidad de que se dieran inconsistencias, si se considera la casuística en geometrías increíblemente complejas. Continuando con esta línea de desarrollo, establecer la complejidad algorítmica en base al número de conexiones visibles podría considerarse delicado, dado que esa información no se conoce de antemano. Es por ello que el artículo original se traduce como *Un algoritmo sensible a los resultados para resolver grafos de visibilidad*.

Todo esto deriva a que los tiempos de computación no sean consistentes entre sí, dado que diferentes geometrías pueden llevar a comportamientos del algoritmo diametralmente opuestos. En el capítulo anterior se pudo observar, que para un número similar de puntos, existen casos que se resuelven en la mitad de tiempo debido a sus peculiaridades. Después de analizar los resultados, se ha estimado que E, el número de conexiones en un grafo de visibilidad en el contexto de los casos de estudio es aproximadamente:

$$E \approx N^{\sqrt{2}}, \tag{4.1}$$

que es sin lugar a dudas, superior a la complejidad algorítmica del término que si era dependiente de N,

$$O(N \log N + E \approx N^{\sqrt{2}}); \tag{4.2}$$

que es similar a la pendiente de la curva de regresión potencial propuesta en el apartado anterior. Aún así, en el propio documento original no se especifica directamente la complejidad algorítmica del algoritmo de triangulación empleado, sólo la complejidad del de Mehlhorn, [9]. Este algoritmo no está planteado para funcionar en recintos con agujeros, pero se indica que este *se generaliza fácilmente*, [6], sin indicar ninguna estrategia, ni aportar consideraciones adicionales acerca de como esto afecta a la complejidad algorítmica del método de triangulación original. De la misma manera, no se han especificado condiciones de parada en la recurrencia propuesta, pero la implementación realizada consigue sacar partido de ella, siendo esta consistente con los tiempos de computación.

Cabe recalcar que, existe una variabilidad de casos que no llega a ser completamente cubierta en el documento original, habiéndose considerado imprescindible aplicar varias capas de parches para cubrir los casos límite, véase la referencia [12], dónde se comentan ciertas dificultadas similares a las documentadas en esta memoria. Estas situaciones, originalmente planteadas como límite y no cubiertas del todo por la casuística, en entornos muy complejos y con muchos obstáculos, podrían darse de forma recurrente. Otra desventaja a comentar sería la necesidad técnica de realizar un preprocesado más extenso que para otros métodos, debido a que se pide como dato, todos los nodos ordenados unos alrededor de otros, tal y como se especifica en la sección 7 de [6], o que los puntos de paso deban estar ordenados en orden de x creciente, esto introduce cuellos de botella independientemente de la estructura de datos que se emplee.

En última instancia, de considerarse apropiado continuar esta línea de investigación a la hora de resolver el grafo de visibilidad, se recomendaría implementar la estructura de datos según lo planteado en el documento original, la propuesta por Gabow & Tarjan, [10]. Las decisiones adoptadas en este trabajo se han visto influenciadas por consideraciones relacionadas con el lenguaje de programación utilizado, *Matlab*™, los conocimientos específicos en el ámbito de la ciencia de la computación y el horizonte temporal del mismo. Aún así, bajo estas circunstancias se han podido alcanzar de forma satisfactoria los objetivos originalmente propuestos.

Apéndice A
Códigos de Matlab

En este apéndice se recopilarán todos los códigos que no hayan sido expuestos a lo largo del documento. Se hará un esquema y resolverán posibles dudas que pudieran surgir acerca de las relaciones de los mismos.

A.1 Relaciones entre los códigos

A lo largo de los meses invertidos en este trabajo, se ha generado una gran cantidad de código que recoge varios programas y subrutinas que sirven de apoyo a las secciones principales comentadas a lo largo del presente documento.

Existen 5 archivos con extensión *.m* que recogen la última versión de los algoritmos empleados:

1. Programas *parent*: Sirven de *hub* y gestionan las llamadas a las funciones *child*. En estas se realiza gestiona el entorno que envolverá a la ejecución.
    - Ejecutaprograma: Diseñado para las ejecuciones únicas del programa: Gestiona la creación de un recinto, llama a Main_v13_C y al *profiler*.
    - Estudio_parametrico: Usando el módulo de *Parallel Computing*, realiza un barrido para tantas semillas como se le indique y para valores de polígonos generados por un *logspace*. Toda esta información se almacena en un archivo de extensión *.xlsx*, para procesar los datos de forma rápida con las herramientas manuales que proporciona.
    - Compatibiliza: Importa los datos empleados en [5]. Adapta la matriz de coordenadas, debido a que los nodos en los polígonos están dados en sentido horario respecto al centroide y llama a la función Main_v13_C para obtener el grafo de visibilidad.
2. Programas *children*: su ejecución está autocontenida y proporcionan una salida concreta.
    - calcula_recinto: Devuelve un recinto parametrizado en función de *seed* y *npol*.
    - Main_v13_C: Devuelve el grafo de Visibilidad y el tiempo necesario para ejecutarlo.

Se proporcionarán los códigos en el orden en el que se han descrito y todas las subfunciones que pudieran tener serán añadidas con una breve contextualización que desarrolle los comentarios adjuntos en el código.

A.2 Programas *parent*

A.2.1 Ejecutaprograma

```
%% Código de arranque del programa
clc, clear all

seed=2;
npol=50;

V=calcula_recinto(seed,npol);

profile clear
profile on

[t2,visibilidad]=Main_v13_C(V);
t2

profile off
profile viewer
```

Código A.1 Ejecutaprograma.m.

A.2.2 Estudio_parametrico

```
%% Itera_ejecuciones en paralelo
tic
reiniciar = true;  % Cambia a 'false' si no se desea borrar el contenido

% Si se indica reiniciar, borra todo el contenido del archivo
if reiniciar
    delete('datos_ejecucion_.xlsx');
end

% Inicializa un grupo de trabajadores si no está activo
if isempty(gcp('nocreate'))
    parpool;  % Inicia un pool de procesamiento en paralelo
end

% Número de iteraciones para cada bucle
nseed = 50;

% Utiliza parfor para el bucle exterior en paralelo
parfor seed = 1:nseed
    for npol = round(logspace(1,2.5,10))

        V = calcula_recinto(seed, npol);
        [t2, visibilidad] = Main_v13(V);

        % Guardar los resultados en el archivo Excel
        guarda_excel(t2, visibilidad, seed, npol);
    end
strcat(num2str(seed))
end

t=toc

function guarda_excel(t2,visibilidad,seed,npol)

% Nombre del archivo Excel
archivo = 'datos_ejecucion.xlsx';

% Definir los valores que se quieren registrar

visibilidad(isnan(visibilidad))=0;
E=sum(sum(visibilidad));
[N,~]=size(visibilidad);
version=13;

% Cargar los datos previos, si existen, para conocer la próxima fila vacía
if isfile(archivo)
```

```
    % Leer el contenido del archivo para determinar la próxima fila vacía
    datosPrevios = readcell(archivo);
    filaInicio = size(datosPrevios, 1) + 1;  % Fila vacía
else
    % Si el archivo no existe, se creará y se iniciará en la primera fila
    filaInicio = 1;
end

% Escribe el encabezado si es la primera vez
if filaInicio == 1
    encabezado = {'seed', 'npol', 'N', 't2','E','version'};
    writecell(encabezado, archivo, 'Sheet', 1, 'Range', 'A1');
    filaInicio = filaInicio + 1;  % La siguiente fila vacía
end

% Escribe los datos en la fila disponible
nuevaFila = {seed, npol,N, t2,E,version};
writecell(nuevaFila, archivo, 'Sheet', 1, 'Range', ['A' num2str(filaInicio)]);

end
```

Código A.2 Estudio_parametrico.

A.2.3 Compatibiliza

```
clc,clear all, close all
% Compatibiliza la matriz de coordenadas para el caso de estudio del método
% de Lee
load('TFG_Narciso.mat')

V=[LON,LAT];
%% Transformación de V
% En este vector de coordenadas los datos de los nodos asociados a los
% polígonos están dado en orden antihorario respecto al centroide, por lo
% que se realizará a continuación una adaptación para que sea compatible
% con el código Main_v13

poligonos = {};     % Celdas para almacenar los polígonos
current_poly = [];  % Almacena los nodos actuales

% Separar los polígonos utilizando NaN
for i = 1:size(V, 1)
    if any(isnan(V(i, :))) % Si es un NaN
        if ~isempty(current_poly) % Si hay nodos acumulados
            poligonos{end + 1} = current_poly; % Guardar el polígono
            current_poly = []; % Reiniciar
        end
    else
        current_poly = [current_poly; V(i, :)]; % Añadir nodo
    end
end

% Agregar el último polígono si existe
if ~isempty(current_poly)
    poligonos{end + 1} = current_poly;
end

% Reordenar los nodos en cada polígono
for i = 1:length(poligonos)
    if ~isempty(poligonos{i})
        % Cambiar el orden; aquí lo hacemos de la manera deseada
        poligonos{i} = poligonos{i}(end:-1:1, :); % Invertir el orden
    end
end

% Combinar los polígonos reordenados en un solo vector
V_reordenado = [];
for i = 1:length(poligonos)
```

```
    V_reordenado = [V_reordenado; poligonos{i}]; % Añadir el polígono
    V_reordenado = [V_reordenado; NaN(1, 2)]; % Añadir el NaN separador
end

% Eliminar el último NaN si es necesario
if all(isnan(V_reordenado(end, :)))
    V_reordenado(end, :) = [];
end

% Mostrar el vector reordenado
disp(V_reordenado);
V=V_reordenado;

V400=V;

vminx=min(V400(:,1));
vminy=min(V400(:,2));

border=10;

Vmod=zeros(size(V400));

if vminx<=0
Vmod(:,1)=V400(:,1)+vminx+border;
Vmod(:,2)=V400(:,2);
% vminx=border;

elseif vminy<=0
Vmod(:,2)=V400(:,2)+vminy+border;
% vminx=border;
Vmod(:,1)=V400(:,1);

else
    Vmod=V400;
end

vminx=min(Vmod(:,1))*0.9;
vminy=min(Vmod(:,2))*0.9;

vmaxx=max(Vmod(:,1))*1.1;
vmaxy=max(Vmod(:,2))*1.1;

Vmod(:,1)=(Vmod(:,1)-vminx)./((vmaxx-vminx));
Vmod(:,2)=(Vmod(:,2)-vminy)./((vmaxy-vminy));

Vmod=[0,0;1,0;1-1e-4,1;1e-4,1,;0,0;NaN(1,2);Vmod;NaN(1,2)];

indprimernan=find(isnan(Vmod(:,1)),1);

vminpostx=min(Vmod(indprimernan+1:end,1));
vmaxpostx=min(Vmod(indprimernan+1:end,1));

D1=vminpostx-Vmod(4,1);
D2=abs(vmaxpostx-Vmod(3,1));

origen=[Vmod(4,1)+0.25*(D1),0.5];

destino=[Vmod(3,1)-0.1*(D2),0.5];

poligonoorigen=[origen;Vmod(4,1)+0.1*(D1),0.5*1.1;Vmod(4,1)+0.1*(D1),0.5*0.9;origen];
poligonodestino=[destino;Vmod(3,1)-0.01*(D2),0.5*0.9;Vmod(3,1)-0.01*(D2),0.5*1.1;destino];

Vmod=[Vmod;poligonoorigen;NaN(1,2);poligonodestino];

[t2,visibilidad]=Main_v13_C(Vmod);
t2
```

Código A.3 Compatibiliza.

A.3 Programas *Children*

A.3.1 calcula_recinto

Esta función realiza la algoritmia desarrollada en 3.2, genera recintos basados en una semilla dada, configurada mediante $rng(seed)$ y normalizados para x e y entre $[0,100]$.

Modificando los siguientes parámetros puede cambiarse la naturaleza de los recintos:

- *maxaristaspol*: Número máximo de aristas permitidos por polígono.
- *minrad*: Tamaño mínimo del radio en relación a la posición frente a la arista del recinto externo más cercano.
- $factormargen$: Define la mínima distancia a mantener entre el centroide del polígono y el margen.

Esta función tiene tres funciones *children* asociadas:

- distancia: Calcula la distancia entre dos puntos dado su vector director. Su usa para comprobar que la distancia entre centroides sea suficiente para evitar interferencias.
- distance_point_to_polygon: Calcula la mínima distancia del centroide a añadir respecto al recinto exterior.
- point_to_segment_distance: Calcula la mínima distancia entre un punto y una recta.

```
1  function V=calcula_recinto(seed,npol)
2  %% Inicio Programa
3
4  %Se va a generar un código que reciba como input el número de polígonos,
5  %máximo de lados por cada uno, y diferentes parámetros de forma y tamaño y
6  %calcule los diferentes dominios y los de como input al algoritmo.
7
8
9  %% Marco inicial
10 V0=[0,0;100,0;99.99,100;0.01,100;0,0;NaN(1,2)];
11 V=[0,0;100,0;99.99,100;0.01,100;0,0;NaN(1,2)];
12
13 rng(seed)
14 maxaristaspol=6;
15 factormargen=10;%10
16
17 minrad=20;%20 %Tamaño mínimo en relación al margen
18
19 margenx=(max(V(:,1))-min(V(:,1)))/factormargen;
20 margeny=(max(V(:,2))-min(V(:,2)))/factormargen;
21
22 margen=max([margenx,margeny]);
23 vectorcentro=[];
24 vectordist=[];
25
26 for ii=1:npol
27
28     xcentro=margen+(100-2*margen)*rand(1);
29     ycentro=margen+(100-2*margen)*rand(1);
30     R=min([distance_point_to_polygon(V0, xcentro, ycentro)*rand(1),minrad]);
31     contador=1;
32
33     if ~isempty(vectorcentro)
34         while contador<=length(vectorcentro(:,1))
35
36             if dist([xcentro,ycentro],vectorcentro(contador,:))<(R+vectordist(contador))*1.1
37                     xcentro=margen+(100-2*margen)*rand(1);
38                     ycentro=margen+(100-2*margen)*rand(1);
```

```
                    R=min([distance_point_to_polygon(V0, xcentro, ycentro)*rand(1),minrad]);
                    contador=0;
            end
            contador=contador+1;
        end
    end

    vectorcentro=[vectorcentro;xcentro,ycentro];
    vectordist=[vectordist;R];

    lados=3+round((maxaristaspol-3)*rand(1));

    xp1=xcentro-R+2*R*(rand(1));
    yp1=sqrt(R^2-(xp1-xcentro)^2)+ycentro;

    vectorpol=[xp1,yp1];
    theta=360*pi/180/lados;

    R=[cos(theta),-sin(theta);sin(theta),cos(theta)];
    v=[xp1;yp1]-[xcentro;ycentro];


    for jj=1:lados-1
        v=R*(v);
        vectorpol=[vectorpol;(v+[xcentro;ycentro])'];
    end

    V=[V;vectorpol;vectorpol(1,:);NaN(1,2)];

    % mapshow(V(:,1),V(:,2))
end


end

function distancia=dist(v1,v2)

distancia=sqrt((v2(1)-v1(1))^2+(v2(2)-v1(2))^2);

end

function d_min = distance_point_to_polygon(V, x0, y0)
    % Inicializar la distancia mínima con un valor grande
    d_min = Inf;

    % Remover el NaN al final del vector V
    V = V(~any(isnan(V), 2), :);

    % Iterar sobre cada segmento del polígono
    num_points = size(V, 1);
    for i = 1:num_points - 1
        % Puntos del segmento
        x1 = V(i, 1);
        y1 = V(i, 2);
        x2 = V(i+1, 1);
        y2 = V(i+1, 2);

        % Calcular la distancia desde el punto (x0, y0) al segmento (x1, y1)-(x2, y2)
        d = point_to_segment_distance(x1, y1, x2, y2, x0, y0);

        % Actualizar la distancia mínima
        if d < d_min
            d_min = d;
        end
    end
end

function d = point_to_segment_distance(x1, y1, x2, y2, x0, y0)
    % Vector del segmento
    v = [x2 - x1, y2 - y1];
    % Vector desde el punto inicial del segmento hasta el punto dado
```

```
110     w = [x0 - x1, y0 - y1];
111
112     % Proyección escalar de w sobre v
113     c1 = dot(w, v);
114     if c1 <= 0
115         % El punto más cercano es (x1, y1)
116         d = norm([x0 - x1, y0 - y1]);
117         return;
118     end
119
120     % Longitud al cuadrado del segmento
121     c2 = dot(v, v);
122     if c2 <= c1
123         % El punto más cercano es (x2, y2)
124         d = norm([x0 - x2, y0 - y2]);
125         return;
126     end
127
128     % Proyección escalar normalizada
129     b = c1 / c2;
130     % Punto más cercano en el segmento
131     pb = [x1, y1] + b * v;
132     % Distancia desde el punto dado al punto más cercano en el segmento
133     d = norm([x0 - pb(1), y0 - pb(2)]);
134 end
```

Código A.4 Función calcula_recinto.

A.3.2 Main_v13_C

Este algoritmo ha sido mostrado a lo largo del capítulo 2 de este documento, por lo que se mostrarán aquellas funciones que no pudieron ser desarrolladas.

calcula_fronteras

Este fragmento de código genera la matriz que sirve como base para calcular *Caminos_prohibidos*, toma la información acerca de los vértices de un polígono y comprueba los caminos internos entre ellos. Si no pasa por dentro del polígono y no corta ninguna de sus aristas necesariamente es un camino permitido.

Es necesaria comprobar si pasa por dentro del polígono o no, dado que en el caso de los polígonos convexos los cortes en esos casos solo se producen con las aristas que conforman los vértices.

Con esta finalidad existen las siguientes subrutinas:

1. doesCrossPolygon: Si hay un corte entre una combinación de segmentos de recta compuesta por vértices no consecutivos a través de una arista del polígono cualesquiera, devuelve un *true*. Para ello, para una recta dada por dos vértices, comprueba si ocurren cortes con cualquier otra combinación de aristas de las que estos vértices no sean partícipes.
2. isOutsideArc: Implementa la propiedad explicada en la figura 1.1 en el capítulo 1. Diferenciando los casos concavidad localizada y convexidad localizada.

doesCrossPolygon se basa en el siguiente principio que se da si se tienen dos segmentos de recta que que podrían o no intersecar en algún punto. Si se calcularan los productos vectoriales resultantes de proyectar los otros dos puntos sobre la arista de referencia, y estos dos productos vectoriales tienen diferente signo; de ser el caso que estos dos puntos proyectados se tomaran ahora como referencia y a continuación se proyectaran sobre esta nueva referencia los puntos de antes, de tener diferente signo también, estas dos aristas tendrían necesariamente que haber intersecado. Se adjunta la figura A.1 para ilustrar este hecho. En el caso de que 3 puntos sean colineares, que no debería tener lugar tal y como se ha planteado la matriz de coordenadas, se comprueba si cualquiera de los

otros puntos se encuentra en las rectas que conforman con *onSegment*. Que comprueba si un tercer punto se encuentra entre los límites de un segmento.

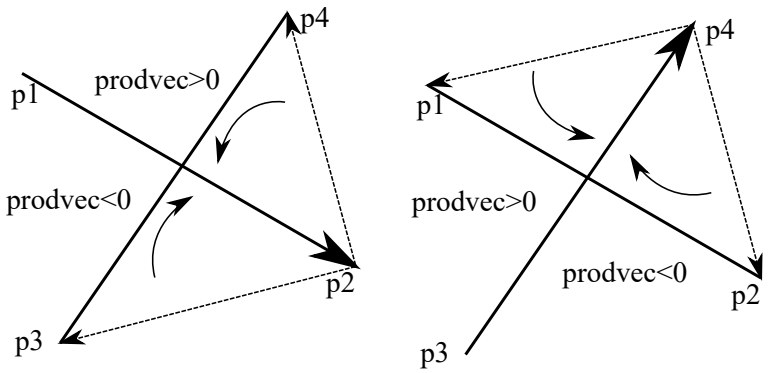


Figura A.1 Aplicación doesCrossPolygon.

```
function possiblePaths=calcula_fronteras(polygonVertices)

N = size(polygonVertices, 1);

% Inicializar matriz de caminos posibles
possiblePaths = true(N);  % Inicialmente se asume que todos los caminos son posibles

% Revisar caminos entre cada par de vértices
for i = 1:N
    for j = i+1:N
        if ~doesCrossPolygon(polygonVertices, i, j) && isOutsideArc(polygonVertices, i, j)
        else
            possiblePaths(i, j) = false;  % Camino prohibido
            possiblePaths(j, i) = false;
        end
    end
end

end

function crosses = doesCrossPolygon(polygonVertices, vi, vj)
N = size(polygonVertices, 1);
p1 = polygonVertices(vi, :);
p2 = polygonVertices(vj, :);

for k = 1:N
    p3 = polygonVertices(k, :);
    p4 = polygonVertices(mod(k, N) + 1, :);

    if ~(vi == k || vi == mod(k, N) + 1 || vj == k || vj == mod(k, N) + 1)
        if segmentsIntersect(p1, p2, p3, p4)
            crosses = true;
            return;
        end
    end
end
crosses = false;
end

function outside = isOutsideArc(polygonVertices, vi, vj)
N = size(polygonVertices, 1);

% Coordenadas de los vértices
p1 = polygonVertices(vi, :);
p2 = polygonVertices(vj, :);

% Vértices adyacentes al vértice vi
prev = mod(vi-1-1, N) + 1;  % Vértice anterior (cerrando el bucle)
```

```
next = mod(vi+1-1, N) + 1;    % Vértice siguiente

% Vectores de ángulo
vPrev = polygonVertices(prev, :) - p1;
vNext = polygonVertices(next, :) - p1;
vCheck = p2 - p1;

prodvec_csobrep=vCheck(1)*vPrev(2)-vCheck(2)*vPrev(1);
prodvec_nsobrep=vNext(1)*vPrev(2)-vNext(2)*vPrev(1);
prodvec_csobren=vCheck(1)*vNext(2)-vCheck(2)*vNext(1);

if prodvec_nsobrep>0
    if prodvec_csobren<0 && prodvec_csobrep>0
        outside=false;
    else
        outside=true;
    end
else
    if prodvec_csobren>=0 && prodvec_csobrep<=0
        outside=true;
    else
        outside=false;
    end
end

end

function intersect = segmentsIntersect(p1, p2, p3, p4)
% Verificar si los segmentos p1p2 y p3p4 se intersectan
    function o = orientation(p, q, r)
        val = (q(2) - p(2)) * (r(1) - q(1)) - (q(1) - p(1)) * (r(2) - q(2));
        if abs(val) < 1e-10
            o = 0; % Colineales
        elseif val > 0
            o = 1; % Sentido horario
        else
            o = 2; % Sentido antihorario
        end
    end

o1 = orientation(p1, p2, p3);
o2 = orientation(p1, p2, p4);
o3 = orientation(p3, p4, p1);
o4 = orientation(p3, p4, p2);

if o1 ~= o2 && o3 ~= o4
    intersect = true;
    return;
end

intersect = (o1 == 0 && onSegment(p1, p3, p2)) || ...
    (o2 == 0 && onSegment(p1, p4, p2)) || ...
    (o3 == 0 && onSegment(p3, p1, p4)) || ...
    (o4 == 0 && onSegment(p3, p2, p4));
end

function onSeg = onSegment(p, q, r)
onSeg = q(1) <= max(p(1), r(1)) && q(1) >= min(p(1), r(1)) && ...
    q(2) <= max(p(2), r(2)) && q(2) >= min(p(2), r(2));
end
```

Código A.5 Función calcula_fronteras.

test_de_paternidad_MAIN

En esta sección se muestra la implementación mostrada en la sección 2.2.4, aplicando los siguientes preceptos:

1. Todos los nodos que pertenecen al polígono al que pertenezca x son potenciales padres. Es por ello que,
2. todos los nodos pueden ser padres para si mismos, dado que ellos son parte de la cadena de la que son ancestros.
3. Si *nodopropuesto* y x pertenecen a diferentes polígonos, solo podrán ser padres los que se encuentren en la cara oculta del polígono de *nodopropuesto* o bien la tangente al polígono ubicada más en el sentido de las agujas del reloj respecto a x.

La primera sentencia *if* aplica una excepción que se produce para el recinto exterior, en el cual ninguno de los nodos puede ser padre para un nodo externo al recinto, sólo son padres válidos cuando son ancestros, conformando la propiedad 2 mencionada anteriormente, pero esto se ha modificado fuera de la llamada a la función.

La sentencia del *elseif* aplica la primera propiedad citada, validando todos los nodos pertenecientes al primer polígono.

Para la tercera propiedad, es necesario extraer todas los nodos que conforman los vértices del polígono propuesto como padre. Se identifican las tangentes utilizando las propiedades ya mencionadas de los productos vectoriales, véase la sección 2.3.2. Para el caso de polígonos cóncavos es posible que por como se plantee la geometría entre las tangentes exteriores, haya tangentes a las concavidades, se identifican revisando el número de positivos localizados y ordenándolos angularmente. Una vez detectadas las tangentes exteriores se reordena el vector de manera que una de las tangentes queda en primera posición del vector, se identifica con un producto vectorial cual es la tangente exterior ubicada más en el sentido de las agujas del reloj y esa tangente y los nodos que la preceden hasta la siguiente tangente no inclusive se marcan como padres posibles. El resto de nodos son entonces, necesariamente padres no válidos.

```
1  function [EVG]=test_de_paternidad_MAIN(nodox,nodopropuestopadre,EVG)
2
3  % 'Comprobación nodopropuestopadre desde nodox'
4
5  npolnodopropuestopadre=EVG.nodo2poligono(nodopropuestopadre); %Se identifica el poligono del
       padre
6  npolnodox=EVG.nodo2poligono(nodox);
7
8  if ismember(nodopropuestopadre,[1,2,EVG.N-1,EVG.N])
9      padresnoposiblespol=[1,2,EVG.N-1,EVG.N];
10     for jj=1:length(padresnoposiblespol)
11         nodonopos=padresnoposiblespol(jj);
12         EVG.MATposiblepadre(nodox,nodonopos)=0;
13     end
14 elseif npolnodox==npolnodopropuestopadre
15     npol=npolnodopropuestopadre; %Se identifica el poligono del padre
16     listaextendaristas=EVG.poligono2nodo(npol,:); %Se extraen las aristas
17     indcero=find(listaextendaristas==0,1);
18     listaristas=listaextendaristas(1:indcero-1); %Se limpian los ceros
19     padresposiblespol=listaristas;
20
21     for jj=1:length(padresposiblespol)
22         nodopos=padresposiblespol(jj);
23         EVG.MATposiblepadre(nodox,nodopos)=1;
24     end
25 else
26
27     npol=npolnodopropuestopadre; %Se identifican el poligono del padre
28     listaextendaristas=EVG.poligono2nodo(npol,:); %Se extraen las aristas
29     indcero=find(listaextendaristas==0,1);
30     listaristas=listaextendaristas(1:indcero-1); %Se limpian los ceros
31
32     seg_aristas=[listaristas',zeros(length(listaristas),1)];
33
34     p4=EVG.WPs(nodox,:)';
```

```
35
36      %Este procedimiento es similar al que se usa en la rutina de
37      %triangulación, se identifican las tangentes del polígono a nodox
38      for jj=1:length(listaristas)
39
40          if jj==1
41              %Se revisa el producto vectorial de los nodos
42              p1=EVG.WPs(listaristas(end),:)';
43              p2=EVG.WPs(listaristas(1),:)';
44              p3=EVG.WPs(listaristas(2),:)';
45          elseif jj==length(listaristas)
46              p1=EVG.WPs(listaristas(end-1),:)';
47              p2=EVG.WPs(listaristas(end),:)';
48              p3=EVG.WPs(listaristas(1),:)';
49          else
50              p1=EVG.WPs(listaristas(jj-1),:)';
51              p2=EVG.WPs(listaristas(jj),:)';
52              p3=EVG.WPs(listaristas(jj+1),:)';
53          end
54
55          v1=p1-p4;
56          v2=p2-p4;
57          v3=p3-p4;
58
59          prodvec1=v1(1)*v2(2)-v1(2)*v2(1);
60          prodvec3=v3(1)*v2(2)-v3(2)*v2(1);
61
62          if prodvec1*prodvec3>0
63              seg_aristas(jj,2)=1; %Borde de nodo
64          else
65              seg_aristas(jj,2)=0;
66          end
67      end
68
69      %Ahora seg aristas tiene dos valores igual a 1 que son tangentes,
70      %reorganizamos el vector
71      indunos=find(seg_aristas(:,2)==1);
72      if length(indunos)~=2
73          %Esto puede verse debido a recintos concavos
74          pA=EVG.WPs(nodox,:);   %Actua nodox
75          pB=EVG.WPs(EVG.N,:); %Actua nodoy
76          vA=pB-pA;
77          listanodouno=[seg_aristas(indunos,1),zeros(length(indunos))];
78
79          for jj=1:length(indunos)
80              nodouno=listanodouno(jj);
81              pC=EVG.WPs(nodouno,:); %Actua nodo a ordenar
82              vB=pC-pA;
83              ang=acos(dot(vB,vA)/(norm(vB)*norm(vA)));
84              prodvec=vB(1)*vA(2)-vB(2)*vA(1);
85
86              if prodvec<0
87                  %Se busca ordenar rotando desde x y
88                  %empezando desde y
89                  ang=2*pi-ang;
90              end
91              listanodouno(jj,2)=ang;
92          end
93
94          listanodouno=sortrows(listanodouno,2);
95
96          for jj=2:length(listanodouno(:,1))-1
97              %Se desechan los nodos que no sean los que pertenecen a los
98              %extremos
99              seg_aristas(seg_aristas(:,1)==listanodouno(jj),2)=0;
100         end
101         indunos=find(seg_aristas(:,2)==1,1);
102     else
103         indunos=find(seg_aristas(:,2)==1,1);
104     end
105
```

```
106
107
108     vectoraristas=[seg_aristas(indunos:end,:);seg_aristas(1:indunos-1,:)];
109
110     indunos=find(vectoraristas(:,2)==1);
111
112     p1=EVG.WPs(vectoraristas(indunos(1),1),:)';
113     p2=EVG.WPs(vectoraristas(indunos(2),1),:)';
114
115     v1=p1-p4;
116     v2=p2-p4;
117
118     prodvec1=v1(1)*v2(2)-v1(2)*v2(1);
119
120     %por definición, uno va a quedar más CW que el otro, buscaremos cual es
121     %cual y a partir de ahí se buscan los valores factibles para nodos padres
122
123     if prodvec1>0
124         %p2 está por encima de p1 (lead de indunos(1))
125         padresposiblespol=vectoraristas(indunos(1):(indunos(2)-1),1);
126         padresnoposiblespol=vectoraristas(indunos(2):end,1);
127
128         for jj=1:length(padresposiblespol)
129             nodopos=padresposiblespol(jj);
130             EVG.MATposiblepadre(nodox,nodopos)=1;
131         end
132
133         for jj=1:length(padresnoposiblespol)
134             nodonopos=padresnoposiblespol(jj);
135             EVG.MATposiblepadre(nodox,nodonopos)=0;
136         end
137     else
138         %p1 está por encima de p2 (lead de indunos(2))
139         padresposiblespol=vectoraristas(indunos(2):end,1);
140         padresnoposiblespol=vectoraristas(indunos(1):(indunos(2)-1),1);
141
142         for jj=1:length(padresposiblespol)
143             nodopos=padresposiblespol(jj);
144             EVG.MATposiblepadre(nodox,nodopos)=1;
145         end
146
147         for jj=1:length(padresnoposiblespol)
148             nodonopos=padresnoposiblespol(jj);
149             EVG.MATposiblepadre(nodox,nodonopos)=0;
150         end
151     end
152 end
153 end
```

Código A.6 Función test_de_paternidad_MAIN.

cortapuntos

Una forma más de calcular intersecciones, esta vez resolviendo un sistema de ecuaciones con el método de Cramer empleando las formulación pendiente-punto de las rectas. Este método presenta el inconveniente de que calcula el punto de intersección independientemente de si se encuentra dentro o fuera de los segmentos de las rectas, por lo que se verifica que el punto de intersección se encuentra entre ambas con una sentencia *if*.

```
1 %% Función Comprueba visibilidad aristas (Intersección)
2
3 function [inter,xinter]=cortapuntos(line1,line2)
4 %Las lineas vendrán dadas en el formato
5 % [x1, x2
6 % ;y1,y2]
7
8 %line1 es la línea que une N con el punto de la secuencia a comprobar
```

```
9  %line2 es la línea con la que se busca que interseque (cualquiera de las aristas de los polí
       gonosconsiderados)
10
11 x1=line1(1,1);
12 y1=line1(2,1);
13 x2=line1(1,2);
14 y2=line1(2,2);
15 %Por simplicidad vamos a forzar
16
17 if x1>x2
18     xaux=x2;
19     yaux=y2;
20
21     x2=x1;
22     y2=y1;
23
24     x1=xaux;
25     y1=yaux;
26
27 end
28
29 x3=line2(1,1);
30 y3=line2(2,1);
31 x4=line2(1,2);
32 y4=line2(2,2);
33
34 m1=(y2-y1)/(x2-x1);
35
36
37 if x3>x4
38     xaux=x4;yaux=y4;
39     x4=x3;y4=y3;
40
41     x3=xaux;
42     y3=yaux;
43
44 end
45
46 m2=(y4-y3)/(x4-x3);
47
48 if m1~=m2
49     xinter=(y3-y1+m1*x1-m2*x3)/(m1-m2);
50
51     %Esta intersección podría provocarse tanto antes como después del
52     %segmento que nos interesa comprobar su visibilidad por lo que habría
53     %que comprobar si la intersección corta la visibilidad.
54
55     if isnan(xinter)
56         warning('NaN en intersección')
57     end
58
59     if xinter>=x1 && xinter<=x2 && xinter>=x3 && xinter<=x4%Para que sea una intersección real
        tiene que dentro de als dos rectas, cualquiera de las condiciones saca del rango
60         % El punto está dentro del margen de las rectas, es que la
61         % intersección está produciendose en el rango de interés de las
62         % rectas
63         inter=1; %Intersección
64     else
65         %En caso de quese vaya del límite de alguna, no se ha producido.
66         inter=0;
67     end
68 else
69     %Si son iguales, es decir que son paralelas por lo que no puede haber punto de corte
70     inter=0;
71     xinter=nan(1);
72     % 'Bandera2'
73 end
74
75 end
```

Código A.7 Función cortapuntos.

Visible_DEF

Una de las rutinas más importantes de todo el código, su propósito es el de establecer si la visión entre un punto del embudo y el nodo *ii* son visibles o ven su visión interrumpida. Para ello se establece un sistema de *switch-case* que va comprobando cada vez con mayor nivel de profundidad si el nodo es potencialmente visible o no. De esta manera pueden descartarse pasos dentro de esta rutina con mayores costes computaciones, como por ejemplo inter_visible_1.

Para ello se hacen las siguientes comprobaciones en orden, siempre y cuando el caso se mantenga en $'proceso'$:

1. Se revisa que el cono se encuentre dentro del cono de visión, si no lo está, no es visible dentro de ese cono. en el caso de que si lo fuera, por como se ha planteado la triangulación, en las anteriores o siguientes iteraciones de las bases de *SPLIT* definidas por *vectorrecorre* se encontraría.
2. Se revisa si el nodo se encuentra detrás de su padre en el árbol inferior, de ser así, es necesariamente no visible y además el modo de salida sería $'fueraconoi1'$, como se mencionó en la sección del capítulo 2 correspondiente. Si no bloquea su visión potencialmente, entonces no es su padre en el árbol inferior, debido a la definición de embudo presentada.
3. Se revisa con *inter_visible_*1 si existen cortes con las aristas asociadas a los nodos con el resto de familias del embudo, o bien con las aristas asociadas a los polígonos cuyos embudos pertenezcan a la familia. Este paso es el más costoso computacionalmente pero el más efectivo a la hora de localizar cortes que pudieran encontrarse a causa de ausencia de embudos en la familia.
4. Se revisa si el padre en el árbol superior bloquea la visibilidad. Se identifica este nodo usando la función sucesores. Este método no es infalible, por lo que no se usa como criterio directo para definir la visibilidad, sino como soporte a *inter_visible_*1. Esto se debe a que no existen garantías de que si el polígono al que pertenece el padre en el árbol superior es pequeño, el nodo de estudio no sea visible a la derecha del ya mencionado polígono, se adjunta la figura A.2 como soporte visual, que presenta un caso de ambigüedad de padre de embudo, que provocaría un falso negativo. En el artículo original se propone que la recurrencia de *SPLIT* debería encontrarlos al aplicar *SPLIT* a los nodos del ala derecha, pero en las pruebas del algoritmo implementado en geometrías complejas esta llamada pudiera no llegar a darse si faltara algún nodo. Con objeto de mantener baja la complejidad, se ha observado que era menos costoso computacionalmente comprobar los cortes que garantizar una familia completa, hecho comentado en las conclusiones del capítulo 3.

Si todas estas condiciones se cumplen satisfactoriamente el nodo se valida como visible.

```
1  function [VIS,modosalida,EVG]=visible_DEF(EVG,familiaembudo,indicevis,ii,nodox,nodoy)
2  caso='proceso';%Variable que gestiona el switch case
3
4  % Se comprueba que el nodo no pertenezca a la base del embudo
5  if indicevis==1 || indicevis==length(familiaembudo) %|| EVG.Visibilidad(ii,familiaembudo(
       indicevis,1))==1
6      caso='nodobase';
7  end
8
9  % Se comprueba que se encuentre dentro del cono de Visibilidad
10 if strcmp(caso,'proceso')
11     %Comprobamos que se encuentre dentro del arco que forman x-v-y, si no está
12     %dentro, es necesariamente no visible.
13
14     p1=EVG.WPs(familiaembudo(1,1),:)';                          %Punto x
15     p2=EVG.WPs(familiaembudo(indicevis,1),:)';                  %Punto vis
16     p3=EVG.WPs(familiaembudo(length(familiaembudo(:,1)),1),:)'; %Punto y
17     p4=EVG.WPs(ii,:)';                                          %Punto v/ii
```

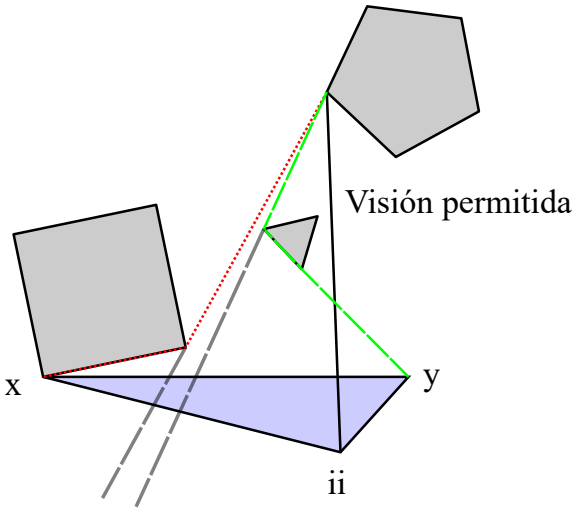


Figura A.2 Ejemplo de caso de ambigüedad de embudo.

```
18
19      v1=p1-p4;
20      v2=p2-p4;
21      v3=p3-p4;
22
23      prodveca=v1(1)*v2(2)-v1(2)*v2(1);
24      prodvecb=v3(1)*v2(2)-v3(2)*v2(1);
25
26      if prodveca*prodvecb>0
27          caso='fuera';
28          if prodveca>0
29              %Es demostrable visualmente que está fuera del cono por la
30              %izquierda, siempre y cuando no se haya comprobado en una
31              %iteración anterior que si era visible
32              caso='fueraconoi1';
33          end
34      end
35  end
36
37  % Se comprueba que el padre en el árbol inferior de nodoVIS no bloquea su
38  % visibilidad
39  if strcmp(caso,'proceso')
40      p1=EVG.WPs(familiaembudo(indicevis,2),:)';                  %Punto padre vis
41      p2=EVG.WPs(familiaembudo(indicevis,1),:)';                  %Punto vis
42      p4=EVG.WPs(ii,:)';                                          %Punto v/ii
43
44      v1=p1-p4;
45      v2=p2-p4;
46
47      prodvec=v2(1)*v1(2)-v2(2)*v1(1);
48      if prodvec<0
49          %Es decir, si v2 tiene que rotar CW, es que padrevis bloquea su
50          %visibilidad
51          caso='fueraconoi1';
52      end
53  end
54
55  %  Se comprueba que no haya cortes con las aristas pertenecientes a los
56  %  polígonos que tomen parte de la familia
57  if strcmp(caso,'proceso')
58      if EVG.Caminos_prohibidos(familiaembudo(indicevis,1),ii)~=1%si no es un camino prohibido se
       computa directamente
59          [VIS,~,EVG]=inter_visible_1(indicevis,familiaembudo,EVG,ii);
60      else
61          VIS=false;
62          caso='inter';
63      end
```

```
64
65      if VIS==true
66          caso='nocorte';
67      else
68          %Se revisa de forma adicional si está oculto detrás de un nodo que le postcede
69          posnodobloq=calculaCW_CCW('CCW', EVG.Matrizorden(familiaembudo(indicevis,1),:), EVG.Mat_
        ind_pos_enMatrizorden(familiaembudo(indicevis,1),:),familiaembudo(indicevis,2),ii,
        familiaembudo(1:end,:));
70
71          p1=EVG.WPs(familiaembudo(indicevis,1),:)';
72          p2=EVG.WPs(posnodobloq,:)';
73          p3=EVG.WPs(ii,:)';                                          %Punto v/ii
74
75          v1=p1-p3;
76          v2=p2-p3;
77          prodvec=v1(1)*v2(2)-v1(2)*v2(1);
78
79          if prodvec>0
80              p2=perpporpunto(EVG,nodox,nodoy,ii);
81
82              v1=p1-p3;
83              v2=p2-p3;
84              prodvec=v1(1)*v2(2)-v1(2)*v2(1);
85
86              if prodvec>0
87                  %lo que corta la visibilidad
88                  p1=EVG.WPs(familiaembudo(indicevis,1),:)';
89                  p2=EVG.WPs(familiaembudo(indicevis,2),:)';
90                  p3=EVG.WPs(familiaembudo(familiaembudo(:,1)==familiaembudo(indicevis,2),2),:)';
                                                %Punto v/ii
91
92                  v1=p1-p3;
93                  v2=p2-p3;
94                  prodvec=v1(1)*v2(2)-v1(2)*v2(1);
95
96                  if prodvec<0 %Para que se de el caso de igualdad, tendría que haber saltado la
        condición de fueraconoi1 del principio
97                      caso='fueraconoi1';
98                  else
99                      caso='inter';
100                 end
101             else
102                 % caso='fueraconoi2';
103                 caso='inter';
104             end
105         else
106             caso='fueraconoi1';
107         end
108     end
109 end
110
111 switch caso
112
113     case 'nodobase'
114         VIS=true;
115         modosalida='VIS';
116     case 'fuera'
117         VIS=false;
118         modosalida='fueraconod';
119     case 'fueraconoi1'
120         VIS=false;
121         modosalida='fueraconoi1';
122     case 'fueraconoi2'
123         VIS=false;
124         modosalida='fueraconoi1';
125     case 'nocorte'
126         VIS=true;
127         modosalida='VIS';
128     case 'inter'
129         VIS=false;
130         modosalida='inter';
```

```
131     otherwise
132         error('Fallo Visibilidad')
133 end
134 end
```

Código A.8 Función Visible_DEF.

Se adjunta la función perpporpunto, que como su nombre indica para una recta dada, calcula la intersección entre la recta y la línea que pasa por un punto externo con pendiente perpendicular a la recta.

```
1  function [p_interseccion]=perpporpunto(EVG,nodox,nodoy,ii)
2
3  p1=EVG.WPs(nodox,:)';
4  p2=EVG.WPs(nodoy,:)';
5  v=EVG.WPs(ii,:)';
6  % Vector director de la línea l1
7  d = p2 - p1;
8
9  % Vector normal a la línea l1 (perpendicular)
10 n = [-d(2), d(1)]; % Este es el vector normal
11
12 % Ecuaciones paramétricas de la línea l1 y de la normal
13 % l1: r(t) = p1 + t * d
14 % Normal: r(s) = v + s * n
15
16 % Resolver el sistema de ecuaciones:
17 % p1(1) + t*d(1) = vx + s*n(1)
18 % p1(2) + t*d(2) = vy + s*n(2)
19
20 A = [d(1), -n(1); d(2), -n(2)];
21 b = [v(1) - p1(1); v(2) - p1(2)];
22
23 % Resolver para t y s
24 sol = A\numberstyleb;
25
26 % Encontrar el punto de intersección usando t
27 t = sol(1);
28 p_interseccion = p1 + t * d;
29
30 end
```

Código A.9 Función perpporpunto.

inter_visible_1

inter_visible_1 es una subrutina dependiente de Visible_DEF que comprueba para los nodos de la familia, si existen cortes con las aristas que pertenecen a ella, usando la misma estrategia de detección de cortes empleada para la triangulación. Existen dos versiones posibles de esta función, uno que aplica esta definición directamente, y otra que extiende esta familia empleando todos los nodos asociados a los polígonos asociados a los nodos de la familia.

La primera estrategia funcionó perfectamente para el barrido de los recintos convexos pero para el caso de estudio del método de Lee, [5], fue necesario aplicar la segunda. Ambas mantienen la complejidad en valores similares al ser aplicadas según la naturaleza de la geometría, por lo que se adjuntan ambas. En la versión estándar del código se añaden todos los vértices asociados al polígono del que *ii* forma parte, a los ya presentes en la familia.

```
1 function [VIS,modosalida,EVG]=inter_visible_1(indicevis,familiaembudo,EVG,ii)
2 nodo2poligono=EVG.nodo2poligono;
3 poligono2nodo=EVG.poligono2nodo;
4
5 nodox=familiaembudo(1,1);
6 nodoy=familiaembudo(end,1);
```

```
nodoVIS=familiaembudo(indicevis,1);
if nodox>nodoy
    nodo=nodox;
    familiaembudo=EVG.nodosTOT_famvy_glob{nodo};
else
    nodo=nodoy;
    familiaembudo=EVG.nodosTOT_famxv_glob{nodo};

end

%Se añaden el resto de nodos del polígono a la lista de comprobación

listapol=[];

poljj=nodo2poligono(ii);
if poljj==0
    N=length(nodo2poligono(:,1));
    aristavector=[1,2,N-1,N,0];                          %se saca el vector que contiene todas las aristas de EVG, atención contiene 0s
else
    aristavector=poligono2nodo(poljj,:);        %se saca el vector que contiene todas las aristas de EVG, atención contiene 0s
end
naristas=find(aristavector==0,1)-1;                      %Se identifica el números de aristas diferentes de 0
aristas=aristavector(1:naristas);                        %se extrae un vector de aristas limpio
familiaembudo=[familiaembudo;aristas'];



[~,indiceunico]=unique(familiaembudo(:,1),'stable');
familiaembudo=familiaembudo(indiceunico,:);
indicevis=find(familiaembudo(:,1)==nodoVIS,1);

VIS=true;
modosalida='';

for jj=[1:(indicevis-1),(indicevis+1):length(familiaembudo)]
    poligono=EVG.nodo2poligono(familiaembudo(jj,1));
    inter1=0;
    inter2=0;

    if poligono==0
        N=length(EVG.nodo2poligono(:,1));
        aristavector=[1,2,N-1,N,0];                      %Se saca el vector que contiene todas las aristas de EVG, atención contiene 0s
    else
        aristavector=EVG.poligono2nodo(poligono,:);      %se saca el vector que contiene todas las aristas de EVG, atención contiene 0s
    end

    naristas=find(aristavector==0,1)-1;                  %Identificamos el números de aristas diferentes de 0
    aristas=aristavector(1:naristas);                    %se extrae un vector de aristas limpio
    indpoljj=find(aristas==familiaembudo(jj,1),1);       %buscamos en que posición se ubica el valor del nodo jj en las aristas


    if indpoljj~=1 && indpoljj~=naristas
        indarista1=[indpoljj+1,indpoljj];
        indarista2=[indpoljj-1,indpoljj];

    elseif indpoljj==1
        indarista1=[indpoljj+1,indpoljj];
        indarista2=[naristas,indpoljj];

    elseif indpoljj==naristas
        indarista1=[indpoljj-1,indpoljj];
        indarista2=[1,indpoljj];
    end
```

```
70
71      aristaincidente1=aristas(indarista1); %Estos valores están en nodos
72      aristaincidente2=aristas(indarista2); %Estos valores están en nodos
73
74      line1=[EVG.WPs(ii,1),EVG.WPs(familiaembudo(indicevis,1),1);...
75          EVG.WPs(ii,2),EVG.WPs(familiaembudo(indicevis,1),2)];
76      line2=[EVG.WPs(aristaincidente1(1),1),EVG.WPs(aristaincidente1(2),1);...
77          EVG.WPs(aristaincidente1(1),2),EVG.WPs(aristaincidente1(2),2)];
78
79      [inter1,xinter1]=cortapuntos(line1,line2);
80
81      line2=[EVG.WPs(aristaincidente2(1),1),EVG.WPs(aristaincidente2(2),1);...
82          EVG.WPs(aristaincidente2(1),2),EVG.WPs(aristaincidente2(2),2)];
83      [inter2,xinter2]=cortapuntos(line1,line2);
84
85      %inter vale 0 cuando no hay intersección
86
87      if  ~(inter1==0 && inter2==0)
88          if inter1==1 &&  ~(abs(xinter1-EVG.WPs(ii,1))<1e-3) &&  ~(abs(xinter1-EVG.WPs(
        familiaembudo(indicevis,1),1))<1e-3)%Comprobamos que el punto de intersección no sea el nodo
         en si mismo
89              %si existe una intersección que no sea ii o nodo vis es necesariamente no visible
90              VIS=false;
91              % line1
92              % line2
93              % xinter1
94              modosalida='inter1_intervisible';
95              break
96          elseif inter2==1 && ~(abs(xinter2-EVG.WPs(ii,1))<1e-3)&&  ~(abs(xinter2-EVG.WPs(
        familiaembudo(indicevis,1),1))<1e-3)
97              VIS=false;
98              modosalida='inter2_intervisible';
99              break
100         end
101     end
102
103     if jj==length(familiaembudo)
104         VIS=true;
105         modosalida='';
106     end
107 end
108
109 end
```

Código A.10 Función inter_visible_1.

A continuación se encuentra la versión alternativa de la extensión de la familia de embudos, nótese el bucle que recorre la familia comprobando si los nodos asociados al polígono han sido ya añadidos, comprobándolo con *listapol*.

```
1  function [VIS,modosalida,EVG]=inter_visible_1(indicevis,familiaembudo,EVG,ii)
2  nodo2poligono=EVG.nodo2poligono;
3  poligono2nodo=EVG.poligono2nodo;
4
5  nodox=familiaembudo(1,1);
6  nodoy=familiaembudo(end,1);
7  nodoVIS=familiaembudo(indicevis,1);
8  if nodox>nodoy
9  nodo=nodox;
10 familiaembudo=EVG.nodosTOT_famvy_glob{nodo};
11 % familiaembudo2=EVG.nodosTOT_famxv_glob{nodo};
12
13 else
14 nodo=nodoy;
15 familiaembudo=EVG.nodosTOT_famxv_glob{nodo};
16 % familiaembudo2=EVG.nodosTOT_famvy_glob{nodo};
17
18 end
19
```

```
%Se añaden el resto de nodos del polígono a la lista de comprobación

listapol=[];
for jj=1:length(familiaembudo)
poljj=nodo2poligono(familiaembudo(jj));
indjj=find(listapol==poljj,1);

    if isempty(indjj)
       listapol=[listapol;poljj];


        if poljj==0
            N=length(nodo2poligono(:,1));
            aristavector=[1,2,N-1,N,0];                          %se saca el vector que contiene
     todas las aristas de EVG, atención contiene 0s
        else
            aristavector=poligono2nodo(poljj,:);        %se saca el vector que contiene todas las
      aristas de EVG, atención contiene 0s
        end
        naristas=find(aristavector==0,1)-1;                       %Identificamos el números de
     aristas diferentes de 0
        aristas=aristavector(1:naristas);                         %se extrae un vector de aristas
     limpio
        familiaembudo=[familiaembudo;aristas'];

    end

end
    [~,indiceunico]=unique(familiaembudo(:,1),'stable');
    familiaembudo=familiaembudo(indiceunico,:);
    indicevis=find(familiaembudo(:,1)==nodoVIS,1);
```

Código A.11 Versión alternativa de la ampliación familiaembudo para método de Lee.

calculaCW_CCW:sucesores

Se utilizan los nodos ordenados unos alrededor de otros que se obtuvieron en el preprocesado para convertir este cálculo en una consulta a una matriz y una búsqueda del siguiente nodo de la familia, filtrando fuera del subvector ordenado los nodos no presentes en la familia de embudos.

```
%% Funcion Sucesores CW/CCW

% nodoorden es el vector fila asociado al nodo i respecto al cual rotaremos
% para buscar el sucesor, tanto CW como CCW del nodo j incidente (nodoinc) en el.
% invnodoorden es el vector fila asociado al nodo i de Mat_ind_pos_enMatrizorden.
% maxnodo es la manera de filtrar los nodos que todavía no se han
% incorporado a la región P, estos no participarían en el cálculo de los
% embudos. Sería el nodo que incorporamos a la región triangulada.


function sucesor=calculaCW_CCW(modo, nodoorden, invnodoorden, nodoinc,maxnodo,familiaembudo)
% indicenodoinc=invnodoorden(nodoinc); ha quedado obsoleto por culpa de que
% solo hay que usar los valores de pertenencientes a familiaembudo que son
% todos menores a maxnodo pero no son todos los menores a maxnodo
% nodoorden
[~, indicefiltrado]= ismember(nodoorden, familiaembudo(:,1));

nodoorden=nodoorden(indicefiltrado>0);
indicenodoinc=find(nodoorden==nodoinc,1);

flag=0;

if ~isnan(indicenodoinc)
    if strcmp(modo,'CCW')
        while flag==0
            if indicenodoinc==length(nodoorden)
                indicenodoinc=1;
                if nodoorden(indicenodoinc)<maxnodo
```

```
                    flag=1;
                end
            else
                indicenodoinc=indicenodoinc+1;
                % [nodoorden(indicenodoinc),maxnodo]
                if nodoorden(indicenodoinc)<maxnodo
                    flag=1;
                end
            end
        end
    elseif strcmp(modo,'CW')
        while flag==0
            if indicenodoinc==1
                indicenodoinc=length(nodoorden);
                if nodoorden(indicenodoinc)<maxnodo
                    flag=1;
                end
            else
                indicenodoinc=indicenodoinc-1;
                if nodoorden(indicenodoinc)<maxnodo
                    flag=1;
                end
            end
        end
    else
        error('Wrong input, please check modo variable input in calculaCW_CCW.')
    end
    sucesor=nodoorden(indicenodoinc);

elseif strcmp('CCW',modo)
    %Cualquier nodo cuyo padre en el lower tree sea si mismo, su padre en
    %el arbol superior debe ser y
    sucesor=familiaembudo(end,1);
else
    sucesor=nodoinc;
end

end
```

Código A.12 Función calculaCW_CCW.

checkIfInsideTriangle

Esa rutina implementa el cálculo para comprobar si para un linea de visión a la que se le va a aplicar una operación de *SPLIT*, el cono asociado está vacío.

Para ello implementa la estrategia desarrollada en la figura 2.23. Para comprobar si existe un punto entre el triángulo formado por una linea de visión y un tercer punto externo o interno, se calculan las áreas utilizando la fórmula del determinante. A continuación se compara si las áreas que emplean el punto externo sumadas proporcionan sumadas valores mayores al triángulo original planteado, el punto se encontraría fuera del cono. De lo contrario de ser las áreas iguales, o ligeramente menores debido a errores numéricos, el punto se encontraría dentro del cono.

```
function isInside = checkIfInsideTriangle(ii, nodoa, nodob, nodocheck, EVG)

% Calcular las áreas
% Área total del triángulo ii-nodoa-nodob
areaTotal = triangleArea(ii, nodoa, nodob, EVG);

% Área del subtriángulo formado por ii, nodoa y nodocheck
area1 = triangleArea(ii, nodoa, nodocheck, EVG);

% Área del subtriángulo formado por ii, nodob y nodocheck
area2 = triangleArea(ii, nodob, nodocheck, EVG);

% Área del subtriángulo formado por nodoa, nodob y nodocheck
area3 = triangleArea(nodoa, nodob, nodocheck, EVG);
```

```
15
16 % Verificar si la suma de las áreas de los subtriángulos es igual al área total, usando
       tolerancia relativa
17 isInside = (abs(areaTotal - (area1 + area2 + area3)))/areaTotal < 1e-5;
18 end
19
20 function area = triangleArea(n1, n2, n3, EVG)
21 % Obtener las coordenadas de los tres nodos
22 P1 = EVG.WPs(n1, :);
23 P2 = EVG.WPs(n2, :);
24 P3 = EVG.WPs(n3, :);
25
26 % Calcular el área del triángulo usando la fórmula del determinante
27 area = abs((P1(1)*(P2(2)-P3(2)) + P2(1)*(P3(2)-P1(2)) + P3(1)*(P1(2)-P2(2))) / 2);
28 end
```

Código A.13 Función checkIfInsideTriangle.

A.3.3 add_familia2

add_familia2 implementa la ordenación de las familias con el propósito de construirlas y proporcionarlas a calculofamilia. Para ello toma los nodos asociados a la $familiaxv$ o $familiavy$, dependiendo de que nodo de la base de la familia de embudos tenga mayor coordenada x, y por tanto un identificador mayor. Todos estos nodos pasan por un proceso gradual consistente en lo siguiente:

1. Se propone como ancestro el *nodox* y se comprueba bajo una serie de criterios que nodos pueden conectar con él. Iterando con el número de nodos no asignados ordenados en sentido de las agujas del reloj alrededor del padre.
2. Se añaden a la familia inmediatamente después del padre todos los hijos que puede adoptar.
3. Los nodos que se asignaron en la generación anterior son ahora los nuevos padres para el resto de nodos restantes no asociados. Para cada uno de estos padres se prueba con los hijos ordenados respecto al padre propuesto. Se usa el mismo criterio de inserción en la familia. Para evitar que se asignen nodos en cadenas de menor número de nodos y no las que presenten una mayor amplitud angular, es decir el camino que se desvía menos entre *nodox* y el nodo de estudio, se evalúa si existen padres más favorables entre los nodos no asignados antes de de añadirlo a la familia con el padre propuesto.
4. Este bucle de búsqueda de relaciones entre nodos y posibles padres continúa hasta que en una iteración no se haya asignado ningún padre. Empleando para ello la bandera $flag\_pruebagracia$, al empezar la iteración con la nueva generación de padres se baja, y si se asigna uno se levanta.

Existe una repesca de nodos que han quedado por asignar para los que se encontró un posible mejor padre, pero finalmente ese padre no pudo ser añadido a la familia o realmente una vez añadido a la cadena y comprobado con todos los criterios, se decretó que no era un padre válido. Esta consiste en el mismo algoritmo de antes pero con la capacidad de activar o desactivar el criterio de padres válidos, el modo de ejecución final. En este se prueba con todos los padres asignados a la familia con los nodos restantes por asignar. Este modo puede configurarse según las necesidades del usuario, pero disminuye el rendimiento computacional.

Aquí entra en juego lo mencionado en la explicación de inter_visible_1, garantizar que las familias se encuentren enteras tiene su coste computacional, por lo que encontrar un equilibrio con inter_visible_1 es clave. Repartiendo la carga de trabajo entre ambas, facilitando que se repartan la complejidad y evitando escalados inadmisibles.

La matriz *seguimientonodos* lleva, como su propio nombre indica el seguimiento de los nodos del conjunto de nodos originalmente propuestos, en ella se registra la siguiente información:

- 1ª columna: Identificador del nodo.
- 2ª columna: Si el nodo ha sido asignado a familia. Por ejemplo $seguimientonodos(:,2) == 0$ es el conjunto de nodos que no han sido aún asignados.

De forma adicional, se mantiene un registro de aquellos nodos que presentan un mejor padre, para comprobar que de realizarse una asignación en siguientes iteraciones se haga a estos nodos. Esta matriz es $MAT futurasgeneraciones$, que es una matriz que se inicia como $zeros(length(vecaux),1)$, es decir una matriz columna de ceros con correspondencia a todos los nodos entregados a add_-familia2. Conforme se identifican los posibles padres futuros, se almacenan en esta matriz usando la función extiendematriz. Que dependiendo del número de columnas de la matriz y el número de componentes del vector fila que se computa con los futuros mejores padres, incorpora estos datos respetando las reglas de concatenación de Matlab.

Existen varios criterios que se usan para discernir si un nodo puede ser asignado o no en la matriz:

1. Criterios asociados a los padres:
   - El padre debe ser válido desde el punto de vista de *nodox*, aquí entra en juego la matriz $EVG.MAT posiblepadre$, que validará el uso de los nodos computados

2. Criterios asociados a la interacción entre padre e hijo:
   - Debe haber línea de visión directa, anteriormente computada, entre el nodo propuesto como padre y el nodo de estudio.
   - *prodvec*: $nodopropuestopadre$ debe bloquear la visibilidad de *nodox* al nodo de estudio $nodoVIS$, si no, no puede ser su padre. Esto se verifica usando el producto vectorial de *nodox* sobre $nodopropuestopadre$ rotando alrededor de $nodoVIS$, comprobando si este es negativo.
   - *prodvec*2: La cadena debe ser convexa, para ello se busca que el producto vectorial de $nodoVIS$ sobre *nodox* rotando alrededor de $nodopropuestopadre$ sea mayor que cero.
   - *prodvec*3: $nodopropuestopadre$ debe encontrarse fuera del cono de visión. Esto se verifica haciendo el producto vectorial de *nodox* sobre $nodopropuestopadre$ visto desde *nodoy*, y debe ser negativo.

     A partir de aquí para seguir comprobando las condiciones, todas las anteriores deben cumplirse, esto se ha programado así para ahorrar recursos en comprobaciones adicionales. Si falla una, el resto son innecesarias.
   - *condconvex*: Es la condición extendida de *prodvec*2 cuando $nodopropuestopadre$ no es *nodox*, salvo que en vez de calcular la convexidad con *nodox* se realiza con el padre de $nodopropuestopadre$, por lo que se comprueba si el producto vectorial es positivo.
   - $condarista$: Comprueba que de insertarse un nodo se haga respetando la geometría asociada a $nodopropuestopadre$. Se ordenan en un vector en orden angular alrededor de $nodopropuestopadre$, los nodos consecutivos asociados al polígono al que pertenece junto con $nodoVIS$. Se revisa que $nodoVIS$ de introducirse se haga desde la cara no visible a *nodox* del polígono. Esto sirve en casos como el propuesto en la figura A.3 donde en el primer caso *nodox* es padre válido pero se intenta insertar a la raíz sin respetar la condición de embudo vacío. En el segundo, puede introducirse $nodoVIS$ dado que va tras las aristas asociadas al polígono.
   - $condpadresfuturo$: se desarrollará junto con la función que ejecuta su cálculo. Cumple una doble función, busca si un nodo podría tener mejores padres y si de haberse encontrado en iteraciones anteriores uno, si $nodopropuestopadre$ es de aquellos padres óptimos.

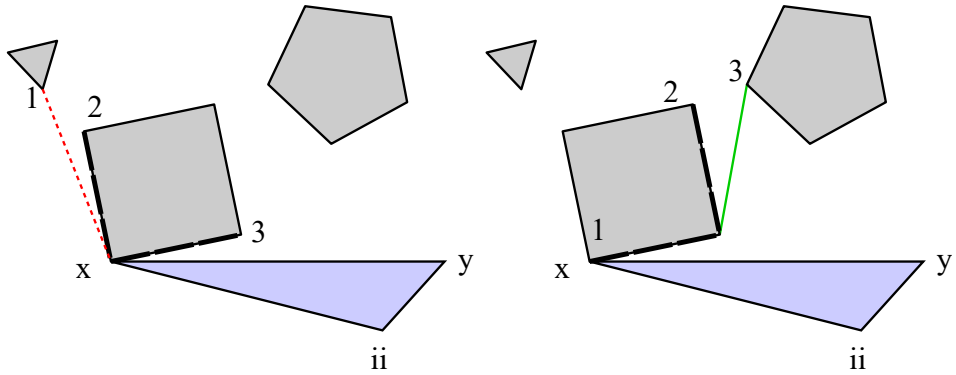


Figura A.3 Ejemplo de ejecución de *condarista*.

Si todas estas condiciones se cumplen satisfactoriamente el nodo se añade a *anyadenodo*, que insertará al final de las comprobaciones de todos los nodos por asignar respecto a *nodopropuestopadre* y la introducirá en la familia en orden de embudo.

Todas las condiciones anteriormente citadas se cumplen en el modo de ejecución final, salvo que con la excepción de que *condpadresfuturo* es desactivable.

```
function [familia]=add_familia2(EVG,vecaux)
%familiavy está vacío
%vecaux es familiaembudo a ordenar

%La segunda columna tiene los valores de los nodos: 0 si no se ha
%procesado, 1 si si
Visibilidad=EVG.Visibilidad;
% Visibilidad(isnan(Visibilidad))=0;
WPs=EVG.WPs;
Consec_aristasrec=EVG.Consec_aristasrec;

seguimientonodos=zeros(length(vecaux(:,1)),3);
seguimientonodos(:,1)=vecaux(:,1);
seguimientonodos(1,2)=1;%Para que haya un padre inicial
familia=[vecaux(1,:),1,0];
seguimientonodos(end,2)=1;%Para evitar que "y" sea usada
MATfuturasgeneraciones=zeros(length(vecaux(:,1)),1);
MAT_posnodosvecaux=zeros(EVG.N,1);%Transforma dato nodo en posición segnodo/vecaux

%Esto es un intento de evitar hacer tantas llamadas para buscar valores en
%seguimientonodos y vecaux

for jj=1:length(vecaux(:,1))
    nodoaux=vecaux(jj,1);
    MAT_posnodosvecaux(nodoaux)=jj;
end

flag_pruebagracia=1;
generacion=2;

p3=WPs(vecaux(1,1),:)'; %Nodox (Padre del padre)
p4=WPs(vecaux(end,1),:)';
pB=p4'; %Actua nodoy

while sum(seguimientonodos(:,2))<length(seguimientonodos(:,2)) && flag_pruebagracia==1 % 1ª Si no
      se han computado todos los nodos continua, 2ª garantiza que no estamos en un bucle infinito
     y desechemos los nodos que no encuentran padre
    flag_pruebagracia=0; %La bandera se levantará si al menos un nodo ha sido añadido

    listanodos=seguimientonodos(seguimientonodos(:,2)==0,1); %Busca todos los nodos que no hayan
     sido computados

```

```
40      listapadres=familia(familia(:,3)==(generacion-1),1)'; %Los nodos computados están guardados
        en orden en familia! buscados en los nodos computados en la generación anterior, esto deberí
        a mejorar el rendimiento
41      listapadres(listapadres==vecaux(end,1))=[];
42
43
44      for jj=1:length(listapadres)
45          nodopropuestopadre=listapadres(jj);
46          p2=WPs(nodopropuestopadre,:)'; %Padre Actual
47
48          anyadenodo=[];
49          padrevalido=EVG.MATposiblepadre(vecaux(1,1),nodopropuestopadre);
50
51
52          if padrevalido==true
53              familia(familia(:,1)==nodopropuestopadre,4)=1;%Se almacena la información de los
        padres válidos para que se pruebe en la siguiente ronda
54              listanodos=generalistanodos(nodopropuestopadre,vecaux(end,1),listanodos,EVG);
55              %Ahora se tiene listanodos ordenado respecto al padre propuesto
56              for kk=1:length(listanodos)
57                  nodoVIS=listanodos(kk);
58                  cond=Visibilidad(nodoVIS,nodopropuestopadre);
59
60                  if seguimientonodos(MAT_posnodosvecaux(nodoVIS),2)==0 && cond %O sea, que no se
        haya introducido ya y sea visible
61                      p1=WPs(nodoVIS,:)'; %Nodo
62
63                      v1=p2-p1;
64                      v2=p3-p1;
65
66                      v3=p1-p2;
67                      v4=p3-p2;
68
69                      prodvec=v1(1)*v2(2)-v1(2)*v2(1);
70                      prodvec2=v3(1)*v4(2)-v3(2)*v4(1);
71
72                      v5=p2-p4;
73                      v6=p1-p4;
74                      prodvec3=v5(1)*v6(2)-v5(2)*v6(1);
75
76                      if prodvec<=0 && prodvec2>=0 && prodvec3<=0
77                          if nodopropuestopadre~=vecaux(1,1)
78                              %Tecnicamente es una condición más desarollada que la
79                              %de prodvec, más genérica pero solo valida para casos
80                              %que padre y abuelo no son iguales
81                              % p1=WPs(nodoVIS,:)'; %Nodo
82                              p2=WPs(nodopropuestopadre,:)'; %Padre Actual
83                              nodoabuelo=familia(familia(:,1)==nodopropuestopadre,2);
84                              pabuelo=WPs(nodoabuelo,:)'; %Nodo abuelo (Padre del padre)
85
86                              v1=p1-p2;
87                              v2=pabuelo-p2;
88                              prodvecx=v1(1)*v2(2)-v1(2)*v2(1);
89
90                              if prodvecx>0
91                                  condconvex=true;
92                              else
93                                  condconvex=false;
94                              end
95
96                          else
97                              %Esta condición no tendría sentido por lo que por
98                              %defecto es verdadera
99                              condconvex=true;
100                         end
101
102                         %Se va a meter una condición adicional, un nodo solo se
103                         %puede añadir si queda CW de la arista
104                         %nodopropuestopadre-siguiente aristapol
105
106                         % nodosconsecpadre=find(EVG.Consec_aristas(nodopropuestopadre,:)==1);
```

```
107
108                         if condconvex==true
109                             nodosconsecpadre=Consec_aristasrec(nodopropuestopadre,:);
110
111                             listasec=nodoVIS;
112
113                             if ~isempty(nodosconsecpadre)
114
115                                 for mm=1:length(nodosconsecpadre)
116                                     if nodosconsecpadre(mm)~=nodoVIS
117                                         listasec=[listasec,nodosconsecpadre(mm)];
118                                     end
119
120                                 end
121
122                             end
123                             %%%%%%%%%% %ordenarlistasec
124                             pA=p2;   %Actua nodopropuestoapdre
125                             vA=pB-pA;
126                             listaauxsec=[listasec',zeros(length(listasec'),1)];
127
128                             for mm=1:length(listasec)
129                                 nodomm=listasec(mm);
130                                 pC=WPs(nodomm,:)'; %Actua nodo a ordenar
131                                 vB=pC-pA;
132                                 ang=acos(dot(vB,vA)/(norm(vB)*norm(vA)));
133                                 prodveck=vB(1)*vA(2)-vB(2)*vA(1);
134
135                                 if prodveck<0
136                                     %Se busca ordenar rotando desde x y
137                                     %empezando desde y
138                                     ang=2*pi-ang;
139                                 end
140                                 listaauxsec(mm,:)=[nodomm,ang];
141                             end
142
143                             listaauxsec=ordenalistasec(listaauxsec);
144                             listasec=listaauxsec(:,1);
145
146                             if length(listasec)==2
147                                 if nodoVIS==listasec(end)
148                                     %Caso D
149                                     condarista=true;
150                                 else %nodoVIS==listasec(1)
151                                     %Caso E
152                                     condarista=false;
153                                 end
154                             elseif length(listasec)==3
155                                 if nodoVIS==listasec(1)
156                                     %CasoA
157                                     condarista=false;
158                                 elseif nodoVIS==listasec(2)
159                                     %Caso B: Este podría verse afectado en
160                                     %recintos no convexos
161                                     condarista=true;
162                                 elseif nodoVIS==listasec(end)
163                                     %caso C
164                                     condarista=true;
165                                 end
166                             else
167                                 listasec
168                                 error('Check condarista: Fallos en Vector coordenadas V')
169                             end
170                         else
171                             condarista=false;
172                         end
173                         %%%%%%%%%%
174                         polnodopropuesto=EVG.nodo2poligono(nodopropuestopadre);
175                         polnodox=EVG.nodo2poligono(vecaux(1,1));
176
177                         indfutgen=MAT_posnodosvecaux(nodoVIS);
```

```
178                         if condarista&&condconvex && nodopropuestopadre ~=vecaux(1,1)&&
    polnodopropuesto~=polnodox&& MATfuturasgeneraciones(indfutgen,1)==0
179                             %Esto es un último check, para evitar que se
180                             %introduzcan nodos antes de tiempo
181                             [condpadrefuturo,posiblespadres]=futurasgeneraciones(EVG,listanodos,
    nodopropuestopadre,nodoVIS,vecaux(1,1));
182                             MATfuturasgeneraciones=extiendematriz(MATfuturasgeneraciones,
    indfutgen,posiblespadres);
183
184                         elseif condarista&&condconvex && nodopropuestopadre ~=vecaux(1,1)&&
    polnodopropuesto~=polnodox&& MATfuturasgeneraciones(indfutgen,1)~=0
185                             %Para el caso de que haya computados nodos, se
186                             %revisará si el nodo que se propone en esta
187                             %iteración es uno de los que se propuso como
188                             %mejor padre
189                             flag=0;
190                             contador=1;
191                             condpadrefuturo=false;
192                             while flag==0
193                                 nodofuturo=MATfuturasgeneraciones(indfutgen,contador);
194
195                                 if nodofuturo==nodopropuestopadre
196                                     %Si el padre coincide con uno de los
197                                     %que se propuso en su momento se da
198                                     %como válido y corta la ejecución
199                                     flag=1;
200                                 end
201                                 contador=contador+1;
202
203                                 if contador<=length(MATfuturasgeneraciones(1,:))
204                                     if MATfuturasgeneraciones(indfutgen,contador)==0
205                                         flag=1;
206                                     end
207                                 else
208                                     %si se ha llegado al final de la
209                                     %matriz, se corta la ejecución del
210                                     %bucle
211                                     flag=1;
212                                 end
213                             end
214                         else
215                             condpadrefuturo=true;
216                         end
217
218                         if condconvex && condarista && condpadrefuturo
219                             flag_pruebagracia=1; %si entra aquí ya hay potencialmente un nodo
    introducible, por lo que habría que continuar
220                             anyadenodo=[anyadenodo;nodoVIS,nodopropuestopadre,generacion,0];%el
    ultimo 0 identifica si es posible padre, en este momento no se ha computado todavía, se hará
     en la siguiente gen
221                             seguimientonodos(MAT_posnodosvecaux(nodoVIS),2)=1;%Lo que significa
    que se ha procesado ese nodo
222                         end
223                     end
224                     %Fin if listandoos
225                 end
226             end
227
228             %Se añaden los nodos del vector anyadenodo
229             if ~isempty(anyadenodo)
230                 %Se añaden los valores
231                 indicepadre=find(familia(:,1)==nodopropuestopadre);
232                 familia=[familia(1:indicepadre,:);anyadenodo;familia(indicepadre+1:end,:)];
233             end
234             %Fin Padrevalido check
235         end
236         %Fin bucle listapadres
237     end
238     generacion=generacion+1;
239
240     %Fin bucle global
```

```
241 end
242
243 % Dado que la condición de salida no es que se asignen todos los nodos, es
244 % posible que queden algunos que pudieron haber sido asignados pero quedaron
245 % pendientes al ser el mejor padre propuesto no válido, o que este nunca se
246 % llegara a asignar, por esto se plantea una segunda ronda.
247
248 flag_pruebagracia=1;
249 flag_padres=0;
250 padresposiblesfuturo=[];
251 % seguimientonodos(isnan(seguimientonodos(:,2)),2)=0;
252
253 MATfuturasgeneraciones=zeros(length(vecaux(:,1)),1);
254 flagmodoejecucionfinal=1;
255
256 p3=WPs(vecaux(1,1),:)';p4=WPs(vecaux(end,1),:)';
257
258 while flag_pruebagracia==1||flagmodoejecucionfinal==1
259     flag_pruebagracia=0;
260     nodospendientes=seguimientonodos(seguimientonodos(:,2)==0,1);
261
262     %En esta ocasión el planteamiento será diferente, se buscará un padre para
263     %cada nodo
264     if flagmodoejecucionfinal==0
265         if flag_padres==0
266             padresposibles=familia(familia(:,4)==1,1);%Se extraen los nodos que pueden ser padres
267             flag_padres=1;
268         else
269             padresposibles=padresposiblesfuturo;
270             padresposiblesfuturo=[];
271         end
272     else
273         padresposibles=familia(familia(:,4)==1,1);%Modo de ejecución final, todos los padres vá
    lidos son ahora posibles
274     end
275
276     for jj=1:length(nodospendientes)
277         nodoVIS=nodospendientes(jj);
278         p1=WPs(nodoVIS,:)';
279         for kk=1:length(padresposibles)
280             nodopropuestopadre=padresposibles(kk);
281             p2=WPs(nodopropuestopadre,:)';
282             cond=Visibilidad(nodoVIS,nodopropuestopadre);
283
284             if cond
285                 v1=p2-p1;v2=p3-p1;
286                 v3=p1-p2;v4=p3-p2;
287                 prodvec=v1(1)*v2(2)-v1(2)*v2(1);
288                 prodvec2=v3(1)*v4(2)-v3(2)*v4(1);
289                 v5=p2-p4;v6=p1-p4;
290                 prodvec3=v5(1)*v6(2)-v5(2)*v6(1);
291
292                 if prodvec<=0 && prodvec2>=0 && prodvec3<=0
293                     if nodopropuestopadre~=vecaux(1,1)
294                         %Tecnicamente es una condición más desarollada que la
295                         %de prodvec, más genérica pero solo valida para casos
296                         %que padre y abuelo no son iguales
297                         % p1=EVG.WPs(nodoVIS,:)'; %Nodo
298                         % p2=EVG.WPs(nodopropuestopadre,:)'; %Padre Actual
299                         nodoabuelo=familia(familia(:,1)==nodopropuestopadre,2);
300                         pabuelo=WPs(nodoabuelo,:)'; %Nodo abuelo (Padre del padre)
301
302                         v1=p1-p2;
303                         v2=pabuelo-p2;
304                         prodvecx=v1(1)*v2(2)-v1(2)*v2(1);
305
306                         if prodvecx>0
307                             condconvex=true;
308                         else
309                             condconvex=false;
310                         end
```

```
311                     else
312                         %Esta condición no tendría sentido si se analiza nodox como padre por lo
    que por
313                         %defecto es verdadera
314                         condconvex=true;
315                     end
316 
317                     %Se va a meter una condición adicional, un nodo solo se
318                     %puede añadir si queda CW de la arista
319                     %nodopropuestopadre-siguiente aristapol
320                     if condconvex==true
321                         nodosconsecpadre=Consec_aristasrec(nodopropuestopadre,:);
322 
323                         listasec=nodoVIS;
324 
325                         if ~isempty(nodosconsecpadre)
326                             for mm=1:length(nodosconsecpadre)
327                                 if nodosconsecpadre(mm)~=nodoVIS
328                                     listasec=[listasec,nodosconsecpadre(mm)];
329                                 end
330                             end
331                         end
332 
333                         % %%%%%%%%%% %ordenarlistasec
334                         % pA=EVG.WPs(nodopropuestopadre,:);   %Actua nodox
335                         pA=p2;   %Actua nodox
336                         % pB=EVG.WPs(vecaux(end,1),:); %Actua nodoy
337                         pB=p4; %Actua nodoy
338 
339                         vA=pB-pA;
340                         listaauxsec=[listasec',zeros(length(listasec'),1)];
341 
342                         for mm=1:length(listasec)
343                             nodomm=listasec(mm);
344                             pC=WPs(nodomm,:)'; %Actua nodo a ordenar
345                             vB=pC-pA;
346                             ang=acos(dot(vB,vA)/(norm(vB)*norm(vA)));
347                             prodveck=vB(1)*vA(2)-vB(2)*vA(1);
348 
349                             if prodveck<0
350                                 %Se busca ordenar rotando desde x y
351                                 %empezando desde y
352                                 ang=2*pi-ang;
353                             end
354                             listaauxsec(mm,:)=[nodomm,ang];
355                         end
356                         listaauxsec=ordenalistasec(listaauxsec);
357                         listasec=listaauxsec(:,1);
358 
359                         if length(listasec)==2
360                             if nodoVIS==listasec(end)
361                                 %Caso D
362                                 condarista=true;
363                             else %nodoVIS==listasec(1)
364                                 %Caso E
365                                 condarista=false;
366                             end
367                         elseif length(listasec)==3
368                             if nodoVIS==listasec(1)
369                                 %CasoA
370                                 condarista=false;
371                             elseif nodoVIS==listasec(2)
372                                 %Caso B: Este podría verse afectado en
373                                 %recintos no convexos
374                                 condarista=true;
375                             elseif nodoVIS==listasec(end)
376                                 %caso C
377                                 condarista=true;
378                             end
379                         else
380                             listasec
```

```
                            error('check condarista')
                        end
                    else
                        condarista=false;
                    end

                    polnodopropuesto=EVG.nodo2poligono(nodopropuestopadre);
                    polnodox=EVG.nodo2poligono(vecaux(1,1));

                    indfutgen=MAT_posnodosvecaux(nodoaux);
                    if flagmodoejecucionfinal==0
                        if condarista&&condconvex && nodopropuestopadre ~=vecaux(1,1)&&
    polnodopropuesto~=polnodox&& MATfuturasgeneraciones(indfutgen,1)==0
                            %Esto es un último check, para evitar que se
                            %introduzcan nodos antes de tiempo
                            [condpadrefuturo,posiblespadres]=futurasgeneraciones(EVG,
    nodospendientes,nodopropuestopadre,nodoVIS,vecaux(1,1));
                            MATfuturasgeneraciones=extiendematriz(MATfuturasgeneraciones,
    indfutgen,posiblespadres);

                        elseif condarista&&condconvex && nodopropuestopadre ~=vecaux(1,1)&&
    polnodopropuesto~=polnodox&& MATfuturasgeneraciones(indfutgen,1)~=0
                            %Para el caso de que haya computados nodos, se
                            %revisará si el nodo que se propone en esta
                            %iteración es uno de los que se propuso como
                            %mejor padre
                            flag=0;
                            contador=1;
                            condpadrefuturo=false;
                            while flag==0
                                nodofuturo=MATfuturasgeneraciones(indfutgen,contador);
                                if nodofuturo==nodopropuestopadre
                                    %Si el padre coincide con uno de los
                                    %que se propuso en su momento se da
                                    %como válido y corta la ejecución
                                    [condpadrefuturo,posiblespadres]=futurasgeneraciones(EVG,
    nodospendientes,nodopropuestopadre,nodoVIS,vecaux(1,1));
                                    if ~isempty(posiblespadres)
                                        MATfuturasgeneraciones=extiendematriz(
    MATfuturasgeneraciones,indfutgen,posiblespadres);
                                    end

                                    flag=1;
                                end

                                contador=contador+1;
                                if contador<=length(MATfuturasgeneraciones(1,:))
                                    if MATfuturasgeneraciones(indfutgen,contador)==0
                                        flag=1;
                                    end
                                else
                                    %si se ha llegado al final de la
                                    %matriz, se corta la ejecución del
                                    %bucle
                                    flag=1;
                                end
                            end
                        else
                            condpadrefuturo=true;
                        end
                    else
                        condpadrefuturo=false;
                    end

                    if cond && prodvec<=0 && prodvec2>=0 && prodvec3<=0 && condconvex &&
    condarista && (condpadrefuturo||flagmodoejecucionfinal==1)
                        %Se busca en familia los hermanos
                        hermanosVIS=familia(familia(:,2)==nodopropuestopadre,1);%Nodos
                        if ~isempty(hermanosVIS)
                            hermanosVIS=[hermanosVIS;nodoVIS];
```

```
444                             posnodoy=EVG.Mat_ind_pos_enMatrizorden(nodopropuestopadre,vecaux(end
    ,1));
445                             nodoorden=EVG.Matrizorden(nodopropuestopadre,:);                  %Se
     saca el vector de orden
446                             nodoorden=[nodoorden(posnodoy+1:end),nodoorden(1:posnodoy)];       %Se
     pone a N último en orden CCW
447                             nodoorden=nodoorden(end:-1:1);                                    %
    Ahora N es primero en orden CW
448                             [~, indicefiltrado]= ismember(nodoorden, hermanosVIS);
449                             hermanosVIS=nodoorden(indicefiltrado>0);
450
451                             indVIS=find(hermanosVIS==nodoVIS,1); %Una vez ordenados se buscan en
    que posición ha quedado nodoVIS
452                             indhermanosVIS=find(familia(:,2)==nodopropuestopadre);%Indices
453
454                             if indVIS==length(hermanosVIS)
455                                 %Si es el último de la lista se introduce después de
456                                 %los descendientes del último hermano
457                                 indiniciobusqueda=indhermanosVIS(end);%Esto es valido por que
    indhermanosVIS(end) se basa en familia
458                                 padresdescendencia=familia(indiniciobusqueda,1);
459
460                                 flag=0;
461                                 indseguimiento=indiniciobusqueda+1;
462
463                                 if indseguimiento>length(familia(:,1))
464                                     flag=1;
465                                     indinserta=indiniciobusqueda;
466                                 end
467
468                                 while flag==0
469                                     if ~ismember(familia(indseguimiento,2),padresdescendencia)%si
     el padre del nodoindseguimiento no es de los permitidos entonces es que no es descendiente
470                                         flag=1;
471                                         %se rompe la ejecución, se ha identificado
472                                         %indiceinserta
473                                         indinserta=indseguimiento-1;%Se resta uno porque se ha
    pasado ya, se inserta sobre el anterior
474                                         break
475                                     end
476
477                                     if flag==0
478                                         padresdescendencia=[padresdescendencia;familia(
    indseguimiento,1)];%Si el nodo es hijo de los padres propuestos, entonces se añade a la
    lista de posibles padres futuros
479                                         indseguimiento=indseguimiento+1;
480                                     end
481
482                                     if indseguimiento==length(familia(:,1))+1 && flag==0
483                                         % Si se ha llegado al final de la lista se
484                                         % inserta en el último valor registrado
485                                         flag=1;
486                                         indinserta=indseguimiento-1;
487                                     end
488                                 end
489
490                                 familia=[familia(1:indinserta,:);nodoVIS nodopropuestopadre NaN
    (1,1) 0;familia(indinserta+1:end,:)];
491                             else
492                                 %En este caso solo es necesario insertarlo justo antes
493                                 %del siguiente hermano
494                                 nodohermanopos=hermanosVIS(indVIS+1);%hermanosVIS es una lista
    basada en nodos
495                                 indinserta=find(familia(:,1)==nodohermanopos,1)-1;%La posición
    del hermano posterior en familia -1
496                                 familia=[familia(1:indinserta,:);nodoVIS nodopropuestopadre NaN
    (1,1) 0;familia(indinserta+1:end,:)];
497                             end
498
499                         else
500                             %Si nodopropuestopadre no tiene hijos ya, se inserta el
```

```
501                             %nodo inmediatamente después del padre
502                             indinserta=find(familia(:,1)==nodopropuestopadre,1);
503                             familia=[familia(1:indinserta,:);nodoVIS nodopropuestopadre NaN(1,1)
      0;familia(indinserta+1:end,:)];
504                             %FIN isempty
505                         end
506
507                         %Se hace una rápida comprobación de si nodoVIS se puede añadir a la lista
       de padres
508
509                         padrevalido=EVG.MATposiblepadre(vecaux(1,1),nodoVIS);
510
511                         if padrevalido==true
512                             padresposiblesfuturo=[padresposiblesfuturo;nodoVIS]; %De esta manera
      en la siguiente iteración solo se trabajará con padres asignados en esta, mejorando el
      rendimiento
513                             familia(familia(:,1)==nodoVIS,4)=1;
514                         end
515
516                         seguimientonodos(seguimientonodos(:,1)==nodoVIS,2)=1;
517
518                         if flagmodoejecucionfinal~=1
519                             flag_pruebagracia=1;
520                         end
521
522                         break % el nodo está asignado, se dejan de revisar padres
523                         %Fin if es válido
524                     end
525                     %fin segundo if
526                 end
527                 %fin VIS
528             end
529             %Fin bucle padres
530         end
531         %Fin bucle nodos
532     end
533
534     if flag_padres==0
535         padresposibles=familia(familia(:,4)==1,1);%Se extraen los nodos que pueden ser padres
536         flag_padres=1;
537     else
538         padresposibles=padresposiblesfuturo;
539         padresposiblesfuturo=[];
540     end
541
542     if isempty(padresposibles)&& flagmodoejecucionfinal~=1
543         flagmodoejecucionfinal=1;
544     elseif flagmodoejecucionfinal==1
545         %Si se llega con la bandera extendida, se corta la ejecución
546         flagmodoejecucionfinal=0;
547     end
548 end
549
550 familia=familia(:,1:2);
551 familia=[familia;vecaux(end,1),vecaux(1,1)];
552
553 end
```

Código A.14 Función add_familia2.

A continuación se mostrarán las funciones auxiliares que emplea add_familia2 para cumplir sus funciones.

futurasgeneraciones

futurasgeneraciones pone en práctica parte de las ideas presentes para validar si un nodo puede presentar mejores padres o no. Por un lado, sólo considera aquellos nodos pendientes de ser asignados, ubicados en un sentido angular, desde *ii* entre *nodopropuestopadre* y *nodox*. Todas estas reglas aplican solo para los nodos que no pertenezcan al polígono de *nodox*, dado que se

considerará que son los mejores candidatos al no poderse trazar rutas más cortas hacia *nodox* que por la frontera del polígono al que pertenece.

Una vez con los nodos que se encuentren entre *nodox*y *nodopropuestopadre* localizados, se verifican las siguientes condiciones de forma secuenciada, descartando aquellos nodos que no pasen por el filtro, siendo la aplicación de estas condiciones cada vez más complejas.

1. Si existen varios nodos asociados a un mismo polígono en la lista de nodos, solo se considera aquel que forme parte de la cadena de tangentes que dirige a *nodox*, que por como está construida add_familia2 va a ser siempre *nodopropuestopadre*, por lo que se eliminan el resto de nodos que pertenezcan a este mismo polígono. De manera similar, no se va a llamar a la función futurasgeneraciones si *nodopadrenextgen* pertenece al polígono del *nodox* por lo que se eliminan también estos nodos.
2. Se comprueba que exista línea de visión directa entre los nodos restantes en la lista y *nodoVIS*, los que no cumplan esta condición se eliminan. De forma análoga a como se hacia en add_-familia2, se comprueba que los nodos propuestos como padre bloqueen la visibilidad de *nodoVIS* respecto a *nodox*.
3. Se descartan aquellos nodos que no sean fatibles para ser padres usando la información del diagrama de visibilidad extendido, $EVG.MAT posiblepadre(nodox,nodopadrenextgen)$.

Los nodos que queden restantes son susceptibles a ser una mejor opción de padre para *nodoVIS*, se pasan como salida de la función y en add_familia2 se almacenará esta información en la matriz *MAT futurasgeneraciones*, usando la función extiendematriz.

```
1  function [condpadrefuturo,posiblespadres]=futurasgeneraciones(EVG,listanodos,nodopropuestopadre,
       nodoVIS,nodox)
2  %listanodos en un vector columna
3  caso='proceso';
4
5  posnodopivote=EVG.Mat_ind_pos_enMatrizorden(nodoVIS,nodox); %Se rota desde nodoVIS a partir de
       nodox
6
7  nodoorden=EVG.Matrizorden(nodoVIS,:); %Sacamos el vector deorden
8  nodoorden=[nodoorden(posnodopivote:end),nodoorden(1:posnodopivote-1)];      %Se pone a nodox
       primero en ordenccw
9
10 %Para reducir el número de comprobaciones innecesarias se recortanlos
11 %potenciales padres futuros a revisar a aquells entre nodo x y
12 %nodopropuesto padre, esos no estarán en listanodos por lo que este
13 %recorte debe hacerse desde nodoorden
14 indnodopadre=find(nodoorden==nodopropuestopadre);
15
16 nodoorden=nodoorden(1:indnodopadre);
17
18 %Se filtra a partir de el vector recortado
19 [~, indicefiltrado]= ismember(nodoorden, listanodos);
20 listanodos=nodoorden(indicefiltrado>0);
21
22 if isempty(listanodos)
23     caso='vacio';
24 end
25
26 if strcmp(caso,'proceso')
27     %Comprobación nodos mismo polígono-nodopropuestopadre
28     [a,b]=size(listanodos);
29
30     if b>a
31         listanodos=listanodos';
32     end
33
34     listanodosextend=[listanodos,zeros(length(listanodos),1)];
35
```

```
36      for jj=1:length(listanodosextend(:,1))
37          % nodoanalisis=listanodosextend(jj,1)
38          % EVG.nodo2poligono(nodoanalisis)
39          listanodosextend(jj,2)=EVG.nodo2poligono(listanodosextend(jj,1));
40      end
41      polnodopropuesto=EVG.nodo2poligono(nodopropuestopadre);
42      polnodox=EVG.nodo2poligono(nodox);
43
44      listanodosextend(listanodosextend(:,2)==polnodopropuesto,:)=[];%El primer intento de engarce
         que funcione con un nodo de un polígono es el válido, el resto de nodos del polígono son sub
         óptimos por lo que hay que eliminarlos
45      listanodosextend(listanodosextend(:,2)==polnodox,:)=[];%No se consideran los nodos asociados
         a los poligonos que pertenezcan a nodox
46
47      if isempty(listanodos)
48          caso='vacio';
49      end
50
51  end
52
53
54  if strcmp(caso,'proceso')
55      % Se revisa si los padres restantes son compatibles con nodoVIS
56      %Dado que falta información se usarán sólo dos criterios: Visibilidad y
57      %convexidad con nodox
58      listanodosextend(:,2)=0; %Se borra la información de los polígonos, ya no es relevante
59
60      for jj=1:length(listanodosextend(:,1))
61          nodopadrenextgen=listanodosextend(jj,1);
62
63          condVIS=EVG.Visibilidad(nodoVIS,nodopropuestopadre);
64
65          p1=EVG.WPs(nodoVIS,:);          %NodoVIS
66          p2=EVG.WPs(nodox,:);            %Nodox: Servirá como referente para comprobar la
         convexidad
67          p3=EVG.WPs(nodopadrenextgen,:); %Nodopadrenextgen: se valorará como pivote
68
69          v1=p1-p3;
70          v2=p2-p3;
71          prodvec=v1(1)*v2(2)-v1(2)*v2(1);
72
73          %Para mantener la convexidad es necesario que nodovis encuentre el
74          %camino más corto pivotando por nodonextgen hacia nodox en sentido
75          %CCW
76          if prodvec>0 && condVIS
77              listanodosextend(jj,2)=1;
78          end
79      end
80
81      listanodosextend(listanodosextend(:,2)==0,:)=[];%Todos los valores que sean cero no son de
         interés, son padres predescartados
82
83      if isempty(listanodos)
84          caso='vacio';
85      end
86  end
87
88  if strcmp(caso,'proceso')
89      % Comprobación test de paternidad: Esos nodos restantes pueden ser
90      % padres?
91      listanodosextend(:,2)=0; %Se borra la información de los polígonos, ya no es relevante
92      flag=0;
93
94      for jj=1:length(listanodosextend(:,1))
95          nodopadrenextgen=listanodosextend(jj,1);
96          padrevalido=EVG.MATposiblepadre(nodox,nodopadrenextgen);
97
98          if padrevalido==true
99              %Se registra de los nodos restantes si hay alguno que pueda
100             %ser padre
101             % nodopadrenextgen
```

```
            listanodosextend(jj,2)=1; %este padre es válido
            flag=1;
        end
    end

    if flag==0
        caso='vacio';
    else
        listanodosextend(listanodosextend(:,2)==0,:)=[];%Todos los valores que sean cero no son
     de interés, son padres predescartados
    end
end

switch caso
    case 'proceso'
        condpadrefuturo=false;
        posiblespadres=listanodosextend(listanodosextend(:,2)==1,1);
    case 'vacio'
        %si no hay padres posibles esta condición debe ser true
        condpadrefuturo=true;
        posiblespadres=[];
end

end
```

Código A.15 Función futurasgeneraciones.

generalistanodos

generalistanodos genera la lista de nodos por asignar ordenados respecto a *nodopropuestopadre*. Para ello toma el conjunto de nodos ordenados al completo y filtra respecto a la lista de nodos no asignada. Esta lista se reordena de manera que comienza en *N*, que su posición relativa es siempre atrás a la izquierda del *nodox*, por como se plantean los conos de visión siempre hacia occidente, y así garantizar que la secuencia es parte de una referencia correcta.

```
function listanodos=generalistanodos(nodopropuestopadre,nodopivote,listanodos,EVG)
%Se mira como se ordenan los nodos alrededor del padre propuesto

posnodopivote=EVG.Mat_ind_pos_enMatrizorden(nodopropuestopadre,nodopivote);

nodoorden=EVG.Matrizorden(nodopropuestopadre,:); %Se saca el vector deorden
nodoorden=[nodoorden(posnodopivote+1:end),nodoorden(1:posnodopivote)];      %Se pone a N último
     en orden CCW
nodoorden=nodoorden(end:-1:1);                              %Ahora N es primero en orden CW

[~, indicefiltrado]= ismember(nodoorden, listanodos);
listanodos=nodoorden(indicefiltrado>0);
end
```

Código A.16 Función generalistanodos.

extiende_matriz incorpora a la matriz, la lista de nodos que podrían conformarse como padres óptimos para un *nodoVIS* dado, este se identifica en la matriz con el indicador de posición pasado como argumento. Rellena con ceros para mantener consistente la concatenación de la matriz, dependiendo si el vector a introducir tiene más columnas que esta o no.

```
function MAT=extiendematriz(MATBASE,indice,vector)

[a,b]=size(MATBASE);
[c,d]=size(vector);%Debe ser fila, por si acaso se añade revisión para hacerlo columna

if c>d
    vector=vector';
    [~,d]=size(vector);
```

```
end

%indice indica donde hay que insertar el valor

if b>d
    %Si la matriz es más ancha que el vector se introduce ampliando el vector
    MAT=[MATBASE(1:indice-1,:);vector,zeros(1,b-d);MATBASE(indice+1:end,:)];
else
    %En caso contrario hay que expandir la matriz
    MAT=[MATBASE(1:indice-1,:),zeros(length(1:indice-1),d-b);vector;MATBASE(indice+1:end,:),zeros
     (length(indice+1:a),d-b)];
end

end
```

Código A.17 Función extiendematriz.

ordenalistasec

ordenalistasec es una función auxiliar del cálculo de *listasec*, que sirve para calcular *condarista*. Los argumentos del programa son la lista de nodos consecutivos en el polígono de $nodopadrepropuesto$ y $nodoVIS$, junto con los ángulos dados respecto a la recta $nodopropuestopadre - nodoy$, para poder así extraer la información con los nodos dados en sentido creciente de las agujas del reloj. Como el número de puntos a ordenar va a ser siempre igual a 2 o a 3, esta ordenación se hace a mano, dado que la función *sortrows* consumía mucho tiempo en su ejecución. Es por ello que se ordena a mano a través de un sistema de sentencias if, siguiendo una lógica similar a de *Bubble-sort*, cambiando posiciones en los vectores con ciertas hipótesis adicionales que pueden brindarse debido a tener, en el peor de los casos, sólo tres componentes.

```
function listaaordenar=ordenalistasec(listaaordenar)
if length(listaaordenar(:,1))==2

    if listaaordenar(1,2)>listaaordenar(2,2)
        listaaordenar=[listaaordenar(2,:);listaaordenar(1,:)];
    else
        listaaordenar=[listaaordenar(1,:);listaaordenar(2,:)];
    end
elseif length(listaaordenar(:,1))==3
    if listaaordenar(1,2)>listaaordenar(2,2)
        listaaordenar=[listaaordenar(2,:);listaaordenar(1,:);listaaordenar(3,:)];
    else
        listaaordenar=[listaaordenar(1,:);listaaordenar(2,:);listaaordenar(3,:)];
    end

    if listaaordenar(3,2)<listaaordenar(2,2)
        listaaordenar=[listaaordenar(1,:);listaaordenar(3,:);listaaordenar(2,:)];

        if listaaordenar(1,2)>listaaordenar(2,2)
            listaaordenar=[listaaordenar(2,:);listaaordenar(1,:);listaaordenar(3,:)];
        end

    end
else
    error('Revisar listasec, longitud no aceptada.')
end

end
```

Código A.18 Función ordenalistasec.

Con ello concluye el conjunto de códigos elaborados a lo largo de este trabajo.

Índice de Figuras

Índice de Tablas

Índice de Códigos

Bibliografía

[1] Ayala, J. F. M. «Análisis de los costos de las aerolíneas». En: *Visionario Digital* (jul. de 2019), págs. 313-326. doi: *10.33262/visionariodigital.v3i3.856*. url: *https://www.researchgate.net/publication/335945201_Analisis_de_los_costos_de_las_aerolineas*.

[2] *El Hospital público La Paz de la Comunidad de Madrid participa en un proyecto para el transporte aéreo de material sanitario con drones | Comunidad de Madrid*. Mayo de 2023. url: *https://www.comunidad.madrid/noticias/2023/05/11/hospital-publico-paz-comunidad-madrid-participa-proyecto-transporte-aereo-material-sanitario-drones*.

[3] *El robot autónomo de Amazon que jubilará a los mozos de almacén: mueve y ordena los paquetes*. Oct. de 2023. url: *https://www.elespanol.com/omicrono/tecnologia/20231020/robot-autonomo-amazon-jubilara-mozos-almacen-mueve-ordena-paquetes/803419865_0.html*.

[4] Johnson, K. *Amazon's 'Safe' New Robot Won't Fix its Worker Injury Problem | WIRED*. Jul. de 2022. url: *https://www.wired.com/story/amazons-worker-injury-problem/*.

[5] González, N. V. et al. *Mejora computacional de herramientas de planificación de trayectorias de avión bajo meteorología adversa*. Ago. de 2023. url: *https://hdl.handle.net/11441/148386*.

[6] Ghosh, S. K. y Mount, D. M. «An Output-Sensitive Algorithm for Computing Visibility Graphs». En: *SIAM Journal on Computing* 20 (5 oct. de 1991), págs. 888-910. doi: *10.1137/0220055*. url: *https://doi.org/10.1137/0220055*.

[7] O'Rourke, J. y Streinu, I. «The vertex-edge visibility graph of a polygon». En: *Computational Geometry* 10 (2 1998), págs. 105-120. doi: *https://doi.org/10.1016/S0925-7721(97)00011-4*. url: *https://www.sciencedirect.com/science/article/pii/S0925772197000114*.

[8] Laguna, G. J. y Bhattacharya, S. «Path planning with Incremental Roadmap Update for Visibility-based Target Tracking». En: *2019 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. Nov. de 2019, págs. 1159-1164. doi: *10.1109/IROS40897.2019.8967602*.

[9] Mehlhorn, K. y Sanders, P. *Algorithms and Data Structures: The Basic Toolbox*. Ed. por I. S. P. Company. 1.ª ed. Vol. 1. Springer Publishing Company, Incorporated, ago. de 2008.

[10] Gabow, H. N. y Tarjan, R. E. «A linear-time algorithm for a special case of disjoint set union». En: *Journal of Computer and System Sciences* 30 (2 dic. de 1985), págs. 209-221. doi: *https://doi.org/10.1016/0022-0000(85)90014-5*. url: *https://www.sciencedirect.com/science/article/pii/0022000085900145*.

[11] Ligthart, L. P., Yanovsky, F. J. y Prokopenko, I. G. «Adaptive algorithms for radar detection of turbulent zones in clouds and precipitation». En: *IEEE Transactions on Aerospace and Electronic Systems* 39 (1 ene. de 2003), págs. 357-367. doi: *10.1109/TAES.2003.1188918*.

[12] Kitzinger, J. «The Visibility Graph Among Polygonal Obstacles: a Comparison of Algorithms». En: *Article*. 2003. url: *https://api.semanticscholar.org/CorpusID:9518166*.