FACULTAD DE MATEMÁTICAS EN INFORMÁTICA

ASIGNATURA DE GEOMETRÍA Y TOPOLOGÍA COMPUTACIONAL

MEMORIA DE LA PRÁCTICA

Trabajo realizado por Ignacio Amaya

Asignatura impartida por:
Manuel Abellanas y Antonio Giraldo

Índice

Índice de figuras

Introducción

La geometría y topología computacional es una rama de la informática que se centra en encontrar algoritmos lo más eficientes posibles para resolver problemas de geometría y topología.

A pesar de que muchos de los problemas a los que trata parecen muy fáciles a simple vista, su implementación eficiente puede suponer un reto bastante considerable. Cabe destacar la ausencia del uso de ángulos a la hora de resolver los problemas geométricos. Esto se debe a que al introducir ángulos siempre aparece el número π , lo que implica que tenemos que aproximar valores. Al trabajar con muchos puntos los errores de las aproximaciones pueden acumularse y estropear el resultado final.

En esta memoria se incluyen los trabajos correspondientes a las dos partes de las que consta la asignatura. En la primera se ha realizado una aplicación gráfica usando Java y aprovechando la herramienta Processing para implementar y probar gráficamente algunos de los algoritmos vistos en la asignatura. En la segunda se ha implementado en Python un algoritmo que se había enseñando en clase. Hemos escogido Python por su versatilidad a la hora del manejo de listas, algo que nos facilitaba bastante las cosas para lo que se quería hacer.

Parte I
Trabajo de geometría

1. Introducción

Para esta parte se han implementado varios de los algoritmos que se habían visto en clase. Para facilitar su visualización se ha optado por realizar una interfaz gráfica con diversos botones que pondrían en funcionamiento nuestros algoritmos.

Esta aplicación se ha realizado en el entorno Processing y debido a ello se han tenido que tener en cuenta diversas consideraciones. Las coordenada $(0, 0)$ en Processing es el punto superior izquierdo de la pantalla, por lo que el eje de coordenadas es simétrico al usual y la mayoría de las operaciones usadas dependientes de la orientación tienen que tener esto en cuenta.

Al iniciar la aplicación tenemos un panel a la derecha donde se muestran las distintas opciones que podemos realizar y en el centro una zona en blanco, que es donde se dibujarán los resultados. Se pueden añadir puntos haciendo

click con el ratón en la zona de dibujo. También se podrán mover si los arrastramos con el botón del ratón pulsado. Sin embargo, esta funcionalidad de arrastrar puntos sólo funciona para menos de 20 puntos, pues también se pueden mover cuando se hayan ejecutado los algoritmos y se modificará la imagen en tiempo real. El problema es que para más de 20 puntos se hace menos fluido el movimiento, ya que la cantidad de veces que se ejecuta cada algoritmo es de 60 por segundo. En la parte inferior se muestran tres contadores que llevan la cuenta de los vértices, las caras y las aristas.

A continuación resumimos la función de cada uno de los botones del panel derecho:

- Envolvente: Dibuja la envolvente convexa de los puntos que se encuentren en la zona de dibujo.
- Triangular: Dibuja una triangulación de los puntos que se encuentren en la zona de dibujo. Se puede seleccionar el tipo de triangulación, que puede ser la incremental o la de Delaunay. Si ninguna de las dos está seleccionada dibujará la incremental.
- Voronoi: Representa en rojo el diagrama de Voronoi superpuesto a la triangulación de Delaunay.
- Puntos: Pinta el número de puntos seleccionado. Si no hay ningún número de puntos seleccionado no pinta nada.
- Borrar: Borrar todos lo que esté dibujado en la zona de dibujo.

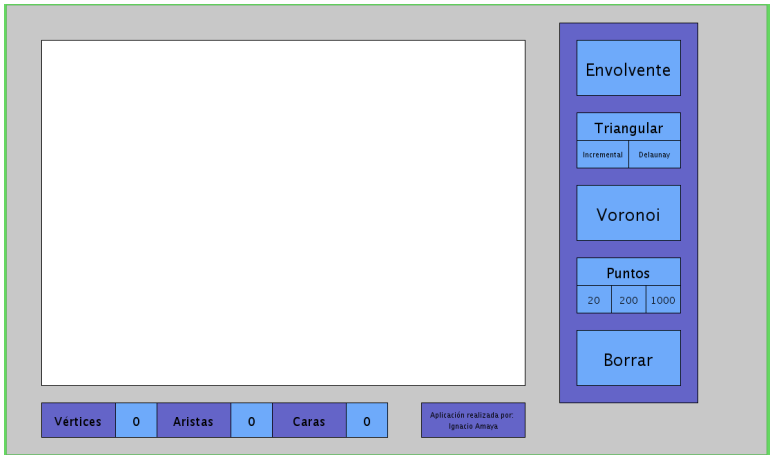


Figura 1: Imagen de la interfaz al iniciarse

2. Algoritmos

A continuación se van a explicar los diferentes algoritmos que se han utilizado en la realización de esta parte.

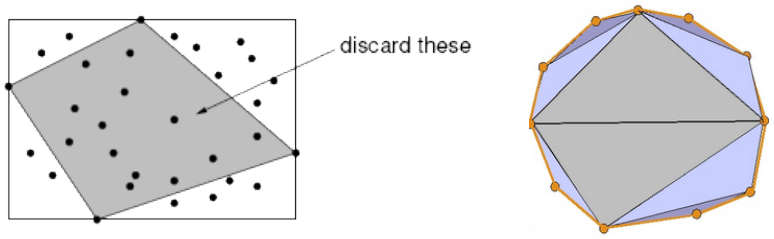
2.1. Cierre convexo

El cierre convexo de un conjunto X de puntos es el mínimo conjunto convexo que contiene a X. En dos dimensiones puede visualizarse como la forma que tomaría una banda elástica que rodee al conjunto. También se lo conoce como envolvente convexa.

Se ha realizando mediante el algoritmo recursivo QuickHull. Su complejidad es, de media, $O(n * log(n))$ en general y $O(n^2)$ en el peor caso, que suele darse cuando los puntos son muy simétricos o si se encuentran formando una circunferencia.

El algoritmo que se ha implementado se diferencia del tradicional en que se parte de cuatro puntos en vez de dos. Los pasos de los que consta son los siguientes:

- Encontrar el máximo y el mínimo en la coordenada x (más a la izquierda) y el máximo y mínimo en la coordenada y (más a la derecha). Podrá darse el caso de que se parta de menos puntos (en el caso de que un puto sea el mayor o el menor en ambas coordenadas).
- Aplicar recursivamente en cada uno de los cuatro vectores (dados en sentido positivo) formados por los puntos anteriores los siguientes pasos:
  - Determinar el punto más alejado a la derecha del vector AB.
  - El punto encontrado forma un triángulo con los dos extremos del vector y todos los puntos contenidos en él son descartados en los siguientes pasos, ya que no formarán parte del cierre convexo.
  - Repetir los dos pasos anteriores en los dos vectores formados por AP y PB hasta que no se encuentren puntos a la derecha del vector (fin de la recursión).



(a) Primer paso en el QuickHull

(b) QuickHull terminado con iteraciones diferenciadas por colores

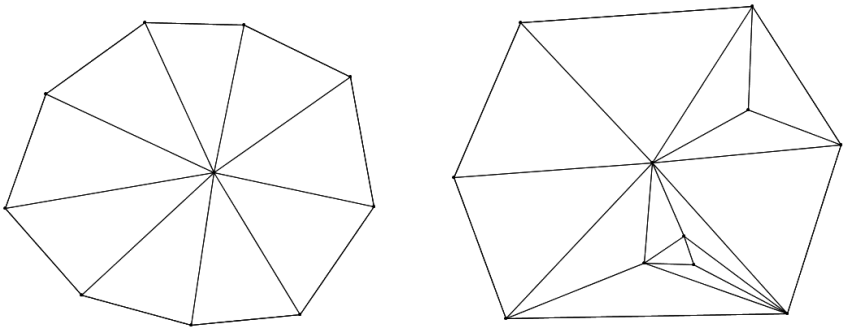
Figura 2: Iteraciones del QuickHull

2.2. Triangulación

Una triangulación de un conjunto X de puntos es equivalente a una triangulación del cierre convexo. Consiste en su descomposición en triángulos, de forma que al final todos los vértices de los triángulos obtenidos sean los que teníamos en nuestro conjunto X. Se ha realizado una triangulación usando un algoritmo incremental. Se ha partido del cierre convexo y se han realizado los siguientes pasos:

- Se ha tomado el primer punto creado que no forma parte del cierre y se ha unido con todos los puntos que componen el cierre convexo. Esta triangulación preeliminar se ha almacenado en un DCEL.
- Para cada uno de los puntos restantes se realiza lo siguiente:
  - Determinar dentro de qué triángulo del DCEL se encuentra el punto.
  - Unir el punto con los tres vértices del triángulo que lo contiene (se añaden tres triángulos al DCEL y se elimina el que estaba anteriormente).

En cada una de las modificaciones del DCEL habrá que cambiar los parámetros de las aristas, los vértices y las caras correspondientes que se vean afectados.



(a) Primer paso en la triangulación (b) Varias iteraciones del algoritmo

Figura 3: Iteraciones de la triangulación incremental

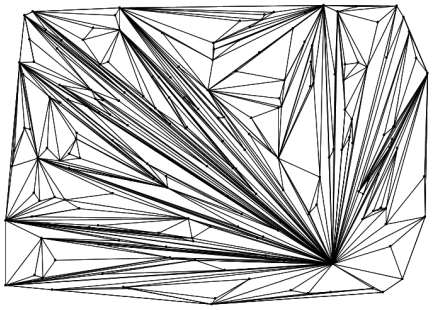


Figura 4: Ejemplo de 200 puntos triangulados

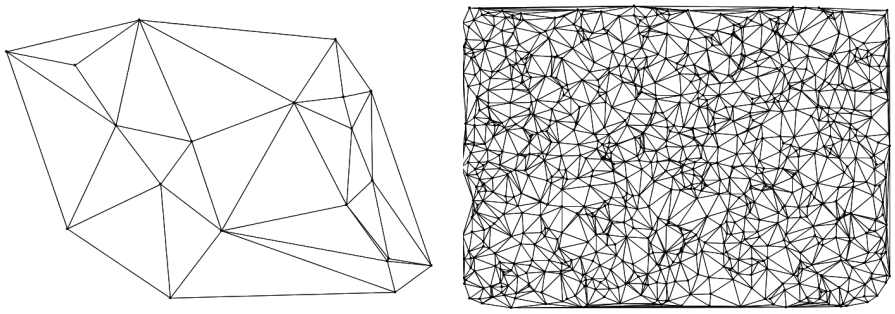
2.3. Triangulación de Delaunay

La triangulación de Delaunay para un conjunto P de puntos es una triangulación ($DT(P)$) que minimiza los ángulos de los triángulos que se obtienen. Se cumple que no puede haber ningún punto dentro de las circunferencias circunscritas de cualquiera los triángulos.

Aprovechando esta última propiedad se ha realizado el algoritmo de los *flips*. Para construir $DT(P)$ tenemos que partir de una triangulación ya existente. En nuestro caso usamos la que hemos obtenido en el apartado anterior. A continuación recorremos las aristas y vemos si son legales o no. Una arista es legal si los triángulos en los que se encuentra contenida cumplen que no tienen puntos en el interior de su circunferencia circunscrita. En caso de encontrarnos ante una arista ilegal, la giramos (en inglés *flip*, de

ahí el nombre del algoritmo). Cuando ya no se encuentren aristas ilegales el algoritmo termina y obtenemos $DT(P)$.

Todo este proceso puede llegar a tener una complejidad $O(n^2)$, pero en general suele realizarse en menos tiempo y la mayor parte del tiempo consumido se emplea en obtener la triangulación inicial.



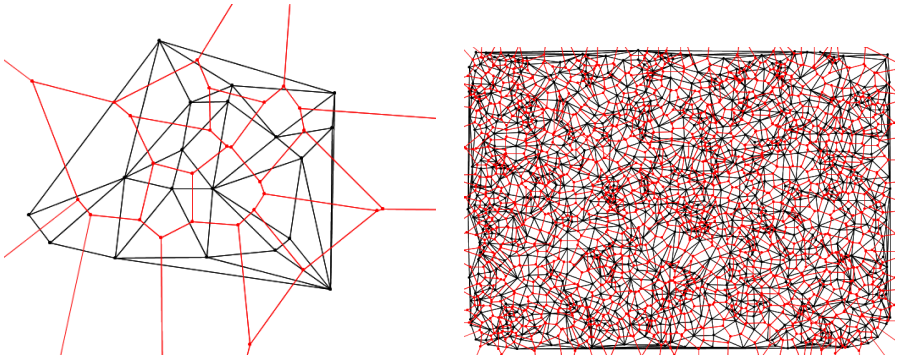
(a) DT para 20 puntos (b) DT para 1000 puntos

Figura 5: Ejemplos de la triangulación de Delaunay

2.4. Diagrama de Voronoi

El diagrama de Voronoi es una partición del plano basada en la distancia a un conjunto P de puntos. Una región de Voronoi de un punto P_i será, por tanto, la zona del plano donde el punto más cercano es P_i y las aristas del diagrama serán regiones donde los puntos están a la misma distancia de varios. El número de regiones será igual al número de puntos.

Dada una triangulación de Delaunay se puede obtener fácilmente su diagrama de Voronoi. Los puntos donde confluyen varias de estas aristas coinciden con los circuncentros de $DT(P)$. Por tanto para hallar el diagrama de Voronoi basta con calcular los circuncentros y unirlos con los circuncentros de los triángulos adyacentes. En los triángulos con aristas en el cierre convexo, se trazará una recta perpendicular a ella que vaya hasta el infinito.



(a) Diagrama de Voronoi para 20 puntos (b) Diagrama de Voronoi para 1000 puntos

Figura 6: Ejemplos del diagrama de Voronoi

3. Clases del código

Para realizar los algoritmos implementados se han utilizado varias clases. A continuación vamos a explicar lo más relevante de cada una de ellas. No se pretende dar una explicación minuciosa de cada una de ellas, sino simplemente una descripción general para hacerse una idea de como están implementados los algoritmos y para explicar algunas de las decisiones de diseño.

3.1. clase *cierreConvexo*

En esta clase se implementa el cierre convexo. Internamente tiene una copia de la lista de los puntos, una lista auxliar para llevar la cuenta de cuáles están fuera y dentro en cada iteración, una lista donde se guardarán los punto del cierre convexo y otra lista que almacena los puntos que se encuentran en línea, pues no los trataremos y tendremos que borrarlos posteriormente. Los métodos que utiliza son los siguientes:

- extremos: Nos devuelve los puntos iniciales que usaremos para comenzar con nuestro algoritmo.
- areaSignada: Nos da el signo del área signada. La usaremos para decidir si los puntos están en la cara interna o externa al ir realizando el cierre, y así descartar los que estén en la interna.
- distanciaRecta: Lo usaremos para tomar el punto más alejado del vector formado por los dos puntos en cada iteración.

- cerrar: Realiza el proceso recursivo dados dos puntos para obtener el cierre convexo.

3.2. clase *DCEL*

Utiliza otras tres clases que están en el mismo fichero: la clase *Vertice*, la clase *SemiArista* y la clase *Cara*. Habrá dos objetos SemiArista por cada arista normal.

Un DCEL estará formado por tres listas, una de cada uno de los tres elementos anteriores. Los vértices, las aristas y las caras tendrán los atributos que les corresponden siguiendo la estructura clásica del DCEL. Esto es:

- Vértices: Punto y una semiarista que sale de él (la de menor índice).
- Semiaristas: Vértice origen, semiaristas gemela, anterior, siguiente y posterior y la cara en la que se encuentra.
- Caras: Semiarista que la define (la de menor índice).

La clase DCEL no realiza ninguna operación, pues sólo es una estructura de datos y las modificaciones se harán desde fuera. Para modificar un DCEL basta con cambiar los atributos de cualquiera de sus vértices, caras y aristas. El único inconveniente de esto es que si se realiza algún cambio en la estructura erróneo es bastante complicado su detección.

3.3. class *Triangulacion*

Realiza tanto la triangulación de Delaunay como la incremental. Tiene una lista donde se guarda el cierre, otra con los puntos, dos para cada una de las triangulaciones y una última para llevar la cuenta de los puntos que se encuentran en la frontera de un triángulo (y que posteriormente borramos).

Los métodos que utiliza son los siguientes:

- enTriangulo: Detecta si un cuarto punto se encuentra dentro, fuera o en la frontera del triángulo que forman los tres primeros.
- enCirculo: Determina si un cuarto punto se encuentra en la circunferencia que determinan los otros tres.
- triangular: Método que crea el DCEL inicial y va llamando a los otros dos métodos auxiliares para hacer la triangulación. Lo primero que hace es unir un vértice con todos los demás en caso de no haber un punto dentro del cierre (hay tres subcasos según se añadan el primer

triángulo, los intermedios o el último). Si hay uno o más puntos dentro, entonces une ese punto con todos los del cierre convexo (también hay tres subcasos según se añadan el primer triángulo, los intermedios o el último). Tras esto va mirando los siguientes puntos y si se encuentran dentro de un triángulo llama a triangularAux y si se encuentran en la frontera llama a triangularAuxDeg. Hay que tener en cuenta que a la hora de añadir al DCEL inicial triángulos hay que ser muy cuidadoso e ir borrando las distintas cara y aristas que vayan desapareciendo además de añadir las nuevas.

- triangularAux: Crea tres triángulos en el DCEL y borra uno existente realizando los cambios oportunos.
- triangularAuxDeg: Está comentado porque no se ha logrado que funcione correctamente. Debería crear dos triángulos y borrar uno ya existente.
- flip: Método que se usa para hacer *flip* en las aristas y así poder conseguir una triangulación de Delaunay.
- convertirDelaunay: Realiza una pasada por el DCEL y realiza un *flip* en las aristas no legales (el test de legalidad se realiza con el método enCirculo). Cada vez que se realiza un *flip* las tres semiaristas de cada triángulo y sus gemelas se bloquean para no poder ser cambiadas en esa misma pasada. Esto se debe a que en Java no está permitido modificar la estructura sobre la que se está iterando (error concurrente). Este método devuelve un booleano que será True si ha habido modificaciones en el DCEL y False en caso contrario. Este método se llamará en la clase Interfaz y se ejecutará hasta que devuelva False. De esta forma obtendremos la triangulación de Delaunay.

3.4. class *Voronoi*

Esta clase tiene un DCEL que contendrá la triangulación de Delaunay sobre la que trabajará y dos diccionarios. El primero relaciona cada circuncentro con su cara y el segundo relaciona cada semiarista del cierre con el punto con el que se tienen que unir, que será un valor muy elevado que se encuentre en la recta perpendicular a ella en la dirección correcta.

- getCircuncentro: Dada una semiarista obtiene el circuncentro del triángulo que determina.

- getPuntoMedio: Obtiene el punto medio de una semiarista. Se usa para pintar las rectas que se van al infinito en el diagrama.
- sustituirEnRecta: Dados dos puntos que determinan una recta y un valor de x, obtiene el valor de y asociado a esa x.
- getVoronoi: Obtiene el diagrama de Voronoi. Para ello va obteniendo los circuncentros y va creando los dos diccionarios mencionados anteriormente, que definen el diagrama.

3.5. clase *Punto*

Cada punto tiene sus dos coordenadas x e y y ciertas propiedades que se usan a la hora de dibujarlos (grosor y color).

3.6. clase *Arista*

Esta clase sólo se usa para dibujar las triangulaciones y el DCEL. Para realizar operaciones siempre se usan las semiaristas del DCEL.

3.7. clase *Boton*

Clase en donde se implementan los botones de la aplicación.

3.8. clase *Interfaz*

Clase principal donde se crean las distintas partes y botones. También se especifican los eventos que se ejecutan al hacer *click* en los botones. Muchos de estos botones llaman a las clases explicadas anteriormente y, a partir de ellas, realizan la representación gráfica de los resultados.

4. Problemas

Los principales problemas que se han afrontado han sido los relacionados con el DCEL, ya que cualquier mínimo fallo a la hora de introducir nuevas semiaristas y cambiar las conexiones existentes tenían que pensarse muy bien. Para ello se han realizado múltiples dibujos en hojas de sucio para determinar bien cómo debían ser los cambios.

Por otra parte, Java es muy rígido en cuando al manejo de los datos, lo que ha hecho que algunas operaciones que en principio parecían bastante sencillas al final requiriesen bastante tiempo y esfuerzo.

La ventaja de contar con soporte gráfico es que en la mayor parte de los casos el error se detectaba fácilmente y las causas a veces estaban claras en el dibujo, por lo que se facilitaba su corrección.

5. Posibles mejoras

Primeramente se debería corregir la aplicación para que se traten los casos de puntos en línea en el cierre y los de puntos en triángulo de la triangulación, ya que ahora mismo se eliminan (por eso al triangular un gran número puntos el contador de vértices marca un poco menos de lo que debería).

Otra posible mejora sería añadir la opción de representar superpuesta al diagrama de Voronoi una sucesión completa de α-complejos para los distintos radios en los que se añaden caras y aristas.

Parte II
Trabajo de topología

1. Introducción

En esta parte se ha realizado en Python un programa que calcula los números de Betti a partir del conjunto de símplices maximales de un complejo simplicial. Para ello se ha utilizado el algoritmo matricial visto en clase.

Se ha escogido Python porque ofrece mucha facilidad a la hora de manejar listas y, como íbamos a trabajar con matrices, esto nos facilitaba mucho las cosas. Además la bilioteca numpy de Python nos ofrece una gran ayuda a la hora de trabajar con matrices.

2. Algoritmos

Los grupos de homología de una figura n-dimensional triangulada pueden representarse mediante homomorfismos borde. Las matrices reducidas nos proporcionan los rangos de los ciclos y de los grupos borde. Su diferencia nos permite obtenener los números de Betti.

Para construir las matrices borde de dimensión p, se representan los $(p-1)$-símplices como columnas y los p-símplices como filas. Se pondrá un

1 o un 0 dependiendo de si los $(p-1)$-símplices son caras de los p-símplices (debe hacerse en un orden preestablecido). El signo será positivo o negativo alternos, siendo el primero positivo o negativo dependiendo de la dimensión en la que nos encontremos.

Estas matrices será necesario reducirlas realizando operaciones por filas y por columnas similares a las que se realizan al resolver un sistema. Tras estos cambios podremos hallar el número de Betti correspondiente mediante la siguiente fórmula:

$$dim(C_K) = dim(Ker\partial) + dim(Im\partial)$$

3. Métodos empleados en el código

Se han empleado métodos básicos para realizar las operaciones por filas y por columnas y poder operar con matrices n-dimensionales. Despúes se han transformado los símplices maximales en matrices para poder calcular sus números de Betti. A continuación se resume lo que hacen las funciones más importantes (lo que hace el resto aparece comentado en el código):

- numBetti: Dadas dos matrices del borde obtiene su número de Betti correspondiente.
- dividirEnListas: Dado un símplice obtiene todos los ciclos asociados de dimensión inferior.
- elementosSmplice: Obtiene todos los símplices existentes a partir de uno dado.
- crearMatrizSimplices: Crea todas las matrices borde dado un conjunto maximal de símplices.
- símplicesBetti: Dado un conjunto maximal de símplices, nos devuelve sus números de Betti.

4. Pruebas

Las pruebas realizadas se han realizado usando los complejos simpliciales que nos proporcionaba SAGE. Para ello se ha usado la biblioteca *simplicial_complexes*. Por ejemplo, para obtener los complejos maximales del toro bastaba con escribir lo siguiente: *simplicial_complexes.Torus().maximal_faces()*.

Los resultados concuerdan con los valores de los números de Betti dados en clase y con los que se obtienen en SAGE. Se han realizado pruebas para ejemplos de símplices que se han construido y otros para los típicos (toro, esfera, botella de Klein y plano proyectivo).

5. Problemas

Se han tenido problemas a la hora de realizar las pruebas, ya que aparte de los complejos simpliciales más conocidos no se podía probar el programa con otros de dimensión de 4 en adelante, ya que no es posible su visualización para seleccionar los complejos maximales.

6. Posibles mejoras

Se debería investigar por qué la prueba del complejo K3 (de dimensión 4) falla en β_3 (da 1 y según SAGE debería ser 0) y determinar si es culpa de algún error en un índice en el código o simplemente se debe a que el complejo simplicial es erróneo (se ha obtenido de la biblioteca de SAGE).

Conclusión

Este trabajo me ha servido para poner en práctica los conocimientos adquiridos en la asignatura y de esa manera afianzarlos. Además al ser la primera vez que realizaba una aplicación gráfica, esto ha supuesto una dificultad añadida. Sin embargo, una vez terminada, el esfuerzo ha merecido la pena.

Por tanto, mi valoración personal de la práctica es bastante positiva, pues he aprendido bastante realizándola y me siento satisfecho con el resultado.

Bibliografía

[1] Devadoss, Satyan L., and Joseph O'Rourke. «Discrete and computational geometry.» Princeton University Press, 2011.

[2] De Berg, Mark, et al. «Computational geometry.» Springer Berlin Heidelberg, 2000.

[3] Edelsbrunner, Herbert, and John Harer. «Computational topology: an introduction.» *American Mathematical Soc.*, 2010.

[4] Rote, Günter, and Gert Vegter. «Effective Computational Geometry for Curves and Surfaces.»

[5] Zomorodian, Afra.«Computational topology.»Algorithms and theory of computation handbook. Chapman & Hall/CRC, 2010.