Universidad Politécnica
de Madrid

Escuela Técnica Superior de Ingenieros Informáticos

Grado en Matemáticas e Informática

Trabajo Fin de Grado

Robustez de la Homología Persistente: el Teorema de Estabilidad

Autor: Alejandro García Castellanos
Tutor(a): Héctor Barge Yañez

Madrid, 06 - 2021

Este Trabajo Fin de Grado se ha depositado en la ETSI Informáticos de la Universidad Politécnica de Madrid para su defensa.

Trabajo Fin de Grado
Grado en Matemáticas e Informática

Título: Robustez de la Homología Persistente: el Teorema de Estabilidad

06 - 2021

Autor: Alejandro García Castellanos
Tutor: Héctor Barge Yañez
Departamento de Matemática Aplicada
ETSI Informáticos
Universidad Politécnica de Madrid

Resumen

Sección por hacer

«Aquí va el resumen del TFG. Extensión máxima 2 páginas.»

Abstract

Sección por hacer

«Abstract of the Final Degree Project. Maximum length: 2 pages.»

Tabla de contenidos

Capítulo 1

Introducción

1.1. Motivación

El crecimiento de la cantidad de datos que producimos y almacenamos es exponencial, estimándose que para 2025 llegaremos a haber creado un total de 160 zettabytes [1]. En este contexto surge la necesidad de analizar y comprender las características de grandes conjuntos de datos. Así pues, nace la disciplina del *Análisis Topológico de Datos* con el fin de responder a las siguientes preguntas sobre las propiedades cualitativas geométricas de nuestros conjuntos de datos: ¿Cuáles son las características topológicas de mi conjunto de datos? Si hemos introducido complejidad adicional a nuestros datos debido a problemas de medición o de discretización, ¿cómo medimos la relevancia de las características observadas?

Para poder intentar contestar a estas preguntas se hace uso de la *homología persistente*, y más concretamente de los *diagramas de persistencia*. Estos diagramas son multiconjuntos de puntos en el plano extendido, dónde cada punto representa una característica cualitativa de nuestros datos y la diferencia en valor absoluto de sus coordenadas cuantifica su relevancia. Algunas de las características usuales que se miden son el número de componentes conexas, el número de agujeros y el número de cavidades.

1.2. Descripción general del trabajo

El trabajo se basa en el estudio y exposición del Teorema de estabilidad, que establece, a grandes rasgos, que pequeñas perturbaciones en los datos implican pequeñas perturbaciones en la homología persistente.

Para ello he buscado y estudiado diversas referencias acerca de este resultado y su demostración [2, 3, 4].

Adicionalmente, he implementado en Python las distancias *Bottleneck* y *Hausdorff* para poder ilustrar este resultado haciendo uso distintos conjuntos de datos.

Las pruebas se sustentan en el *pipeline* usual del Análisis Topológico de Datos, ilustrado en la figura 1.1. Partiremos de un conjunto de datos, en este caso, una nube de puntos. Posteriormente se obtendrá una familia de espacios topológicos sobre los

obtendremos la homología persistente que será representada en el diagrama de persistencia. Para estudiar la estabilidad, se introducirán perturbaciones en conjunto de puntos inicial y se compararán los diagramas de persistencia.

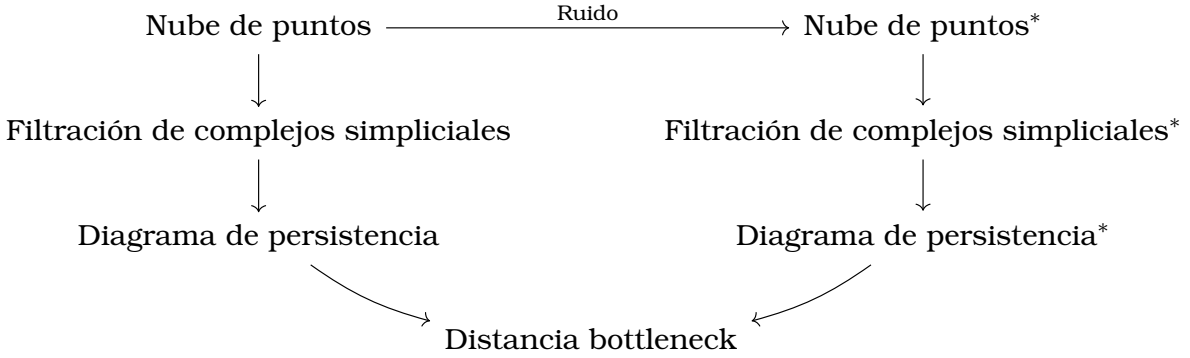


Figura 1.1: Pipeline para la comprobación del Teorema de estabilidad

1.3. Estructura del trabajo

En la sección 2.1 se introducirán las nociones topológicas básicas para hacer autocontenido el trabajo. En ella introduciremos los *complejos simpliciales*, la *homología* y la *persistencia*. Continuando con la sección 2.2, donde enunciaremos y demostraremos el *Teorema de estabilidad*, usando como referencia el siguiente artículo [4]. Y terminando con la sección 2.3, en la que entraremos en detalle en los algoritmos propuestos para la implementación de las distancias *Bottleneck* y *Hausdorff*, mostrando diversas pruebas para ilustrar los resultados del teorema.

Capítulo 2

Desarrollo

2.1. Conocimientos previos y definiciones

En esta sección se introducirán las nociones topológicas básicas para hacer autocontenido este trabajo. Dichas nociones nos darán el contexto y conocimientos necesarios para poder profundizar en el *Teorema de Estabilidad* y ser capaces de abordar su demostración.

2.1.1. Complejos Simpliciales

Una forma de representar algunos espacios topológicos es a través de su descomposición en piezas más sencillas. Una descomposición de estas características se denomina complejo si sus piezas son topológicamente simples y sus intersecciones son piezas del mismo tipo, pero de dimensión inferior [2]. Existe una gran variedad de complejos con distintos grados de abstracción. En este trabajo nos centraremos en los complejos simpliciales, que permiten representar una gran variedad de espacios y son especialmente adecuados para cuestiones computacionales.

Los complejos simpliciales pueden ser estudiados desde un enfoque geométrico y desde un enfoque combinatorio. Partiremos de la definición de complejo simplicial desde el punto de vista geométrico. Para ello recordaremos algunos conceptos de geometría afín.

Definición 2.1.1. El conjunto de puntos $\{u_0, u_1, ..., u_k\}$ de $\mathbb{R}^d$ es *afínmente independiente* si los vectores $\{\overrightarrow{u_0u_1}, ..., \overrightarrow{u_0u_k}\}$ son linealmente independientes.

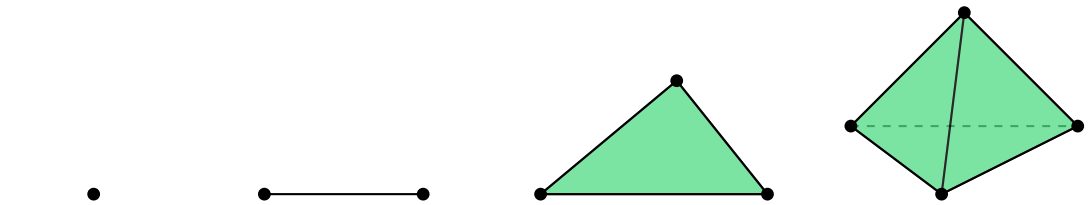
Definición 2.1.2. Diremos que $x \in \mathbb{R}^d$ es *combinación convexa* de los puntos $u_0, u_1, ..., u_k$ si $x = \sum_{i=0}^{k} \lambda_i u_i$ con $\lambda_i \geq 0$ para todo $i \in \{0, ..., k\}$ y $\sum_{i=0}^{k} \lambda_i = 1$.

Definición 2.1.3. Llamaremos *envolvente convexa* de $u_0, u_1, ..., u_k$, denotado por $\text{conv}\{u_0, u_1, ..., u_k\}$, al conjunto de todas las combinaciones convexas de dichos puntos.

Haciendo uso de este conjunto podremos definir nuestras piezas de la descomposición de la siguiente manera:

Definición 2.1.4. Un *k-símplice* σ en $\mathbb{R}^d$ con $d \geq k$ es la envolvente convexa de $k+1$ puntos afínmente independientes $u_0, u_1, ..., u_k \in \mathbb{R}^d$, es decir, $\sigma \coloneqq \text{conv}\{u_0, u_1, ..., u_k\}$.

Diremos que el k-símplice σ tiene dimensión k y llamaremos *vértices de* σ a los puntos $u_0, u_1, ..., u_k$.



(a) 0-símplice (b) 1-símplice (c) 2-símplice (d) 3-símplice

Figura 2.1: Representación de los símplices de dimensión 0, 1, 2 y 3

Se puede observar que cualquier subconjunto de los vértices de σ será afínmente independiente y por lo tanto definirá un símplice τ de dimensión inferior. De esta forma diremos que τ *es una cara de* σ si es una combinación convexa de un subconjunto no vacío de los vértices de σ, y lo denotaremos por $\tau \leq \sigma$. Si el subconjunto es propio, diremos que τ *es cara propia de* σ, y lo denotaremos por $\tau < \sigma$. Por otro lado, diremos que σ *es cocara (propia) de* τ si $\sigma \geq \tau$ ($\sigma > \tau$).

Haciendo uso de la definición de caras de un símplice σ podemos definir *el borde y el interior* de σ.

Definición 2.1.5. Sea σ un símplice. Entonces

- Se define el *borde de* σ como
$$\text{bd } \sigma = \bigcup_{\tau < \sigma} \tau \,.$$
- Se define el *interior de* σ como
$$\text{int } \sigma = \sigma - \text{bd } \sigma \,.$$

Observación. Se sigue directamente de la definición que un punto $x \in \sigma$ pertenece al interior de σ si y sólo si todos sus coeficientes λ_i de la combinación convexa son positivos. Se sigue que cada punto $x \in \sigma$ pertenece únicamente al interior de la cara generada por los puntos con coeficientes λ_i positivos.

Una vez que ya conocemos las piezas de nuestra descomposición vamos a ver como tenemos que unirlas y cuáles son las principales propiedades de los complejos resultantes.
Como ya hemos visto al principio de la sección, para que una descomposición sea un complejo sus piezas tienen que ser topológicamente simples y sus intersecciones tienen que ser piezas de dimensión inferior del mismo tipo. La manera natural de hacer esto es pegar unos símplices con otros por sus caras.

Definición 2.1.6. Un *complejo simplicial* es una colección finita de símplices K que satisface las siguientes propiedades:

A. Si $\sigma \in K$ y $\tau \leq \sigma$ entonces $\tau \in K$.

B. Si $\sigma_0, \sigma_1 \in K$ y $\sigma_0 \cap \sigma_1 \neq \emptyset$ entonces $\sigma_0 \cap \sigma_1 \leq \sigma_i$ para $i = 1, 2$.

Se define la dimensión de como el máximo de las dimensiones de sus símplices.

Un ejemplo de complejo simplicial es lo que se muestra en la figura 2.2, mientras que en la figura 2.3 muestra un ejemplo que no es complejo simplicial.

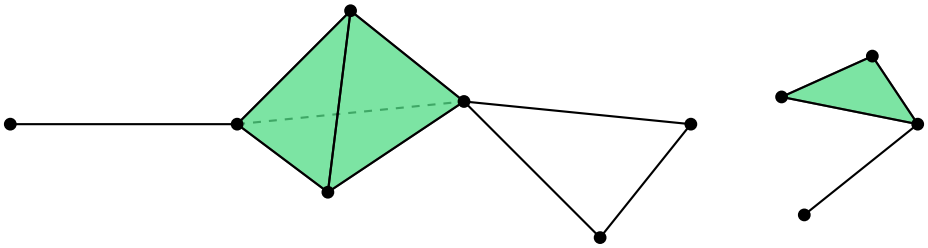


Figura 2.2: Ejemplo de complejo simplicial

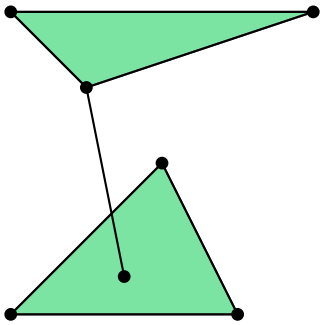


Figura 2.3: Ejemplo de conjunto de símplices que no cumplen las condiciones de complejo simplicial

Definición 2.1.7. El *espacio subyacente* de un complejo simplicial K, denotado $|K|$, es la unión de los símplices de K con la topología heredada del $\mathbb{R}^d$ donde viven sus símplices. Este espacio subyacente también es llamado *poliedro*.

Como se puede observar, el espacio subyacente de un complejo simplicial es compacto, siendo unión finita de símplices. El siguiente resultado caracteriza los abiertos y cerrados del espacio subyacente $|K|$ de un complejo simplicial K.

Proposición 2.1.1 ([2])**.** *Sea K un complejo simplicial y $A \subset |K|$ un subconjunto. Entonces A es un abierto (cerrado) en K si y sólo si para cada $\sigma \in K$, $A \cap |\sigma|$ es un abierto (cerrado) de $|\sigma|$.*

Definición 2.1.8. Una *triangulación* de un espacio topológico X es un par (K, h) donde K es un complejo simplicial y $h : X \to |K|$ es un homeomorfismo (h continua, biyectiva y h^{-1} continua).

Diremos que un espacio topológico es *triangulable* si admite una triangulación.

También nos será de utilidad poder estudiar los complejos simpliciales contenidos en otro complejo simplicial.

Definición 2.1.9. Un *subcomplejo* L de un complejo simplicial K es un complejo simplicial $L \subseteq K$.

Un subcomplejo de gran interés son los j*-esqueletos*, definidos de la siguiente forma:

$$K^{(j)} = \{\sigma \in K \mid dim\ \sigma\ \leq j\}\,.$$

Otro subconjunto de símplices que nos será de gran ayuda más adelante es la *estrella de un símplice* τ, la cual consiste de las cocaras de τ, denotado por St τ. Este conjunto no será siempre un complejo simplicial, así que se define la *estrella cerrada* $\overline{\text{St}}\ \tau$ como el menor subcomplejo de K que contiene a St τ. Adicionalmente, se define el *link* de τ como: Lk $\tau = \{v \in \overline{\text{St}}\ \tau \mid v \cap \tau = \emptyset\}$.

Complejos simpliciales abstractos

Una vez que ya conocemos los complejos simpliciales desde el punto de vista geométrico, vamos a abordarlos desde un enfoque combinatorio, el cual nos será de gran ayuda para poder programar los complejos simpliciales.

Definición 2.1.10. Un *complejo simplicial abstracto* A es una colección finita de conjuntos finitos tal que si $\alpha \in A$ y $\beta \subset \alpha$ entonces $\beta \in A$.

De esta forma se cumple que

- Los conjuntos en A no vacíos se denominan *símplices abstractos*.
- La *dimensión* de un símplice abstracto $\alpha \in A$ es dim $\alpha = \text{card}(\alpha) - 1$. Y la dimensión del complejo es el máximo de las dimensiones de sus símplices.
- Una *cara* de $\alpha \in A$ es cualquier subconjunto no vacío de $\beta \subset \alpha$.
- El *conjunto de vértices* de A, denotado por Vert A, es la unión de todos sus símplices.
- Un *subcomplejo* B de un complejo simplicial abstracto A es un complejo simplicial abstracto $B \subset A$.

Ejemplo 2.1.1. Un ejemplo de complejo simplicial abstracto es el siguiente conjunto

$$A = \{\{0\},\{1\},\{2\},\{3\},\{4\},\{5\},\{6\},\{0,1\},\{1,2\},\{1,3\},\{1,4\},\{2,3\},\{2,4\},\{3,4\},\{4,5\},\\ \{4,6\},\{5,6\},\{1,2,3\},\{1,2,4\},\{1,3,4\},\{2,3,4\},\{1,2,3,4\}\}.$$

Donde el conjunto de vértices es: Vert $A = \{0,1,2,3,4,5,6\}$.

Definición 2.1.11. Sean A y B dos complejos simpliciales abstractos. Diremos que A y B son *isomorfos* si existe una biyección

$$b : \text{Vert } A \to \text{Vert } B$$

tal que $\alpha \in A$ si y sólo si $b(\alpha) \in B$.

Cada complejo geométrico induce de manera natural un complejo abstracto de la siguiente forma:

Definición 2.1.12. Sea K un complejo simplicial y V el conjunto de vértices de K. Llamaremos *esquema de vértices* al complejo simplicial abstracto A formado por todos aquellos subconjuntos de V que generan símplices en K.

Y bajo ciertas circunstancias podremos hacer el paso opuesto de construir un complejo simplicial (geométrico) a partir de otro abstracto:

Definición 2.1.13. Sean A un complejo simplicial abstracto y K un complejo simplicial. Diremos que K es una *realización geométrica* de A, si A es isomorfo al esquema de vértices de K.

Teorema 2.1.1 ([2])**.** *Todo complejo simplicial abstracto de dimensión d admite una realización geométrica en $\mathbb{R}^{2d+1}$.*

Así pues, los complejos simpliciales abstractos son una representación fiel de un complejo simplicial (geométrico).

Aplicaciones simpliciales

Una vez que ya conocemos las principales propiedades de los complejos simpliciales, veremos cuales son las aplicaciones que preservan la estructura de complejo simplicial. Como vimos anteriormente, cada punto de un k-símplice pertenece al interior de exactamente una cara. Por lo tanto, todo punto $x \in |K|$, siendo K un complejo simplicial de vértices $u_0, u_1, ..., u_n$, pertenece al interior de uno de los símplices de K. Si $\sigma = \text{conv}\{u_0, u_1, ..., u_k\}$ es dicho símplice, entonces $x = \sum_{i=0}^{n} b_i(x)u_i$, donde

$$b_i(x) = \begin{cases} \lambda_i & \text{si } 0 \leq i \leq k \\ 0 & \text{si } k+1 \leq i \leq n \end{cases}, \text{ con } \lambda_i \text{ tal que } x = \sum_{i=0}^{k} \lambda_i u_i$$

se denominan *coordenadas baricéntricas* de x en K.

Haremos uso de estas coordenadas para construir una función, lineal a trozos inducida por una función entre los vértices de dos complejos simpliciales, denominada *aplicación de vértices*

Definición 2.1.14. Sean K y L complejos simpliciales y $\varphi : \text{Vert } K \to \text{Vert } L$ una aplicación. Diremos que φ es una *aplicación de vértices* si satisface que para cada $\sigma \in K$ su imagen $\varphi(\sigma) \in L$.

Una aplicación de vértices $\varphi : \text{Vert } K \to \text{Vert } L$ induce una aplicación, lineal a trozos $f : |K| \to |L|$ dada por

$$f(x) = f\left(\sum_{i=0}^{n} b_i(x)u_i\right) = \sum_{i=0}^{n} b_i(x)u_i$$

a la que llamaremos *aplicación simplicial* asociada a φ. Para enfatizar que es una aplicación lineal en cada símplice del complejo, se suele notar la aplicación de la siguiente forma $f : K \to L$.

Subdivisiones

Veremos que hay ocasiones que nos interesará controlar el tamaño de los símplices de nuestro complejo simplicial conservando el espacio subyacente. Por esta razón, se introduce la noción de *subdivisión de un complejo simplicial*.

Definición 2.1.15. Sea K un complejo simplicial. Diremos que un complejo simplicial L es una *subdivisión* de K si:

- $|K| = |L|$.
- Cada símplice de L está contenido en un símplice de K.

Hay muchas maneras de obtener subdivisiones de un complejo simplicial, pero un tipo particular de subdivisión que es muy utilizada es la *subdivisión baricéntrica*, denotada por $L = \text{Sd}K$. Para la construcción de esta subdivisión, introducimos el *baricentro* de un símplice y el *cono* de un símplice de vértice v.

Definición 2.1.16. Sea σ un k-símplice, tal que $\sigma = \text{conv}\{v_0, v_1, ..., v_k\}$. Llamaremos *baricentro* de σ al punto

$$b_\sigma = \sum_{i=0}^{k} \frac{v_i}{k+1} \in \text{int}\, \sigma\,.$$

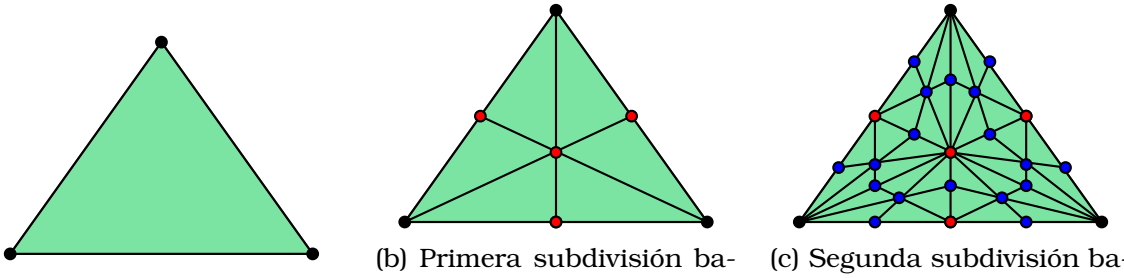
Definición 2.1.17. Sea σ un k-símplice, tal que $\sigma = \text{conv}\{v_0, v_1, ..., v_k\}$ y v un punto no contenido en el subespacio afín generado por $\{v_0, v_1, ..., v_k\}$. Se define el *cono* de σ con vértice v y se denota por $\sigma * v$ como el $k+1$-símplice generado por $\{v, v_0, v_1, ..., v_k\}$.

Definición 2.1.18. Sea K un complejo simplicial. Se define la *subdivisión baricéntrica* de K como el complejo simplicial $\text{Sd}K$ que se construye inductivamente sobre el j-esqueleto como sigue:

A. $\text{Sd}K^{(0)} = K^{(0)}$.

B. $\text{Sd}K^{(j)}$ es la unión de $\text{Sd}K^{(j-1)}$ con el conjunto de todos los símplices de la forma $b_\sigma * \tau$, donde σ es un j-símplice y τ es cualquier símplice de $\text{Sd}K^{(j-1)}$ contenido en una cara de σ.

En la figura 2.4 se muestra la primera y segunda subdivisión baricéntrica de un complejo simplicial.



(a) 2-símplice (b) Primera subdivisión baricéntrica (c) Segunda subdivisión baricéntrica

Figura 2.4: Primera y segunda subdivisión baricéntrica de un 2-símplice

Recordemos que el *diámetro* de un subconjunto $A \subset \mathbb{R}^d$ es el supremo sobre las distancias entre sus puntos.

Lema 2.1.2 ([2])**.** *Si σ es un k-símplice, entonces el diámetro de cada símplice en la subdivisión baricéntrica de σ es como máximo $\frac{k}{k+1}$*diam σ*.*

De forma que gracias al lema anterior podremos hacer el diámetro de los símplices de los complejos simpliciales tan pequeño como queramos, ya que el diámetro de los símplices de la n-ésima subdivisión baricéntrica del complejo simplicial K, denotado por $\text{Sd}^n K = \text{Sd}(\text{Sd}^{n-1}K)$, es

$$\left(\frac{k}{k+1}\right)^n \text{diam}\, \sigma \underset{n\to\infty}{\longrightarrow} 0, \text{ con } \sigma \in K \text{ y } k = \dim \sigma\,.$$

Aproximaciones simpliciales

Para estudiar la topología de los poliedros es fundamental aproximar funciones continuas por aplicaciones simpliciales. Para poder definir estas aproximaciones primero vamos a definir un tipo de entorno de los vértices de un complejo como se puede ver en la figura 2.5.

Definición 2.1.19. Sea K un complejo simplicial y v un vértice de K. El conjunto

$$N(v) = \bigcup_{\sigma \in \mathrm{St}\ v} \mathrm{int}\ \sigma$$

es un entorno abierto de v en $|K|$ al que llamaremos *entorno estrellado* de v.

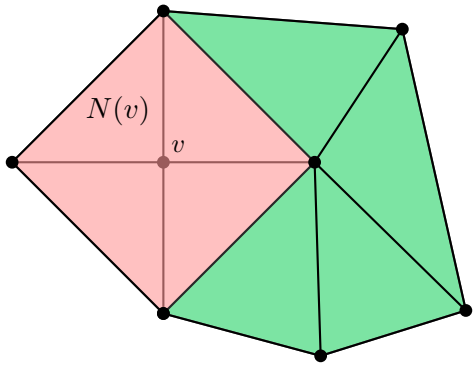


Figura 2.5: Entorno estrellado de v marcado en color rojo

Así pues, definimos una aproximación simplicial de la siguiente forma:

Definición 2.1.20. Sean K y L complejos simpliciales, $g : |K| \to |L|$ una aplicación continua y $f : K \to L$ una aplicación simplicial. Diremos que f es una *aproximación simplicial* de g si verifica la *condición de estrella*, es decir, si para cada vértice $v \in K$ se tiene que $g(N(v)) \subset N(f(v))$.

Además, la condición de estrella será una condición suficiente para garantizar la existencia de una aproximación simplicial:

Lema 2.1.3 ([2])**.** *Sean K y L complejos simpliciales, $g : |K| \to |L|$ una aplicación continua que satisface la condición de estrella. Entonces g tiene una aproximación simplicial $f : K \to L$.*

En la figura 2.6 podemos ver un ejemplo de aproximación simplicial de una aplicación continua.

Teorema 2.1.4 (Aproximación simplicial [2])**.** *Sean K y L complejos simpliciales, $g : |K| \to |L|$ una aplicación continua. Entonces existe $n \in \mathbb{N}$ tal que g tiene una aproximación simplicial $f : \mathrm{Sd}^n\ K \to L$.*

2.1.2. Complejos simpliciales de nubes de puntos

Desde el punto de vista computacional nos encontramos con el problema de que tenemos una representación de un espacio topológico a través de una discretización finita de los puntos de dicho espacio, y nuestro objetivo es poder recuperar propiedades del espacio topológico original a partir de esta nube de puntos. Para ello asociaremos complejos simpliciales a dicha nube de puntos.

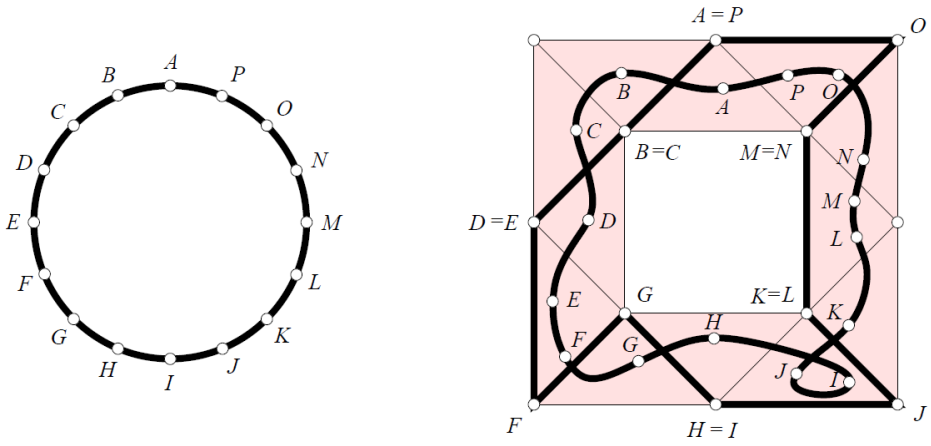


Figura 2.6: Aplicación continua del círculo en una corona circular y una aproximación simplicial de dicha aplicación. Fuente: [2]

Complejo de Čech

El complejo de Čech se define a partir de la intersección de una colección de bolas cerradas. La idea que subyace a esta construcción es la del nervio de una colección, que se introduce a continuación.

Definición 2.1.21. Sea F una colección finita de conjuntos. Se define el *nervio* de F como el complejo simplicial abstracto

$$\text{Nrv } F = \left\{ X \subseteq F \mid \bigcap X \neq \emptyset \right\}.$$

Consideramos el caso particular en el que los conjuntos de la familia son las bolas cerradas $\overline{B}_r(x) = \{y \in \mathbb{R}^d \mid d(x,y) \leq r\}$ en $\mathbb{R}^d$.

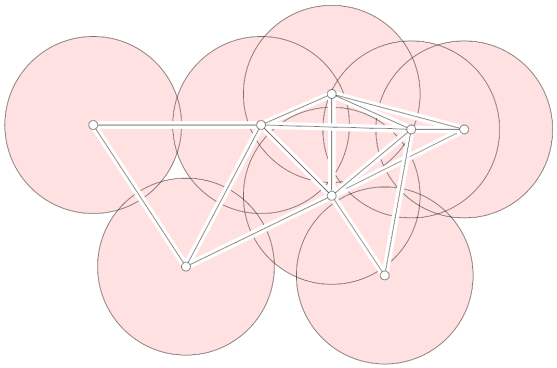


Figura 2.7: Complejo de Čech para un conjunto de nueve puntos y un radio r. Fuente: [2]

Definición 2.1.22. Sea $S \subset \mathbb{R}^d$ un conjunto finito de puntos. Llamaremos *complejo de Čech* de S de radio r al complejo simplicial abstracto

$$\check{\text{C}}\text{ech}(r) = \left\{ \sigma \subset S \mid \bigcap_{u \in \sigma} \overline{B}_r(u) \neq \emptyset \right\}.$$

El complejo de Čech es isomorfo al nervio de la colección de las bolas cerradas de radio r centrada en los puntos de S. En la figura 2.7 podemos observar un ejemplo de complejo de Čech.

Podemos comprobar [2] que para valores de r lo suficientemente grandes, Čech(r) es un símplice de dimensión card$(S)-1$, por lo que el complejo de Čech es poco eficiente desde el punto de vista computacional.
Además, en general, el complejo de Čech de un conjunto de puntos $S \subset \mathbb{R}^d$ no posee una realización geométrica en $\mathbb{R}^d$.

Complejo de Vietoris-Rips

Definición 2.1.23. Sea $S \subset \mathbb{R}^d$ un conjunto finito de puntos. Llamamos *complejo de Vietoris-Rips* de S de radio r al complejo simplicial abstracto

$$\text{VR}(r) = \{\sigma \subseteq S \mid \text{diam } \sigma \leq 2r\}$$

donde diam σ denota el diámetro del subconjunto σ.

En la figura 2.8 podemos observar como se generan los diversos complejos de VR a medida que se va aumentando el radio.

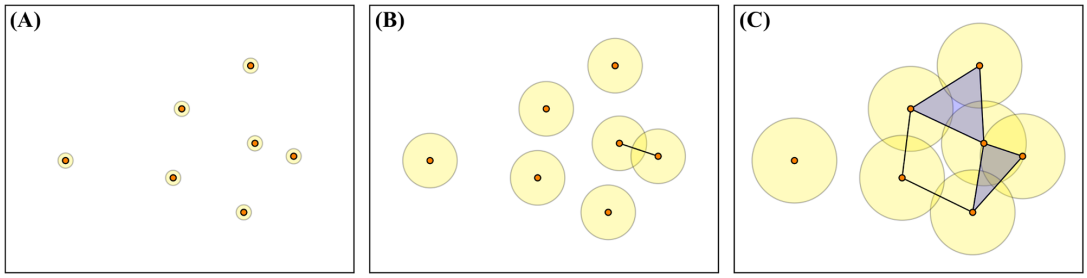


Figura 2.8: Complejos de Vietoris-Rips para un conjunto de siete puntos a medida que aumentamos el radio de izquierda a derecha. Fuente: [5]

Sea $\sigma \subset S$, entonces recordamos que el diámetro se define como

$$\text{diam } \sigma = \max_{u,v \in \sigma} d(u,v)\,.$$

Esta observación garantiza que $\sigma \in \text{VR}(r)$ si y sólo si todas sus aristas están en VR(r). Dicho de otro modo, VR(r) está completamente determinado por su 1-esqueleto. Esto hace que el complejo de Vietoris-Rips sea mucho más eficiente que el complejo de Čech desde el punto de vista computacional. Sin embargo, al igual que ocurre con el complejo de Čech, no admite una realización geométrica en $\mathbb{R}^d$.

Por otro lado, el complejo de Vietoris-Rips no es el nervio de ningún recubrimiento. Sin embargo, el siguiente resultado garantiza que el complejo de VR aproxima al complejo de Čech.

Lema 2.1.5 (Lema de Vietoris-Rips [2])**.** *Sea $S \subset \mathbb{R}^d$ un conjunto finito de puntos y sea $r \geq 0$. Entonces,*

$$\text{Čech}(r) \subset \text{VR}(r) \subset \text{Čech}(\sqrt{2}r)\,.$$

Complejo de Delaunay

En esta sección introduciremos construcciones geométricas que nos limitarán la dimensión de los símplices que obtenemos del nervio de una colección finita de conjuntos.

Definición 2.1.24. Sea $S \subset \mathbb{R}^d$ un conjunto finito. Se define la *celda de Voronoi* de un punto $u \in S$ como el conjunto de los puntos

$$V_u = \{x \in \mathbb{R}^d \mid d(x,u) \leq d(x,v), \text{ para todo } v \in S\}.$$

La colección de las celdas de Voronoi de los puntos de S se denomina *diagrama de Voronoi* de S.

En la figura 2.9 se puede ver el diagrama de Voronoi de un conjunto de puntos. Nótese que las celdas de Voronoi recubren todo el espacio.

Definición 2.1.25. Sea $S \subset \mathbb{R}^d$ un conjunto finito. Se define el *complejo de Delaunay* de S como el complejo simplicial abstracto

$$\text{Del} = \left\{ \sigma \subseteq S \mid \bigcap_{u \in \sigma} V_u \neq \emptyset \right\}.$$

Definición 2.1.26 ([6])**.** Un conjunto de puntos en un espacio afín d-dimensional está en *posición general* si ningún subconjunto de k puntos está contenido en un subespacio afín $(k-2)$-dimensional, para $k = 2, 3, ..., d+1$.

El complejo de Delaunay es un complejo isomorfo al nervio del diagrama de Voronoi. Además, si los puntos de S están en posición general, se obtiene una realización del complejo de Delauney en $\mathbb{R}^d$ considerando envolventes convexas de los símplices abstractos. Esta realización geométrica se denomina *triangulación de Delaunay*.

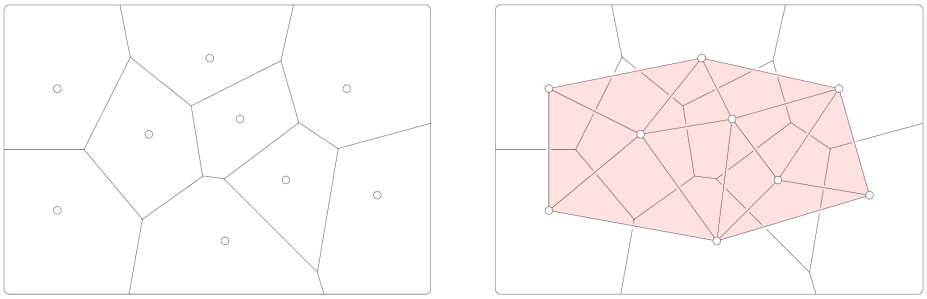


Figura 2.9: A la izquierda tenemos el Diagrama de Voronoi de un conjunto de nueve puntos en el plano, y a la derecha triangulación de Delaunay superpuesta al diagrama de Voronoi. Fuente: [2]

Alfa complejo

Sea $S \subset \mathbb{R}^d$ un conjunto finito de puntos y $r \geq 0$. Para cada $u \in S$ consideramos la región $R_u(r) = \overline{B}_r(u) \cap V_u$, es decir, la intersección de la región de Voronoi de u con la bola cerrada de centro u y radio r.

Definición 2.1.27. Sea $S \subset \mathbb{R}^d$ un conjunto finito de puntos y $r \geq 0$.Se define el *Alfa complejo* de radio r asociado a S como el complejo simplicial abstracto

$$\text{Alpha}(r) = \left\{ \sigma \in S \mid \bigcap_{u \in \sigma} R_u(r) \neq \emptyset \right\}.$$

En la figura 2.10 se puede observar la unión de dichas regiones y su correspondiente alfa complejo. Se puede observar que el alfa complejo es isomorfo al nervio de la colección formada por los $R_u(r)$.

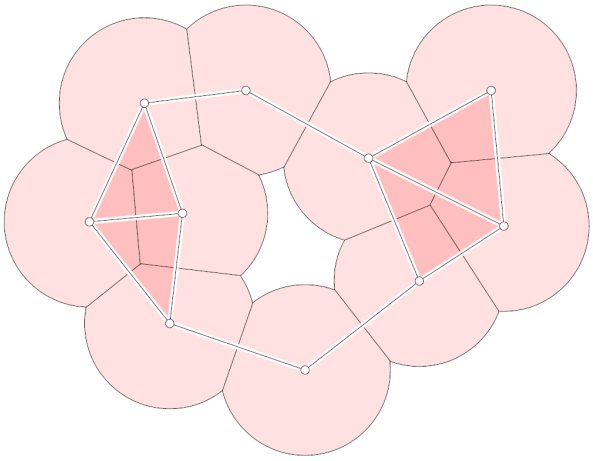


Figura 2.10: Unión de las regiones $R_u(r)$ asociadas a un radio r y un conjunto finito de puntos S. El correspondiente alfa complejo es superpuesto a esta unión de regiones. Fuente: [2]

Puesto que $R_u(r) \subset \overline{B}_r(u)$ para cada $u \in S$, se tiene que $\text{Alpha}(r) \subset \check{\text{C}}\text{ech}(r)$. Del mismo modo, dado que $R_u(r) \subset V_u$ para cada $u \in S$, se tiene que $\text{Alpha}(r) \subset \text{Del}(S)$.

Además, el alfa complejo tiene menos símplices que el complejo de Čech. Y como es un subcomplejo del complejo de Delaunay, admite de manera natural una realización en $\mathbb{R}^d$. Por lo que hace que los alfa complejos sean una buena opción desde el punto de vista computacional.

2.1.3. Homología

Como se puede ver en [7], la homotopía es una herramienta algebraica para poder obtener propiedades de los espacios topológicos. Sin embargo, los métodos para el cálculo de la homotopía no son manejables computacionalmente. Así pues, se propone la homología como formalismo algebraico, que, aunque no es capaz de obtener tanta información topológica sobre el espacio como con otros formalismos, es muy computable.

Comenzaremos estudiando los diversos grupos que están involucrados en la definición de la homología.

Grupos de cadenas

Sea K un complejo simplicial y p un número entero no negativo. Una *p-cadena* en K es una suma formal de p-símplices en K. Más concretamente, c es una p-cadena en

K si

$$c = \sum a_i \sigma_i$$

con σ_i es un p-símplice para cada i y a_i son los *coeficientes*. Estos coeficientes pueden tomarse de cualquier anillo conmutativo, sin embargo, nosotros usaremos con coeficientes en el cuerpo de dos elementos, es decir, $a_i \in \mathbb{Z}_2$.

Ejemplo 2.1.2. Escribiremos los símplices como la lista de sus vértices, $\sigma = [u_0, u_1, ..., u_p]$.

- En la figura 2.11 se muestra en rojo la 0-cadena $c = [0] + [2] + [6] + [9]$.

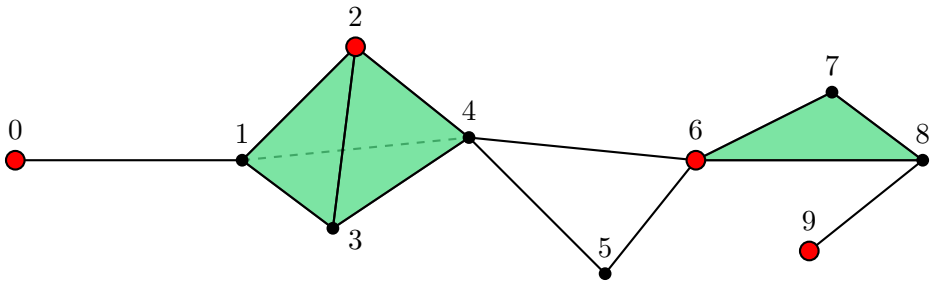


Figura 2.11: Ejemplo de 0-cadena

- En la figura 2.12 se muestra en rojo la 1-cadena $c = [0, 1] + [1, 2] + [2, 4] + [8, 9]$.

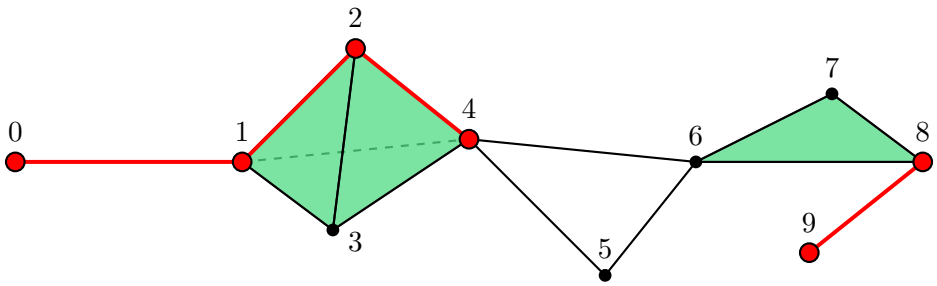


Figura 2.12: Ejemplo de 1-cadena

- En la figura 2.13 se muestra en rojo la 2-cadena $c = [1, 2, 3] + [2, 3, 4] + [6, 7, 8]$.

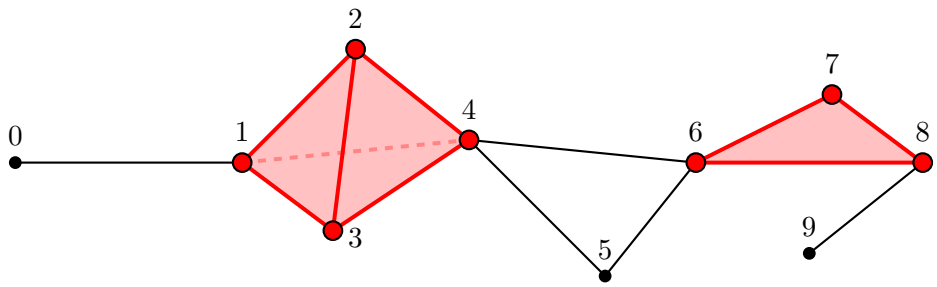


Figura 2.13: Ejemplo de 2-cadena

Dadas dos p-cadenas $c = \sum a_i \sigma_i$ y $c' = \sum b_i \sigma_i$, se define su suma como

$$c + c' = \sum (a_i + b_i)\sigma_i \,.$$

Las p-cadenas con la operación suma $+$ forman el *grupo de p-cadenas* denotado por $(\mathrm{C}_p, +)$, pero como la operación se sobrentiende, se suele nombrar como $\mathrm{C}_p = \mathrm{C}_p(K)$.

Este grupo es un grupo abeliano, y como en nuestro caso los coeficientes están tomados en el cuerpo $\mathbb{Z}_2$, $\mathrm{C}_p(K)$ es un espacio vectorial sobre $\mathbb{Z}_2$. Fijado $p \in \mathbb{Z}$, una base del espacio vectorial $\mathrm{C}_p(K)$ es el conjunto $\{\sigma_i^p \mid i = 1, ..., s_p\}$ formado por los símplices de dimensión p de K. Como consecuencia $\mathrm{C}_p(K) = \{0\}$, siendo $0 = \sum 0 \cdot \sigma_i$, si $p < 0$ ó $p > \dim(K)$.

Operador borde

Para poder relacionar estos grupos definiremos el *operador borde*, así pues, partiremos con la definición del borde de un símplice.

Definición 2.1.28. Sea p un número entero y $\sigma \in K$ un p-símplice $\sigma = [v_0, v_1, ..., v_p]$ se define su *borde*, $\partial_p \sigma$, como la suma formal de sus caras $(p-1)$-dimensionales, es decir,

$$\partial_p \sigma = \sum_{j=0}^{p} [v_0, ..., \hat{v}_j, ..., v_p]$$

donde $\hat{v}_j$ denota que v_j se omite.

En general, dada una p-cadena $c = \sum a_i \sigma_i$, se define su borde mediante la extensión lineal como $\partial_p c = \sum_{j=0}^{p} a_i \partial_p \sigma_i$. Como consecuencia, el borde define una aplicación lineal $\partial_p : \mathrm{C}_p \to \mathrm{C}_{p-1}$ entre espacios vectoriales de cadenas denominada *operador borde*. Para simplificar la notación suele omitirse el subíndice p del operador borde, ya que siempre coincide con la dimensión de la cadena a la que se le aplica.

Ejemplo 2.1.3. Sea la 2-cadena $c = [0,1] + [4,5]$, entonces el borde de c es:

$$\partial c = \partial[0,1] + \partial[4,5] = [0] + [1] + [4] + [5]\,.$$

Ciclos y bordes

Distinguiremos dos tipos de cadenas, las cuales usaremos para poder definir los grupos de homología.

Definición 2.1.29. Diremos que una p-cadena c es un *p-ciclo* si

$$\partial c = 0$$

o, equivalentemente, si $c \in \ker \partial$.

Debido a que ∂ conmuta con la suma $+$, el conjunto de p-ciclos $\mathrm{Z}_p = \ker \partial_p$ es un subgrupo (subespacio vectorial en nuestro caso) de C_p.

Ejemplo 2.1.4. Veremos que geométricamente los p-ciclos representan ciclos en el complejo simplicial. Estos a su vez pueden ser agujeros de dimensión p. En la figura 2.14 se muestra en rojo el 1-ciclo $[4,5] + [4,6] + [5,6]$, el cual es un agujero. Mientras que en azul se representa el 1-ciclo $[6,7] + [6,8] + [7,8]$, que no es un agujero.

Definición 2.1.30. Diremos que una p-cadena c es un *p-borde* si existe una $(p+1)$-cadena c' tal que

$$\partial c' = c$$

o, equivalentemente, si $c \in \mathrm{im}\, \partial_{p+1}$.

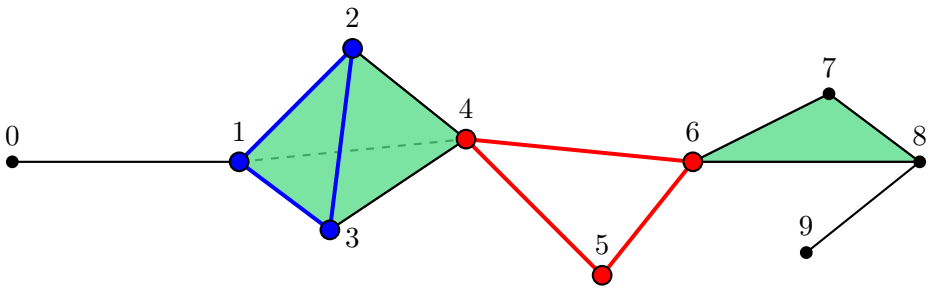


Figura 2.14: Ejemplos de 1-ciclos

Debido a que ∂ conmuta con la suma $+$, el conjunto de p-bordes $\mathrm{B}_p = \text{im } \partial_{p+1}$ es un subespacio vectorial de C_p.

Ejemplo 2.1.5. El 1-ciclo que habíamos destacado en azul en la figura 2.14 es un 1-borde.

Probaremos que los p-bordes son p-ciclos, como ocurre en el ejemplo. Para ello enunciaremos el siguiente lema.

Lema 2.1.6 (Lema fundamental de la homología [2])**.** $\partial_p \partial_{p+1} c = 0$ *para todo entero* p *y toda* $(p+1)$*-cadena* c*.*

Se sigue que B_p es un subespacio vectorial de Z_p, es decir $\mathrm{B}_p \subset \mathrm{Z}_p$. Además, podemos definir el *complejo de cadenas* asociado a un complejo simplicial K como la sucesión de grupos de cadenas conectados por los operadores borde

$$\ldots \xrightarrow{\partial_{p+2}} \mathrm{C}_{p+1} \xrightarrow{\partial_{p+1}} \mathrm{C}_p \xrightarrow{\partial_p} \mathrm{C}_{p-1} \xrightarrow{\partial_{p-1}} \ldots$$

La figura 2.15 muestra esta relación entre el grupo de cadenas C_p, el grupo de ciclos Z_p y el grupo de bordes B_p; y sus conexiones generadas por el operador borde.

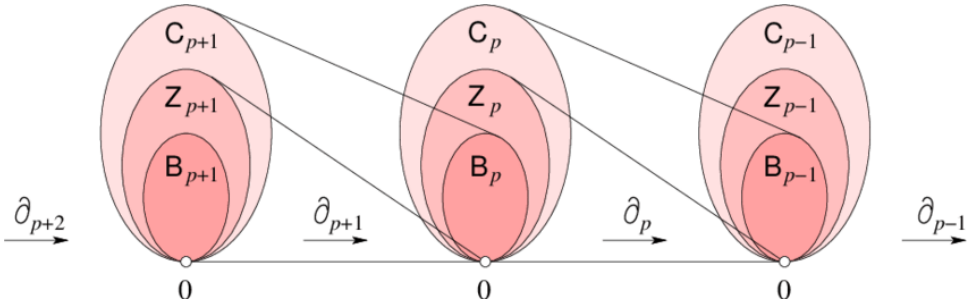


Figura 2.15: Complejo de cadenas representando el grupo de cadenas, el grupo de ciclos y el grupo de bordes. Fuente: [2]

Grupos de homología simplicial

La idea general de los grupos de homología es poder encontrar los agujeros a partir de los ciclos. Para ello tendremos que "descartar" aquellos ciclos que son bordes. Es por esto por lo que cocientaremos el grupo de los ciclos por el grupo de bordes, ya que así todos los bordes serán triviales en homología.

Definición 2.1.31. Dado un complejo simplicial K se define su *grupo de homología p-dimensional* como el cociente

$$\mathrm{H}_p(K) = \frac{\mathrm{Z}_p}{\mathrm{B}_p}\,.$$

El *número de Betti p-dimensional* $\beta_p(K)$ como la dimensión de $\mathrm{H}_p(K)$.

Luego los elementos $z \in \mathrm{H}_P = \mathrm{H}_p(K)$ son de la forma $z = c + \mathrm{B}_p$ con $c \in \mathrm{Z}_p$, donde $c+\mathrm{B}_p$ es la *clase lateral* de B_p en Z_p. Dos ciclos $c_1, c_2 \in \mathrm{Z}_p$ representan la misma *clase de homología* $z \in \mathrm{H}_p$ si y sólo si $z = c_1 + \mathrm{B}_p = c_2 + \mathrm{B}_p$; lo que equivale a que $(c_1 - c_2) \in \mathrm{B}_p$.

Definición 2.1.32. Diremos que dos ciclos $c_1, c_2 \in \mathrm{Z}_p$ son *homólogos* si existe $b \in \mathrm{B}_p$ tal que

$$c_1 = c_2 + b\,.$$

Como $\mathrm{H}_p(K)$ es un grupo finito, por el *teorema de Lagrange* sabemos que el número de clases de homología es

$$\text{ord } \mathrm{H}_p(K) = \frac{\text{ord } \mathrm{Z}_p}{\text{ord } \mathrm{B}_p}\,.$$

Además, como Z_p, B_p y H_p son espacios vectoriales sobre $\mathbb{Z}_2$ se sigue que

$$\beta_p = \dim \mathrm{H}_p = \dim \mathrm{Z}_p - \dim \mathrm{B}_p\,.$$

Aplicaciones inducidas

Veremos que una aplicación simplicial entre dos complejos simpliciales lleva ciclos a ciclos y bordes a bordes. Luego, esta aplicación induce una aplicación entre grupos de homología.

Sean K y L complejos simpliciales y $f : K \to L$ una aplicación simplicial. Para cada p-símplice σ^p se define

$$f_\#(\sigma^p) = \begin{cases} f(\sigma^p) & \text{si dim } f(\sigma^p) = p \\ 0 & \text{en otro caso} \end{cases}$$

Puesto que los símplices forman una base de los espacios vectoriales $\mathrm{C}_p(K)$ y $\mathrm{C}_p(L)$, mediante una extensión lineal se obtiene una aplicación lineal $f_\# : \mathrm{C}_p(K) \to \mathrm{C}_p(L)$.

Propiedad 2.1.1 ([2])**.** Sean ∂_K y ∂_L los operadores borde de K y L respectivamente. Entonces $f_\# \circ \partial_K = \partial_L \circ f_\#$.

La propiedad anterior garantiza que $f_\#(\mathrm{Z}_p(K)) \subset \mathrm{Z}_p(L)$ y $f_\#(\mathrm{B}_p(K)) \subset \mathrm{B}_p(L)$. Por tanto $f_\#$ induce una aplicación lineal $f_* : \mathrm{H}_p(K) \to \mathrm{H}_p(L)$, que denominaremos *homomorfismo inducido por f*.

Utilizando aproximaciones simpliciales podemos ver que aplicaciones continuas entre poliedros inducen aplicaciones lineales en homología. Para ello definiremos el siguiente operador:

Definición 2.1.33. Sea K un complejo simplicial y consideremos la aplicación $\lambda : \mathrm{C}_p(K) \to \mathrm{C}_p(\mathrm{Sd}^n\ K)$ definida sobre los p-símplices como

$$\lambda_p(\sigma^p) = \sum_{\tau^p \in \mathrm{Sd}^n \sigma^p} \tau^p\,.$$

La aplicación λ_p se denomina *operador subdivisión*.

Sean K y L complejos simpliciales y $f : |K| \to |L|$ una aplicación continua y $g : \mathrm{Sd}^n\ K \to L$ una aproximación simplicial de f. Se define el *homomorfismo inducido* por la aplicación f como la aplicación lineal $f_* : \mathrm{H}_p(K) \to \mathrm{H}_p(L)$ dada por

$$f_* = g_* \circ \lambda_{p*}\,.$$

Donde g_* es el homomorfismo inducido por g y $\lambda_{p*} : \mathrm{H}_p(K) \to \mathrm{H}_p(\mathrm{Sd}^n\ K)$ es el isomorfismo inducido por λ_p.

Teorema 2.1.7. *Sean K y L dos complejos simpliciales y $f : |K| \to |L|$ un homeomorfismo. Entonces $f_* : \mathrm{H}_p(K) \to \mathrm{H}_p(L)$ es un isomorfismo para todo p.*

Propiedades topológicas

En esta sección veremos algunas propiedades topológicas que podemos obtener del estudio de la homología de un complejo simplicial.

Definición 2.1.34. La característica de Euler de un complejo simplicial K es

$$\chi(K) = \sum_{p=0}^{\dim\ K} (-1)^p s_p$$

donde $s_p = \dim\ \mathrm{C}_p(K)$.

La podremos calcular a partir de los números de Betti:

Teorema 2.1.8 ([2])**.** $\chi(K) = \sum_{p=0}^{\dim\ K} (-1)^p \beta_p(K)$.

Por el teorema 2.1.7 sabemos que si los espacios subyacentes de dos complejos simpliciales son homeomorfos, entonces sus grupos de homología son isomorfos, y por tanto tendrán la misma dimensión.

Corolario. *Sean K y L dos complejos simpliciales tales que $|K| \approx |L|$. Entonces, $\chi(K) = \chi(L)$.*

Uno de los valores más importantes que obtenemos al calcular los grupos de homología son sus correspondientes números de Betti, ya que estos nos darán mucha información sobre el espacio subyacente.

Teorema 2.1.9. *Sea K un complejo simplicial. Entonces $\beta_0(K)$ coincide con el número de componentes conexas de $|K|$.*

Corolario. *$|K|$ es conexo si y sólo si $\beta_0(K) = 1$.*

El *Teorema de dualidad de Alexander* [2] nos permite interpretar los números de Betti de un poliedro contenido en $\mathbb{R}^3$:

- $\beta_0(K)$ nos indica el número de componentes conexas.
- $\beta_1(K)$ nos indica el número de túneles.
- $\beta_2(K)$ nos indica el número de cavidades.

Homología singular

Hay una gran variedad de teorías de homología en topología. La homología que hemos definido, denominada *homología simplicial*, supone que nuestro espacio está expresado como el poliedro subyacente de un complejo simplicial. La *homología singular* generaliza la homología simplicial y permite estudiar otros espacios no triangulables [8]. Este tipo de homología tiene la ventaja que existe para cualquier espacio topológico y que facilita definir conceptos como las aplicaciones inducidas. Sin embargo, los grupos de cadenas singulares tienen dimensión infinita, lo que hace que no sea una buena opción desde el punto de vista computacional. Cabe destacar que, sobre poliedros ambas teorías coinciden [3].

Además, para el *teorema de estabilidad* no nos hará falta hacer uso de la homología singular, ya que se parte de la hipótesis de que el espacio es triangulable.

2.1.4. Persistencia

Introduciremos el concepto de persistencia primero para funciones de una variable. Después veremos en el caso de funciones morse, luego profundizaremos en el caso de los complejos simpliciales y por último para funciones tame. En esta sección seguiré [3] como referencia.

Funciones reales de una variable

Sea $f : \mathbb{R} \to \mathbb{R}$ una función suave. Recordemos que x es un *punto crítico* y $f(x)$ un *valor crítico de* f si $f'(x) = 0$. Además, un punto crítico x es *no degenerado* si $f''(x) \neq 0$. Así pues, supongamos que f sólo contiene puntos críticos no degenerados con valores críticos distintos.

Si consideramos el *conjunto de subnivel* $\mathbb{R}_t = f^{-1}(-\infty, t]$ para cada $t \in \mathbb{R}$, entonces veremos que a medida que incrementemos t, el número de componentes conexas de $\mathbb{R}_t$ permanecerá constate hasta que pasemos por un t_0 valor crítico de f. Como podemos ver en la figura 2.16, cuando pasamos por un mínimo local se crea una nueva componente conexa y cuando pasamos por un máximo local se combinan dos componentes conexas en una.

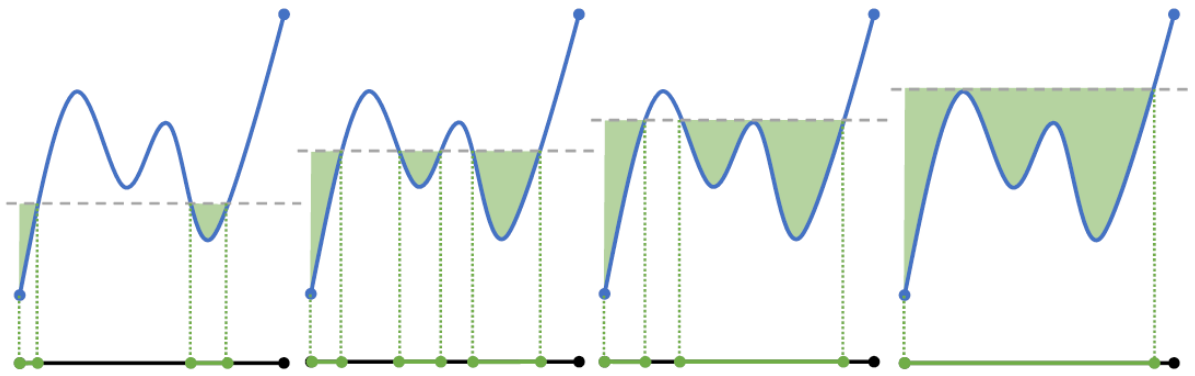


Figura 2.16: Componentes conexas en $\mathbb{R}$ en las diferentes filtraciones. Fuente: [9]

Los puntos críticos de f se emparejan de la siguiente forma:

A. Cuando aparece una nueva componente conexa, diremos que el mínimo local que lo crea *representa* esa componente.

B. Cuando pasamos por un máximo local y se juntan dos componentes, emparejamos el máximo con el mayor (el más joven) de los dos mínimos locales que representan dichas componentes. El otro mínimo (el más antiguo) pasa a ser el representante de la nueva componente resultante de juntar las dos anteriores.

Cuando los puntos x_1 y x_2 se emparejan siguiendo este método, definimos la *persistencia* del par como $f(x_2)-f(x_1)$. Esta persistencia es codificada a través del *diagrama de persistencia*, representando cada par con el punto $(f(x_1), f(x_2))$, como se puede ver en la figura 2.17. Se puede observar que todos los puntos se encontrarán por encima de la diagonal $y = x$, y que la persistencia es la distancia vertical de un punto a la diagonal. Por razones que explicaremos después se añadirán los puntos de la diagonal al diagrama de persistencia.

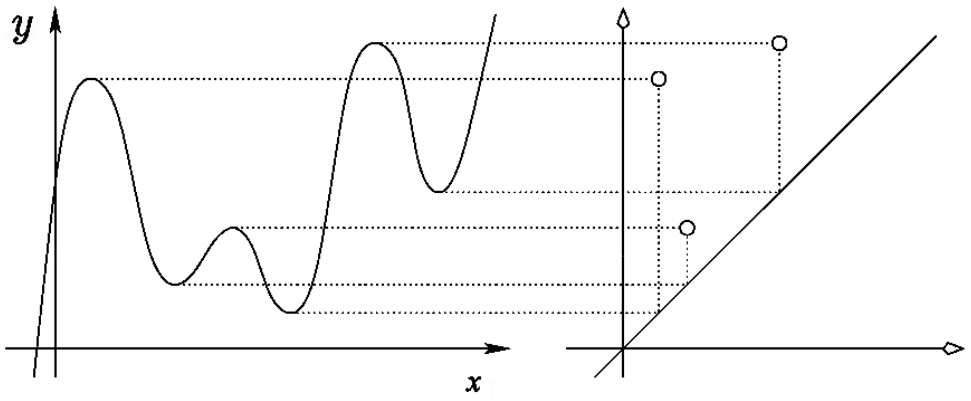


Figura 2.17: Emparejamiento de los puntos críticos de la función de la función de la izquierda representados como puntos en el diagrama de persistencia de la derecha. Fuente: [3]

Funciones Morse

Vamos a generalizar lo visto con funciones de una variable en $\mathbb{R}$ a funciones suaves sobre *variedades diferenciables* con ciertas propiedades que explicaremos más adelante. Primero recordaremos que son las variedades diferenciables.

Definición 2.1.35. Una *variedad diferenciable* un espacio topológico $\mathbb{M}$ que satisface:

A. $\mathbb{M}$ es Hausdorff (T_2).

B. $\mathbb{M}$ es segundo numerable, es decir, su topología tiene una base numerable.

C. Todo punto de $\mathbb{M}$ posee un entorno abierto difeomorfo a $\mathbb{R}^n$.

Sea $f : \mathbb{M} \to \mathbb{R}$ una aplicación suave. En este caso, un *punto crítico* es un punto $p \in \mathbb{M}$ tal que $\dfrac{\partial f}{\partial x_i}(p) = 0$ para $i = 1, ..., n$. Un punto crítico p es *no degenerado* si la matriz Hessiana de las segundas derivadas parciales,

$$(H_f)_{i,j} = \left(\frac{\partial^2 f}{\partial x_i \partial x_j} \right)_{i,j}$$

es no singular. Si p es un punto crítico no degenerado se define su *índice* como el número de autovalores negativos de la matriz Hessiana en p.

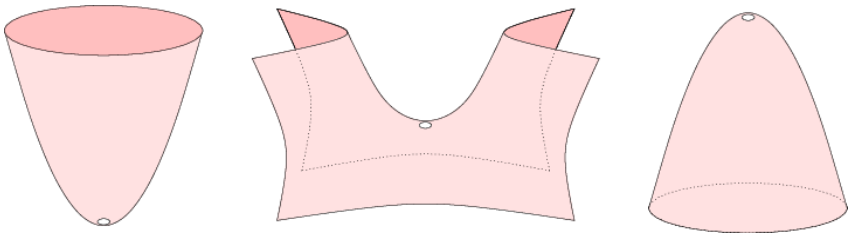


Figura 2.18: De izquierda a derecha tenemos: un punto crítico no degenerado de índice 0, 1 y 2. Fuente: [3]

Definición 2.1.36. Sea $f : \mathbb{M} \to \mathbb{R}$ una aplicación diferenciable. Diremos que f es una *función Morse* si todos sus puntos críticos son no degenerados y tienen distintos valores críticos.

Se puede demostrar que las funciones Morse poseen un número finito de puntos críticos. Elegimos los valores regulares $t_0 < t_1 < ... < t_m$ tal que existe un único punto crítico $p_i \in (t_i, t_{i+1})$ para todo $i = 0, ..., m-1$. Sea $\mathbb{M}_j = f^{-1}(-\infty, t_j]$ el *conjunto de subnivel* que contiene los primeros j puntos críticos.

Cuando pasamos de $\mathbb{M}_{j-1}$ a $\mathbb{M}_j$ la homología (singular) puede cambiar de dos formas distintas:

A) H_p incrementa la dimensión en uno, es decir, $\beta_p(\mathbb{M}_j) = \beta_p(\mathbb{M}_{j-1}) + 1$.

B) H_{p-1} disminuye la dimensión en uno, es decir, $\beta_{p-1}(\mathbb{M}_j) = \beta_{p-1}(\mathbb{M}_{j-1}) - 1$.

Donde p es el índice del j-ésimo punto crítico. En el primer caso denotaremos a ese punto crítico como *punto crítico positivo* y en el segundo como *punto crítico negativo*.

La persistencia nos dará un emparejamiento de algunos de los puntos críticos positivos de índice p con puntos críticos negativos de índice $p+1$. La idea es determinar el "momento" en el que nace una clase de homología y cuando muere, de forma que la persistencia será la diferencia de los tiempos. Para ello haremos uso de funciones entre grupos de homología inducidos por la inclusión de los conjuntos de subnivel $\mathbb{M}_i \subseteq \mathbb{M}_j$ para $i \leq j$. Definiremos de forma más precisa los conceptos de nacimiento y muerte de una clase de homología de la siguiente forma:

- Una clase de homología α *nace* en $\mathbb{M}_i$ si no existe en $\mathbb{M}_{i-1}$.
- Una clase de homología α nacida en $\mathbb{M}_i$ *morirá al entrar en* $\mathbb{M}_j$ si la imagen de la función inducida por $\mathbb{M}_{i-1} \subseteq \mathbb{M}_{j-1}$ no contiene a la imagen de α pero la imagen de la función inducida por $\mathbb{M}_{i-1} \subseteq \mathbb{M}_j$ si. Siguiendo lo que vimos en funciones de una variable, lo que ocurre es que al entrar en $\mathbb{M}_j$ se junta la clase α con una clase que ya existía en $\mathbb{M}_{i-1}$.

Si α nace en $\mathbb{M}_i$ y muere al entrar $\mathbb{M}_j$, entonces emparejaremos sus puntos críticos correspondientes, x e y, y diremos que su persistencia es $j - i$ ó $f(y) - f(x)$ según convenga. Esta persistencia es codificada a través de los *diagramas de persistencia*, $\mathrm{Dgm}_p(f)$, representando cada emparejamiento de un punto crítico positivo de índice p con un punto crítico negativo de índice $p+1$ añadiendo el punto $(f(x), f(y))$ al diagrama. Al igual que hicimos en el caso de funciones reales de una variable, añadiremos

los puntos de la diagonal en el diagrama de persistencia.

Funciones tame

Se puede comprobar que las funciones Morse sobre variedades diferenciables limitarán demasiado para algunas aplicaciones. Es por ello que consideraremos un tipo de función $f : \mathbb{X} \to \mathbb{R}$, donde f y $\mathbb{X}$ cumplen una serie de propiedades menos restrictivas. Empezaremos extendiendo la noción de punto crítico de la siguiente forma:

Definición 2.1.37. Sea $\mathbb{X}$ un espacio topológico, f una función real en $\mathbb{X}$ y $\mathbb{X}_t = f^{-1}(-\infty, t]$ el conjunto de subnivel definido para el valor t. Un *valor crítico de homología* de f es un número real a para el cual existe un entero k tal que para todo $\epsilon > 0$ lo suficientemente pequeño, el homomorfismo $\mathrm{H}_k(\mathbb{X}_{a-\epsilon}) \to \mathrm{H}_k(\mathbb{X}_{a+\epsilon})$ [1] inducido por la inclusión, $\mathbb{X}_{a-\epsilon} \subseteq \mathbb{X}_{a+\epsilon}$, no es un isomorfismo.

En otras palabras, los valores críticos de homología son los niveles en los cuales la homología de los conjuntos de subnivel cambia. Como ya hemos visto, en el caso de las funciones Morse, estos puntos críticos de homología coinciden con los valores críticos de la función.

Definición 2.1.38. Una función $f : \mathbb{X} \to \mathbb{R}$ es *tame* si los grupos de homología de cada conjunto de subnivel son finito-dimensionales y f posee un número finito de valores críticos de homología.

En particular, las funciones Morse sobre variedades compactas son funciones tame, ya que la compacidad y el carácter aislado de los puntos críticos garantizan que estas funciones posean un número finito de puntos críticos. Para simplificar la notación, para cada entero k fijo, escribimos $F_x = \mathrm{H}_k(f^{-1}(-\infty, x])$, y para $x < y$, denotamos como $f_x^y : F_x \to F_y$ la aplicación lineal inducida por la inclusión $\mathbb{X}_x \subseteq \mathbb{X}_y$. Una vez establecida la notación, probaremos el lema 2.1.10, que nos será de gran ayuda para la demostración del *teorema de estabilidad*.

Propiedad 2.1.2. La familia de aplicaciones $(F_x^y)_{x \leq y}$ satisface las siguientes propiedades:

- $f_x^x = \mathrm{id}_{F_x}$.
- $f_m^y \circ f_x^m = f_x^y$, con $x \leq m \leq y$.

Lema 2.1.10 (Lema del valor crítico)**.** *Si un intervalo cerrado $[x, y]$ no contiene ningún valor crítico de homología de f, entonces f_x^y es un isomorfismo para todo entero k.*

Demostración. Sea $m = (x + y)/2$, tenemos que $f_x^y = f_m^y \circ f_x^m$. Supongamos que f_x^y no es un isomorfismo. Entonces, al menos una de las funciones f_m^y y f_x^m no es un isomorfismo.
Repitiendo este argumento sobre las funciones no isomorfas de la composición obtenemos una sucesión de intervalos encajados cerrados y acotados, $I_n = [x_n, y_n]$, con

$$\lim_{x \to \infty} |y_n - x_n| = 0 \text{ y tal que } f_{x_n}^{y_n} \text{ no es un isomorfismo para todo } n \in \mathbb{N}$$

por lo que, aplicando el principio de intervalos encajados en $\mathbb{R}$, sabemos que su intersección es un punto $a \in \mathbb{R}$, que verifica que $f_{a-\epsilon}^{a+\epsilon}$ no es un isomorfismo para todo $\epsilon > 0$.

[1] En esta sección consideraremos la *homología singular* como teoría de homología, dado que los espacios topológicos $\mathbb{X}$ no requieren ser triangulables.

Luego, el punto a es un valor crítico de homología en $[x, y]$, contradiciendo nuestra hipótesis inicial. □

Definición 2.1.39. Sea $f_x^y : F_x \to F_y$ la aplicación lineal inducida por la inclusión $\mathbb{X}_x \subseteq \mathbb{X}_y$. Se definen los *grupos de homología persistente* como la imagen de F_x en F_y de la aplicación f_x^y, es decir,

$$F_x^y = \text{im } f_x^y \,.$$

Los correspondientes *números de Betti persistentes* se definen como los rangos de estos grupos, es decir, $\beta_x^y = \dim F_x^y$, para todo $-\infty \leq x \leq y \leq +\infty$.

Por convención, se establece que $F_x^y = \{0\}$ cuando x ó y son infinito. El grupo de homología persistente consiste de las clases que han nacido antes de x y siguen vivas en y.

Observación. Si analizamos las aplicaciones f_x^y, observamos que el ker f_x^y son aquellos elementos $\gamma \in F_x$ tales que $f_x^y(\gamma) = 0$. Esto significa que si c es un ciclo representando a γ, $c \in \mathrm{B}_k(\mathbb{X}_y)$. Como consecuencia

$$\ker f_x^y = \frac{\mathrm{Z}_k(\mathbb{X}_x) \cap \mathrm{B}_k(\mathbb{X}_y)}{\mathrm{B}_k(\mathbb{X}_x)}$$

para cada dimensión k fija.

Sea $f : \mathbb{X} \to \mathbb{R}$ una función tame, $(a_i)_{i=1..n}$ sus valores críticos homológicos y se considera la sucesión entrelazada $(b_i)_{i=0..n}$, tal que $b_{i-1} < a_i < b_i$ para $1 \leq i \leq n$. Para capturar la homología a lo largo de todo el proceso hacemos $b_{-1} = a_0 = -\infty$ y $b_{n+1} = a_{n+1} = +\infty$. Entonces,

Definición 2.1.40. Se define la multiplicidad del par (a_i, a_j) como

$$\mu_i^j = \beta_{b_{i-1}}^{b_j} - \beta_{b_i}^{b_j} + \beta_{b_i}^{b_{j-1}} - \beta_{b_{i-1}}^{b_{j-1}}, \text{ para todo } i, j \in \mathbb{Z} \text{ tal que } 0 \leq i < j \leq n+1 \,.$$

Podemos visualizar la multiplicidad, μ_i^j, como se muestra en la figura 2.19. Donde, considerando β_x^y como una función sobre el plano real extendido $\overline{\mathbb{R}}^2$, donde $\overline{\mathbb{R}} = \mathbb{R} \cup \{-\infty, +\infty\}$; entonces, μ_i^j es la suma alternada de los números de Betti persistentes en las esquinas del cuadrado $[b_{i-1}, b_i] \times [b_{j-1}, b_j]$.

Observación. Si x y x' se encuentran dentro del intervalo (a_i, a_{i+1}), e y e y' en el intervalo (a_{j-1}, a_j), entonces $\beta_x^y = \beta_{x'}^y$. Este resultado se sigue como consecuencia del *Lema del valor crítico*, que garantiza que F_x^y y $F_{x'}^{y'}$ son isomorfos.

Definición 2.1.41. El *diagrama de persistencia* $\mathrm{Dgm}(f) \subset \overline{\mathbb{R}}^2$ de f es el multiconjunto de puntos (a_i, a_j) con multiplicidad μ_i^j para todo $0 \leq i < j \leq n+1$, unión los puntos de la diagonal, $\Delta = \{(x, y) \in \overline{\mathbb{R}}^2 \mid y = x\}$, con multiplicidad infinito.

Denotaremos por $\#(A)$ la *multiplicidad total* de un multiconjunto A, que, por definición es la suma de las multiplicidades de los elementos de A. Por tanto, la multiplicidad total del diagrama de persistencia menos la diagonal es

$$\#(\mathrm{Dgm}(f) \setminus \Delta) = \sum_{i<j} \mu_i^j \,.$$

Esta multiplicidad se denomina *tamaño* del diagrama de persistencia.

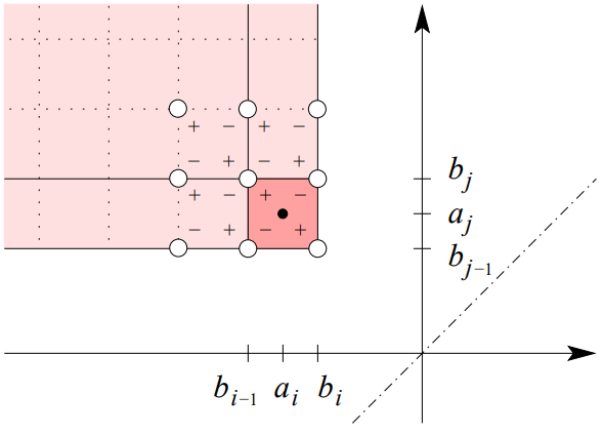


Figura 2.19: La multiplicidad del punto (a_i, a_j) es la suma alternada de los números de Betti persistentes en las esquinas del cuadrado $[b_{i-1}, b_i] \times [b_{j-1}, b_j]$. Fuente: [4]

Denotaremos el cuadrante superior izquierda cerrado con vértice en el punto (x, y) como $Q_x^y = [-\infty, x] \times [y, \infty]$.

Lema 2.1.11 (Lema del k-Triángulo)**.** *Sea f una función tame y $x < y$ diferentes de los valores críticos homológicos de f. Entonces, la multiplicidad total del diagrama de persistencia en el cuadrante superior izquierdo con vértice (x, y) es*

$$\#(\mathrm{Dgm}(f) \cap Q_x^y) = \beta_x^y \,.$$

Demostración. Podemos asumir sin pérdida de generalidad que $x = b_i$ y $y = b_{j-1}$. Por definición, la multiplicidad total en el cuadrante superior izquierdo es igual a la suma de las multiplicidades de los puntos contenidos en dicho cuadrante, luego

$$\#(\mathrm{Dgm}(f) \cap Q_x^y) = \sum_{k \leq i} \sum_{l > j} \mu_k^l = \sum_{k \leq i} \sum_{l > j} (\beta_{b_{k-1}}^{b_l} - \beta_{b_k}^{b_l} + \beta_{b_k}^{b_{l-1}} - \beta_{b_{k-1}}^{b_{l-1}}) \,.$$

Como se muestra en la figura 2.19, cuando se suman las multiplicidades, ocurre la cancelación entre signos positivos y negativos de las esquinas de los cuadrados. Entonces:

$$\begin{aligned} \#(\mathrm{Dgm}(f) \cap Q_x^y) &= \beta_{b_{-1}}^{b_{n+1}} - \beta_{b_i}^{b_{n+1}} + \beta_{b_i}^{b_{j-1}} - \beta_{b_{-1}}^{b_{j-1}} = \\ &= \beta_{-\infty}^{+\infty} - \beta_{b_i}^{+\infty} + \beta_{b_i}^{b_{j-1}} - \beta_{-\infty}^{b_{j-1}} = \beta_{b_i}^{b^{j-1}} = \beta_x^y \end{aligned}$$

ya que $F_x^y = \{0\}$ cuando x ó y son infinito, y por lo tanto su dimensión, es decir, su número de Betti persistente, es cero. □

Este lema nos garantiza que el diagrama de persistencia codifica toda la información sobre los grupos de homología persistente [2].

Persistencia en complejos simpliciales

Veremos que podemos particularizar la persistencia vista para funciones tame a complejos simpliciales. Para ello utilizaremos las *filtraciones* de un complejo simplicial como conjuntos de subnivel y haremos uso de la *homología simplicial* como teoría de homología.

Definición 2.1.42. Sea K un complejo simplicial y $f : K \to \mathbb{R}$ una función. Se dice que, f es *monótona* si $f(\sigma) \leq f(\tau)$ si σ es una cara de τ.

La monotonía de f garantiza que para cada $a \in \mathbb{R}$, el conjunto de subnivel $K(a) = f^{-1}(-\infty, a]$ es un subcomplejo de K.

Definición 2.1.43. Sean $a_1 < a_2 < ... < a_n$ los valores que toma la función en los símplices y sea $a_0 = -\infty$. Entonces f induce una *filtración*

$$\emptyset = K_0 \subseteq K_1 \subseteq ... \subseteq K_n = K, \text{ con } K_i = K(a_i)\,.$$

De esta forma, al igual que vimos con las funciones Morse, una clase de homología α nace en K_i si no está en la imagen de la función inducida por la inclusión $K_{i-1} \subseteq K_i$. Además, una clase α que nace en K_i muere al entrar en K_j si la imagen de la función inducida por $K_{i-1} \subseteq K_{j-1}$ no contiene la imagen de α, pero la imagen de la función inducida por $K_{i-1} \subseteq K_j$ sí.

Introduciremos los grupos de homología persistente, reduciendo la notación de la siguiente forma: $F_i = F_{b_i}$, $F_i^j = F_{b_i}^{b_j}$ y $\beta_i^j = \beta_{b_i}^{b_j}$. Así podemos redefinir la noción de nacimiento y muerte de una clase de homología como sigue

- Una clase $\gamma \in F_i$ *nace* en K_i si $\gamma \notin F_{i-1}^i$.
- Una clase $\gamma \in F_i$ nacida en K_i *muere* al entrar en K_j si $f_i^{j-1}(\gamma) \notin F_{i-1}^{j-1}$, pero $f_i^j(\gamma) \notin F_{i-1}^j$.

Definición 2.1.44. Sea γ una clase de homología que nace en K_i y muere al entrar en K_j. Se define la *persistencia* de γ como $\text{pers}(\gamma) = a_j - a_i$. Asimismo, la diferencia $j - i$ se denomina *índice de persistencia* de la clase γ. Si una clase γ nace en K_i pero nunca muere, entonces diremos que su persistencia, al igual que su índice, es infinito.

Siguiendo esta notación, se define la multiplicidad como

$$\mu_i^j = (\beta_i^{j-1} - \beta_i^j) - (\beta_{i-1} - \beta_{i-1}^i)\,.$$

Donde β_i^{j-1} se puede interpretar como el número de clases de homología que están vivas en K_i y siguen vivas en K_{j-1}. Por lo tanto, la primera diferencia de la igualdad se interpreta como el número de clases independientes que están vivas en K_i y mueren en K_j, mientras que la segunda diferencia son el número de clases independientes que nacen antes de K_i y mueren en K_j. En conclusión, la multiplicidad, μ_i^j, se interpreta como el número de clases de homología que nacen en K_i y mueren en K_j.

Cada punto (a_i, a_j) representa μ_i^j clases de homología independientes cuya persistencia coincide con la distancia del punto (a_i, a_j) a su proyección vertical sobre la diagonal Δ. Por razones técnicas, los puntos de la diagonal se añaden al diagrama de persistencia con multiplicidad infinito.

Adicionalmente de los diagramas de persistencia, podemos codificar la información sobre la homología persistente a través de los denominados *códigos de barras*. Estas representaciones se pueden obtener a partir del diagrama de persistencia dibujando por cada punto (a_i, a_j) con $a_i < a_j$ de dicho diagrama μ_i^j intervalos semiabiertos $[a_i, a_j)$, como se muestra en la figura 2.20.

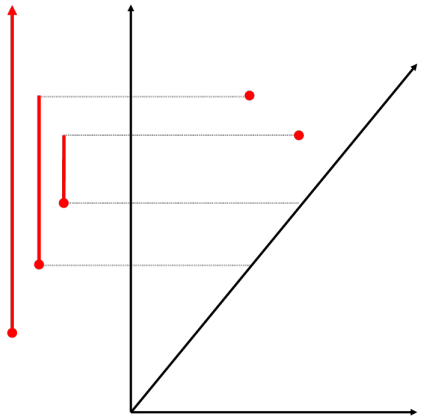


Figura 2.20: Código de barras asociado a un diagrama de persistencia. Fuente: [10]

Funciones PL

Un caso especial de las funciones tame son las *funciones lineales a trozos* (en inglés: *piecewise linear function*) que asocian valores reales al espacio subyacente de un complejo simplicial.

Definición 2.1.45. Sea K un complejo simplicial con valores reales asignados en todos sus vértices. Se define la función lineal a trozos $f : |K| \to \mathbb{R}$ como la extensión linear de los valores de los vértices sobre los símplices, es decir,

$$f(x) = \sum_i b_i(x) f(u_i)$$

donde u_i son los vértices de K y $b_i(x)$ son las coordenadas baricéntricas de x.

Por simplicidad se asume que $f|_{\text{Vert } K}$ es inyectiva. Reindexando los vértices de forma que $f(u_i) < f(u_2) < ... < f(u_n)$, definimos K_i como el subcomplejo definido por los primeros i vértices.

Definición 2.1.46. La *estrella inferior* de un vértice $u_i \in \text{Vert } K$ se define como el subconjunto de símplices para los cuales u_i es el vértice de mayor valor de f:

$$\text{St\_}u_i = \{\sigma \in \text{St } u_i \mid x \in \sigma \Rightarrow f(x) \leq f(u_i)\}.$$

Como ocurría con la estrella, la estrella inferior generalmente no es un subcomplejo. Añadiendo las caras restantes a los símplices en St_u_i, obtenemos la *estrella inferior cerrada* $\overline{\text{St}}\text{\_}u_i$, que es el menor subcomplejo de K que contiene a St_u_i. Como f es inyectiva en sus vértices, cada símplice tiene un único vértice con valor máximo, y por tanto pertenece a una única estrella inferior. Luego, K_i es la unión de las primeras i estrellas inferiores; obteniendo la siguiente filtración de K:

Definición 2.1.47. Sea K un complejo simplicial y $f_i : |K| \to \mathbb{R}$ una función PL. Se define la *filtración de K por las estrellas inferiores de f* como la filtración de subcomplejos $\emptyset = K_0 \subseteq K_1 \subseteq ... \subseteq K_n = K$, donde $K_i = K_{i-1} \cup \overline{\text{St}}\text{\_}u_i$.

Esta filtración cumple las siguientes propiedades:

Propiedad 2.1.3 ([2])**.** K_i tiene el mismo tipo de homotopía que el subnivel $|K|_t = f^{-1}(-\infty, t]$, para todo $f(u_i) \leq t < f(u_{i+1})$.

Propiedad 2.1.4 ([2])**.** La la variación de la homología en los conjuntos de subnivel $|K|_t = f^{-1}(-\infty, t]$ es la misma que la homología de la filtración por las estrellas inferiores de f.

Propiedad 2.1.5 ([3])**.** Sea $\mathbb{X}$ un espacio topológico triangulable. Entonces podemos aproximar toda función tame en $\mathbb{X}$ a partir de una función PL en su triangulación.

2.2. Teorema de estabilidad

En esta sección enunciaremos y demostraremos el *teorema de estabilidad de los diagramas de persistencia*, siguiendo [4]. Primero estudiaremos la estabilidad para la *distancia de Hausdorff* y, después, reforzaremos el resultado estudiando la estabilidad con la *distancia bottleneck*.

2.2.1. Proposición del teorema

El teorema de estabilidad nos va a garantizar la robustez de los diagramas de persistencia. Dicho de otro modo, que "pequeñas" perturbaciones en las funciones, dan lugar a diagramas de persistencia "cercanos". Así pues, primero precisaremos el concepto de cercanía entre funciones y diagramas de persistencia.

Sean X e Y dos diagramas de persistencia. Recordamos que X e Y son dos multiconjuntos de puntos del plano extendido $\overline{\mathbb{R}}^2$, constituidos por un número finito de puntos sobre la diagonal, y por los puntos de la diagonal con multiplicidad infinito.

Definición 2.2.1. Sean los puntos $p = (p_1, p_2)$ y $q = (q_1, q_2)$ en $\overline{\mathbb{R}}^2$. Entonces, la distancia infinito entre los puntos es:

$$d_\infty(p, q) = \|p - q\|_\infty = \max\{|p_1 - q_1|, |p_2 - q_2|\}.$$

Definición 2.2.2. Sean $f, g : \mathbb{X} \to \mathbb{R}$ dos funciones continuas. Entonces, la distancia infinito entre las funciones es:

$$d_\infty(f, g) = \|f - g\|_\infty = \sup_{x \in \mathbb{X}} |f(x) - g(x)|.$$

Definiremos las distancias Hausdorff y bottleneck sobre multiconjuntos (diagramas de persistencia en nuestro caso) de la siguiente forma

Definición 2.2.3. La *distancia Hausdorff* y la *distancia bottleneck* entre X e Y son, respectivamente

$$H(X, Y) = \max\left\{\sup_{x \in X} \inf_{y \in Y} \|x - y\|_\infty, \sup_{y \in Y} \inf_{x \in X} \|y - x\|_\infty\right\},$$
$$W_\infty(X, Y) = \inf_{\eta: X \to Y} \sup_{x \in X} \|x - \eta(x)\|_\infty$$

siendo $\eta : X \to Y$ las biyecciones de X a Y.

Las biyecciones entre dos diagramas de persistencia generan tres tipos de emparejamientos:

- Ambos puntos fuera de la diagonal.

- Un punto fuera de la diagonal y otro en la diagonal.
- Ambos puntos en la diagonal.

Se puede observar que los puntos que determinar en mayor escala la distancia bottelneck son los del primer tipo, y los que menor importancia tienen son los del último tipo, ya que completarán el emparejamiento sin afectar en el cálculo de las distancias.

Observación. Debido a que la distancia bottleneck satisface una restricción adicional respecto a la distancia Hausdorff, es decir, la biyección entre los puntos; entonces, se cumple $H(X,Y) \leq W_\infty(X,Y)$.

Teorema 2.2.1 (Teorema de estabilidad para funciones tame)**.** *Sea $\mathbb{X}$ un espacio topológico triangulable y sea $f, g : \mathbb{X} \to \mathbb{R}$ dos funciones tame continuas. Entonces, para cada dimensión k, la distancia bottleneck entre los diagramas de persistencia esta acotada por la distancia infinito entre las funciones, es decir,*

$$W_\infty(\mathrm{Dgm}(f), \mathrm{Dgm}(g)) \leq \|f - g\|_\infty .$$

Luego, se garantiza que los diagramas de persistencia son estables bajo perturbaciones de baja amplitud. Este resultado se puede observar gráficamente en la figura 2.21, donde se observa que los valores críticos "superfluos" de la función perturbada definen puntos en el diagrama muy próximos a la diagonal, y los valores críticos "relevantes" definen puntos muy próximos a los puntos del diagrama asociados a la función original.

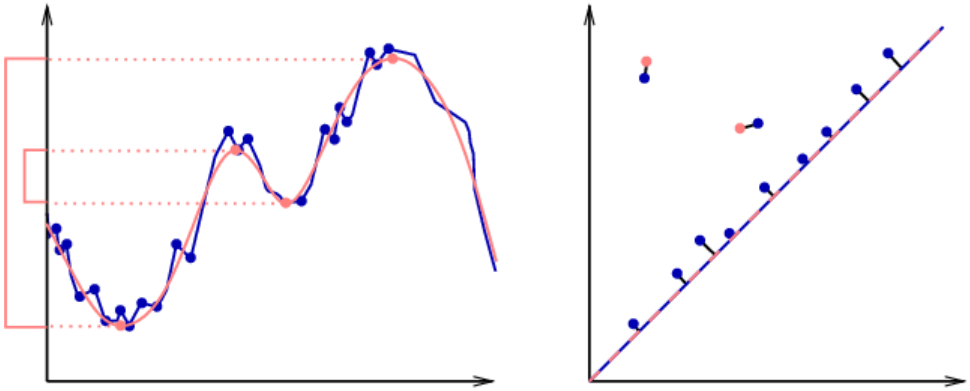


Figura 2.21: A la izquierda se muestran dos funciones cercanas, una con muchos valores críticos y otra con cuatro. A la derecha se muestran los diagramas de persistencia superpuestos, con la biyección que da lugar a la distancia bottleneck. Fuente: [4]

2.2.2. Estabilidad para la distancia Hausdorff

Partiremos de la demostración de la estabilidad con la distancia Hausdorff, que sigue sigue así

Teorema 2.2.2 (Teorema de estabilidad con la distancia Hausdorff para funciones tame)**.** *Sea $\mathbb{X}$ un espacio topológico triangulable y sea $f, g : \mathbb{X} \to \mathbb{R}$ dos funciones tame continuas. Entonces, para cada dimensión k, la distancia Hausdorff entre los diagramas de persistencia esta acotada por la distancia L_∞ entre las funciones, es decir,*

$$H(\mathrm{Dgm}(f), \mathrm{Dgm}(g)) \leq \|f - g\|_\infty .$$

Relaciones entre cuadrantes superiores izquierdos

Primero estudiaremos la relación entre las multiplicidades de los cuadrantes superiores izquierdos de dos diagramas de persistencia.

Proposición 2.2.1. *Sean $f, g : \mathbb{X} \to \mathbb{R}$ dos funciones tame continuas. Si denotamos $\epsilon = \|f - g\|_\infty$, entonces $f^{-1}(-\infty, x] \subseteq g^{-1}(-\infty, x + \epsilon]$ para todo $x \in \mathbb{R}$*

Demostración. Sea $y \in \mathbb{X}$ tal que $y \in f^{-1}(-\infty, x] = \{x \in \mathbb{X} \mid f(x) \in (-\infty, x]\}$. Como $\|f-g\|_\infty = \sup_{x\in\mathbb{X}} |f(x)-g(x)| = \epsilon$, entonces $|f(y)-g(y)| < \epsilon$ por lo que $g(y) \in (-\infty, x+\epsilon]$, de donde se sique que $y \in g^{-1}(-\infty, x + \epsilon]$. □

Denotamos por $\varphi_x : F_x \to G_{x-\epsilon}$ a la aplicación inducida por esta inclusión. La inclusión análoga, $g^{-1}(-\infty, x] \subseteq f^{-1}(-\infty, x + \epsilon]$, induce la aplicación $\psi_x : G_x \to F_{x+\epsilon}$. Sea $b < c$, estas dos aplicaciones dan lugar a los siguientes diagramas conmutativos:

$$\begin{array}{ccc} F_{b-\epsilon} & \xrightarrow{f_{b-\epsilon}^{c+\epsilon}} & F_{c+\epsilon} \\ {\scriptstyle \varphi_{b-\epsilon}}\downarrow & & \uparrow{\scriptstyle \psi_c} \\ G_b & \xrightarrow{g_b^c} & G_c \end{array} \qquad \begin{array}{ccc} F_{b+\epsilon} & \xrightarrow{f_{b+\epsilon}^{c+\epsilon}} & F_{c+\epsilon} \\ {\scriptstyle \psi_b}\uparrow & & \uparrow{\scriptstyle \psi_c} \\ G_b & \xrightarrow{g_b^c} & G_b^c \end{array}$$

Los diagramas son conmutativos, dado que están inducidos por los diagramas de inclusiones que son conmutativos.

Del primer diagrama tenemos que $f_{b-\epsilon}^{c+\epsilon} = \psi_c \circ g_b^c \circ \varphi_{b-\epsilon}$. Sea $\xi \in F_{b-\epsilon}^{c+\epsilon} = \text{im } f_{b-\epsilon}^{c+\epsilon}$, de forma que $\xi = f_{b-\epsilon}^{c+\epsilon}(\eta)$ para un $\eta \in F_{b-\epsilon}$. Luego, $\xi = \psi_c(\zeta)$, con $\zeta = g_b^c(\varphi_{b-\epsilon}(\eta)) \in G_b^c$, por tanto $F_{b-\epsilon} \subseteq \psi_c(G_b^c)$.

Del segundo diagrama, tenemos que $\psi_c(G_b^c) = \psi_c \circ g_b^c(G_b)$, ya que $G_b^c = \text{im } g_b^c = g_b^c(G_b)$. A su vez, se cumple que $\psi_c \circ g_b^c(G_b) = f_{b+\epsilon}^{c+\epsilon} \circ \psi_b(G_b) \subseteq F_{b+\epsilon}^{c+\epsilon}$ de donde se sigue que $\psi_c(G_b^c) \subseteq F_{b+\epsilon}^{c+\epsilon}$.

Así pués, se cumple:

$$F_{b-\epsilon}^{c+\epsilon} \subseteq \psi_c(G_b^c) \subseteq F_{b+\epsilon}^{c+\epsilon}\,. \tag{2.1}$$

De manera análoga podemos demostrar que se cumple que

$$G_{b-\epsilon}^{c+\epsilon} \subseteq \varphi_c(F_b^c) \subseteq G_{b+\epsilon}^{c+\epsilon}$$

intercambiando F_x y G_y en los diagramas y sustituyendo las aplicaciones inducidas correctamente.

Recordamos que por la *Fórmula de las dimensiones*[11], si una aplicación $f : U \to V$ es lineal entonces se cumple que

$$\dim \ker f + \dim \text{im } f = \dim U\,. \tag{2.2}$$

De la primera inclusión de 2.1 obtenemos que $\dim F_{b-\epsilon}^{c+\epsilon} \leq \dim \psi_c(G_b^c) \overset{(2.2)}{\leq} \dim G_b^c$. Aplicando el *Lema del k-Triángulo* a la anterior desigualdad y denotando a los cuadrantes superiores izquierdos como $Q = Q_b^c$ y $Q_\epsilon = Q_{b-\epsilon}^{c+\epsilon}$, se obtiene el siguiente resultado:

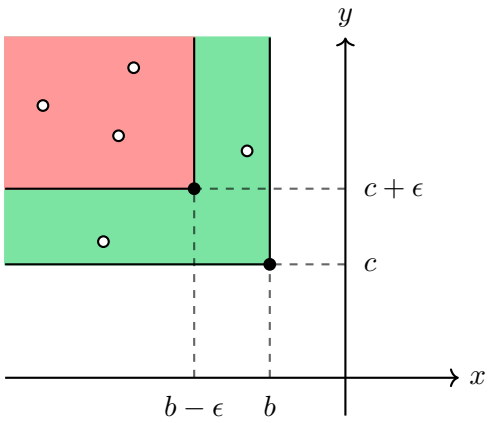


Figura 2.22: Representación del *Lema del cuadrante*

Lema 2.2.3 (Lema del cuadrante)**.** $\#(\mathrm{Dgm}(f) \cap Q_\epsilon) \leq \#(\mathrm{Dgm}(g) \cap Q)$.

Demostración. Si b y c no son valores críticos homológicos de g y $b-\epsilon$, $c+\epsilon$ no son valores críticos homológicos de f, entonces por el Lema del k-Triángulo

$$\#(\mathrm{Dgm}(g) \cap Q) = \beta_b^c = \dim G_b^c \text{ y } \#(\mathrm{Dgm}(f) \cap Q_\epsilon) = \beta_{b-\epsilon}^{c+\epsilon} = \dim F_{b-\epsilon}^{c+\epsilon}.$$

Y como se tiene $\dim F_{b-\epsilon}^{c+\epsilon} \leq \dim G_b^c$, entonces $\#(\mathrm{Dgm}(f) \cap Q_\epsilon) \leq \#(\mathrm{Dgm}(g) \cap Q)$.

En el caso que los puntos b y c sean valores críticos homológicos de g y $b-\epsilon$, $c+\epsilon$ valores críticos homológicos de f, entonces podemos engordar los cuadrantes una cantidad $0 < \delta < \epsilon$, tal que

$$\#(\mathrm{Dgm}(f) \cap Q_\epsilon) = \#(\mathrm{Dgm}(f) \cap Q_{b-\epsilon+\delta}^{c+\epsilon-\delta}) \text{ y } \#(\mathrm{Dgm}(g) \cap Q) = \#(\mathrm{Dgm}(g) \cap Q_{b+\delta}^{c-\delta}),$$

siendo estas nuevas coordenadas distintas de los valores críticos de f y g respectivamente. □

Este lema nos garantiza que la multiplicidad total de $\mathrm{Dgm}(g)$ en el cuadrante superior izquierda con vértice en el punto (b, c) esta acotada inferiormente por la multiplicidad total de $\mathrm{Dgm}(f)$ en el cuadrante superior izquerda reducida por ϵ. Esto se puede observar en la figura 2.22.

Regiones como subespacios vectoriales

Sin embargo, el *Lema del cuadrante* no es lo suficientemente fuerte para nuestros propósitos. Vamos a obtener un resultado similar al *Lema del cuadrante*, pero en este caso para cajas encajadas. Esto se debe a que si se cumple $H(\mathrm{Dgm}(f), \mathrm{Dgm}(g)) \leq \|f-g\|_\infty = \epsilon$, entonces para todo punto $(x, y) \in \mathrm{Dgm}(f)$ debe haber un punto en $\mathrm{Dgm}(g)$ a distancia menor o igual que ϵ. Lo que significa que debe haber un punto $q \in \mathrm{Dgm}(g)$ dentro del cuadrado $[x-\epsilon, x+\epsilon] \times [y-\epsilon, y+\epsilon]$ [12].

Para definir estas regiones introduciremos subespacios vectoriales de $\overline{\mathbb{R}}^2$ y haremos uso del *Lema del k-triángulo* para poder expresar sus dimensiones como la multiplicidad total del diagrama de persistencia en dichas regiones.

Sean $w < x < y < z \in \mathbb{R}$ puntos diferentes a los valores críticos homológicos de $f : \mathbb{X} \to \mathbb{R}$. Recordamos que la dimensión del grupo de homología F_x es igual a la multiplicidad total en el cuadrante superior izquierdo de vértice el punto de la diagonal (x, x), y la dimensión del grupo de persistencia F_x^y es igual a la multiplicidad total en el cuadrante superior izquierdo de vértice el punto (x, y). Estas regiones se pueden observar en las figuras 2.23 (a),(b).

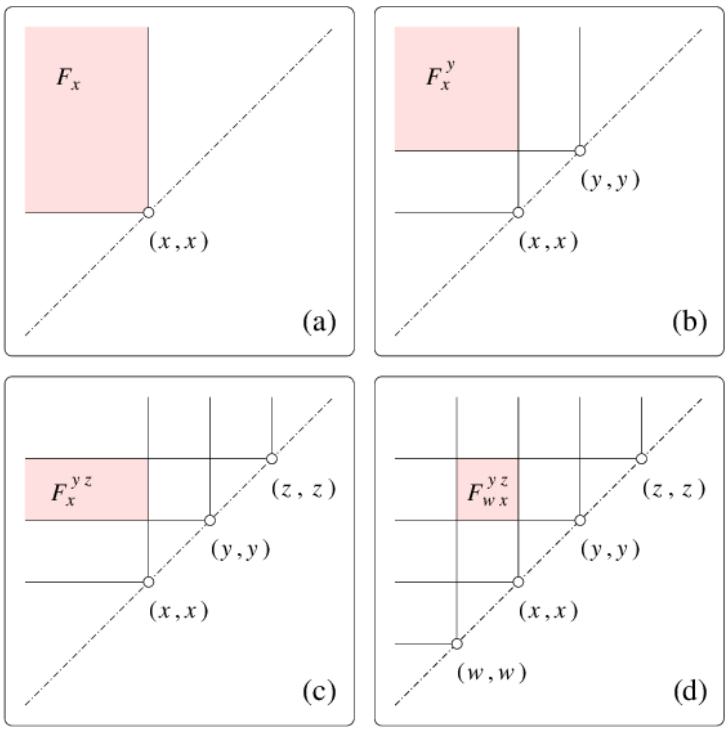


Figura 2.23: (a) Grupo de homología del conjunto de subnivel $f^{-1}(-\infty, x]$. (b) Imagen de F_x en F_y. (c) Núcleo de la sobreyección $F_x^y \to F_x^z$. (d) Cociente de $F_x^{y\,z}$ y $F_w^{y\,z}$. Fuente: [4]

Si restringimos $f_x^y : F_y \to F_z$ al espacio vectorial F_x^y tenemos la epimorfismo $f_x^{y\,z} : F_x^y \to F_x^z$, ya que todas las clases de homología que están vivas en x y siguen vivas en z, deben de seguir vivas en $y < z$. Denotando por $F_x^{y\,z}$ al núcleo de dicha aplicación, tenemos que $\dim F_x^{y\,z} = \dim F_x^y - \dim F_x^z$. Lo que corresponde con la sección marcada en la figura 2.23 (c), que contiene las clases que nacen antes de x y mueren entre y y z.

Además, podemos observar que se cumple $F_w^y \subseteq F_x^y$, ya que todo elemento de F_w^y, que es la imagen de un $\xi \in F_w$ por la aplicación f_w^y, es también la imagen de $f_w^x(\xi)$ por la aplicación f_x^y. Como consecuencia, $F_w^{y\,z} \subseteq F_x^{y\,z}$, y por tanto podemos definir el siguiente cociente

$$F_{w\,x}^{y\;\;z} = \frac{F_x^{y\,z}}{F_w^{y\,z}}\,.$$

Al ser un cociente de subespacios vectoriales, su dimensión es la diferencia de los dos núcleos, es decir, $\dim F_{w\,x}^{y\;\;z} = \dim F_x^{y\,z} - \dim F_w^{y\,z}$, que equivale a la multiplicidad total en del diagrama de persistencia en la caja $[w, x] \times [y, z]$; como se puede observar en la figura 2.23 (d). Por tanto, este rectángulo contiene las clases de equivalencia que nacen entre w y x y mueren entre y y z.

Relaciones entre cajas encajadas

Lema 2.2.4 (Lema de la caja)**.** *Sean $a < b < c < d \in \mathbb{R}$, $R = [a,b] \times [c,d]$ una caja en $\overline{\mathbb{R}}^2$ y $R_\epsilon = [a+\epsilon, b-\epsilon] \times [c+\epsilon, d-\epsilon]$ la caja obtenida de reducir R en todos sus lados. Entonces se cumple*

$$\#(\mathrm{Dgm}(f) \cap R_\epsilon) \leq \#(\mathrm{Dgm}(g) \cap R)\,.$$

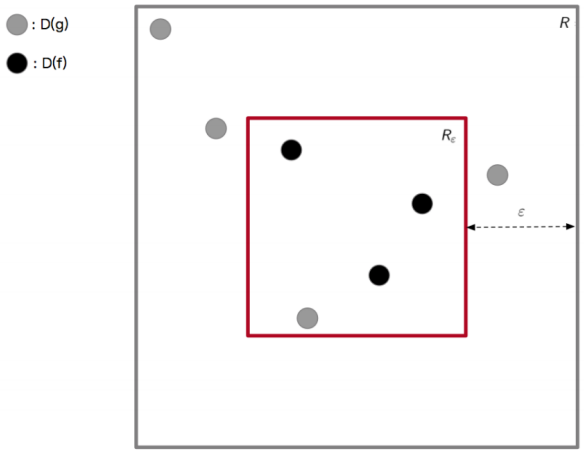


Figura 2.24: Representación del *Lema de la caja*. Fuente: [12]

Para poder demostrar el *Lema de la caja* primero recordemos el *Segundo teorema de isomorfía*:

Teorema 2.2.5 (Segundo teorema de isomorfía [13, Theorem 14.3])**.** *Sea V un espacio vectorial y sean S y T dos subespacios de V, entonces*

$$S/(S \cap T) \cong (S+T)/T\,.$$

Demostración del Lema 2.2.4 (Lema de la caja). Podemos asumir sin perder generalidad que a, b, c y d no son valores críticos homológicos de g y $a+\epsilon$, $b-\epsilon$, $c+\epsilon$ y $d-\epsilon$ no son valores críticos homológicos de f. Además consideraremos que $a+\epsilon < b-\epsilon$ y $c+\epsilon < d-\epsilon$, de forma que R_ϵ este bien definido.

Para el cálculo de las multiplicidades totales dentro de las cajas haremos uso de las dimensiones de los subespacios vectoriales asociados, es decir,

$$\dim F_{a+\epsilon\, b-\epsilon}^{c+\epsilon\, d-\epsilon} = \#(\mathrm{Dgm}(f) \cap R_\epsilon)\,, \tag{2.3}$$

$$\dim G_{a\, b}^{c\, d} = \#(\mathrm{Dgm}(g) \cap R)\,. \tag{2.4}$$

Para demostrar que $\dim F_{a+\epsilon\, b-\epsilon}^{c+\epsilon\, d-\epsilon} \leq \dim G_{a\, b}^{c\, d}$, buscaremos un epimorfismo entre un subespacio vectorial de $G_{a\, b}^{c\, d}$ y $F_{a+\epsilon\, b-\epsilon}^{c+\epsilon\, d-\epsilon}$. Para la construcción de dicha aplicación haremos uso del diagrama que se muestra en la figura 2.25, que, como veremos, esta bien definido y es conmutativo.

Definimos E_a^c como la preimagen, por la restricción de ψ_c a G_b^c, del núcleo de u_3 (ver figura 2.25), es decir, $E_b^c = \psi_c^{-1}(F_{b-\epsilon}^{c+\epsilon\, d-\epsilon}) \cap G_b^c$. Por (2.1) se cumple que $F_{b-\epsilon}^{c+\epsilon} \subseteq \psi_c(G_b^c)$, por lo que $s_3 = \psi_c|_{E_c^b}$ tiene al núcleo de u_3, $F_{b-\epsilon}^{c+\epsilon\, d-\epsilon}$, como su imagen.

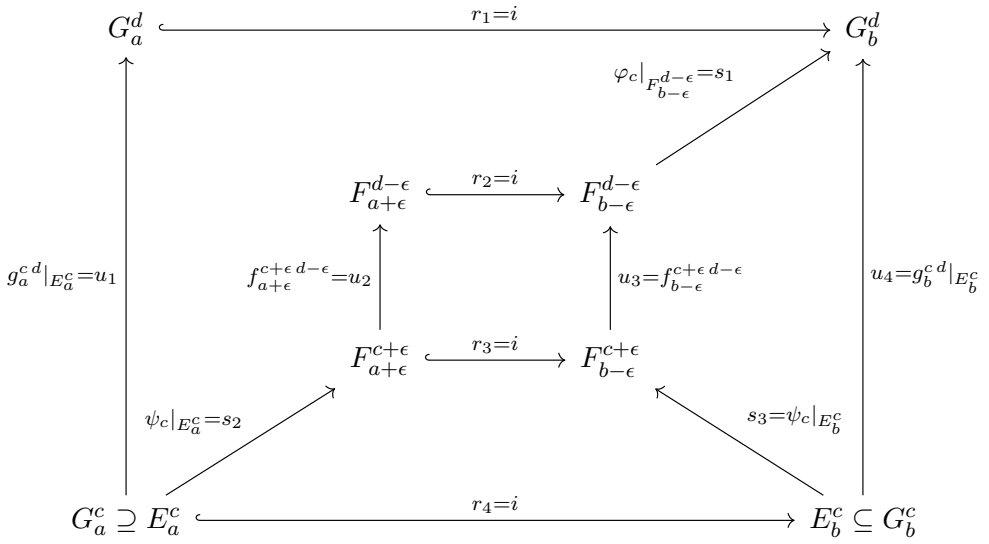


Figura 2.25: Diagrama conmutativo con la notación reducida explicada.

También definimos $E_a^c = G_a^c \cap E_b^c$. Veremos posteriormente que E_b^c/E_a^c es el subespacio de $G_{a\,b}^{c\,d}$ del que podremos encontrar un epimorfismo a $F_{a+\epsilon\,b-\epsilon}^{c+\epsilon\,d-\epsilon}$.

Continuando la descripción del diagrama conmutativo tenemos las aplicaciones r_1, r_2, r_3 y r_4 que son las inclusiones entre los respectivos espacios vectoriales. Además, u_1 es la restricción de $g_a^{c\,d}$ en E_a^c y u_2 es la restricción de $g_b^{c\,d}$ en E_b^c. También tenemos $s_2 = \psi_c|_{E_a^c}$ y por (2.1) se cumple que $\psi_c(G_a^c) \subseteq F_{a+\epsilon}^{c+\epsilon}$, lo que garantiza que s_2 esta bien definido ya que su imagen esta contenida en $F_{a+\epsilon}^{c+\epsilon}$. Finalmente, $s_1 = \varphi_c|_{F_{b-\epsilon}^{d-\epsilon}}$ y por (2.1) con F y G intercambiados se cumple que $\varphi_{d-\epsilon}(F_{b-\epsilon^{d-\epsilon}}) \subseteq G_b^d$, lo que garantiza que s_1 esta bien definido.

Por tanto, el diagrama esta bien definido y es conmutativo (ya que las aplicaciones son inclusiones o bien aplicaciones inducidas por inclusiones).

Como se puede observar en la figura 2.26a, $u_4 = s_1 \circ u_3 \circ s_3$, lo que implica que $E_b^c = \ker u_4$, ya que $u_3 \circ s_3$ es cero. Además, como se puede observar en la figura 2.26b, $r_1 \circ u_1 = u_4 \circ r_4$, lo que implica que $E_a^c = \ker u_1$, ya que $u_4 \circ r_4$ es cero y r_1 es inyectivo al ser una inclusión. Expresamos estas relaciones denotando $E_b^c = E_b^{c\,d} \subseteq G_b^{c\,d}$ y $E_a^c = E_a^{c\,d} \subseteq G_a^{c\,d}$.

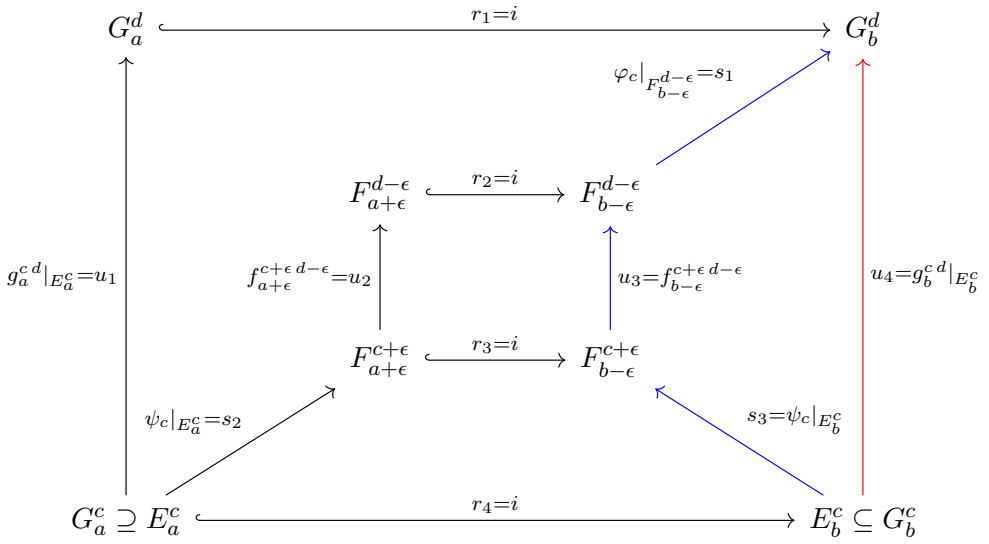
Como $E_a^{c\,d} = E_b^{c\,d} \cap G_a^{c\,d}$, el cociente

$$E_{a\,b}^{c\,d} = \frac{E_b^{c\,d}}{E_a^{c\,d}} = \frac{E_b^{c\,d}}{E_b^{c\,d} \cap G_a^{c\,d}} \overset{Th.\ 2.2.5}{\cong} \frac{E_b^{c\,d} + G_a^{c\,d}}{G_a^{c\,d}},$$

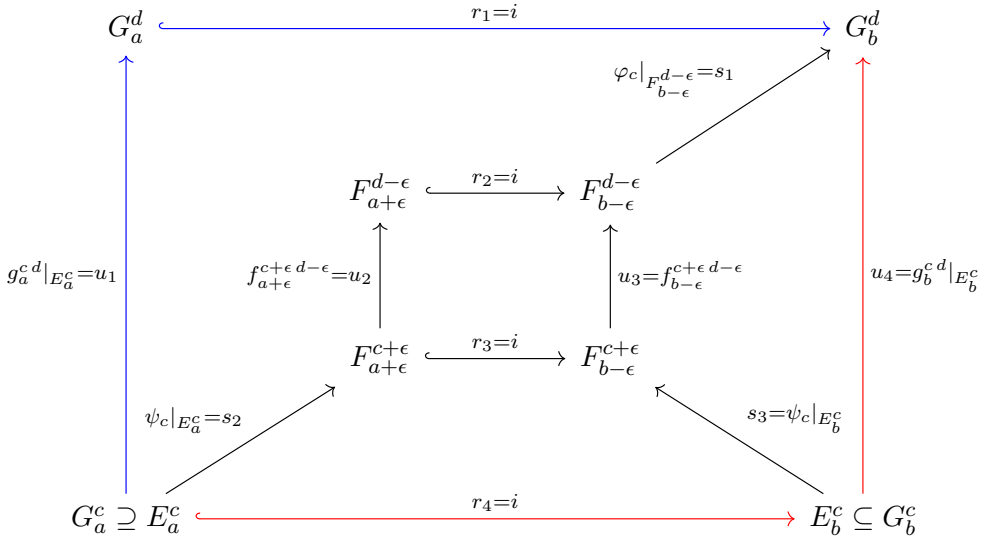
es decir, es un conjunto de clases laterales de elementos en $E_b^{c\,d} \subseteq G_b^{c\,d}$ módulo $G_a^{c\,d}$, por tanto $E_{a\,b}^{c\,d} \subseteq G_{a\,b}^{c\,d}$. Luego,

$$\dim E_{a\,b}^{c\,d} \leq \dim G_{a\,b}^{c\,d}. \tag{2.5}$$

Recordemos que $E_{a\,b}^{c\,d} = \ker u_4/\ker u_1$ y que $F_{a+\epsilon\,b-\epsilon}^{c+\epsilon\,d-\epsilon} = \ker u_3/\ker u_2$. Además, hemos observado que $s_3(\ker u_4) = s_3(E_b^c) = \ker u_3$. Así pues, para demostrar que s_3 induce un epimorfismo entre los cocientes $E_{a\,b}^{c\,d}$ y $F_{a+\epsilon\,b-\epsilon}^{c+\epsilon\,d-\epsilon}$ sólo quedaría por garantizar que $s_3(\ker u_1) = s_2(\ker u_1)$ esta incluida en $\ker u_2$. Sin embargo, esto se cumple, ya que



(a) El camino en azul representa $s_1 \circ u_3 \circ s_3$ y el camino en rojo u_4.



(b) El camino en azul representa $r_1 \circ u_1$ y el camino en rojo $u_4 \circ r_4$.

Figura 2.26: Representación de las composiciones como caminos en el diagrama conmutativo.

como se puede observar en la figura 2.27, $u_3 \circ s_3 \circ r_4(\xi) = r_2 \circ u_2 \circ s_2(\xi) = 0$ para todo $\xi \in \ker u_1$, y r_2 es inyectiva al ser una inclusión.

Como consecuencia, aplicando la fórmula de las dimensiones tenemos que $\dim E_{a\,b}^{c\,d} = \dim F_{a+\epsilon\,b-\epsilon}^{c+\epsilon\,d-\epsilon} + \dim \ker s3$, entonces

$$\dim F_{a+\epsilon\,b-\epsilon}^{c+\epsilon\,d-\epsilon} \leq \dim E_{a\,b}^{c\,d}\,. \tag{2.6}$$

Finalmente, obtenemos la desigualdad al concatenar (2.3), (2.6), (2.5) y (2.4), en este orden, es decir,

$$\#(\mathrm{Dgm}(f) \cap R_\epsilon) = \dim F_{a+\epsilon\,b-\epsilon}^{c+\epsilon\,d-\epsilon} \leq \dim E_{a\,b}^{c\,d} \leq \dim G_{a\,b}^{c\,d} = \#(\mathrm{Dgm}(g) \cap R)\,.$$

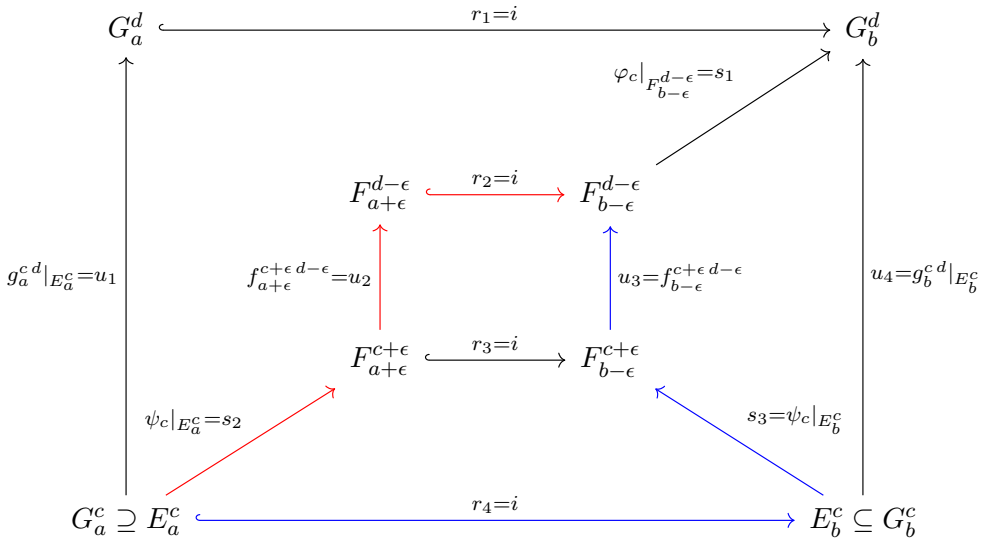


Figura 2.27: El camino en azul representa $u_3 \circ s_3 \circ r_4$ y el camino en rojo $r_2 \circ u_2 \circ s_2$.

□

Como comentábamos previamente, una consecuencia inmediata del *Lema de la caja* es que la distancia Hausdorff entre $\mathrm{Dgm}(f)$ y $\mathrm{Dgm}(g)$ no es mayor que ϵ. Ya que si $R_\epsilon = [x,x] \times [y,y] = (x,y)$ es un punto de $\mathrm{Dgm}(f)$, entonces debe haber un punto de $\mathrm{Dgm}(g)$ a distancia menor o igual que ϵ, porque la multiplicidad total de $\mathrm{Dgm}(g)$ en la caja $R = [x-\epsilon, x+\epsilon] \times [y-\epsilon, y+\epsilon]$ es mayor o igual que uno.

2.2.3. Estabilidad para la distancia bottleneck

Una vez demostrada que la distancia Hausdorff entre dos diagramas de persistencia esta acotada por las distancia infinito de las funciones tame, vamos a probar el resultado para la distancia bottleneck.

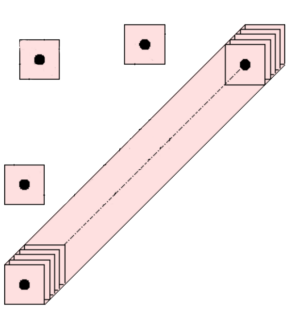
Estabilidad para la distancia bottleneck en un caso sencillo

Empezaremos demostrando la estabilidad para un caso concreto que tiene una demostración sencilla. Dada una función tame $f : \mathbb{X} \to \mathbb{R}$, consideramos la mínima distancia entre dos puntos fuera de la diagonal o bien entre un punto fuera de la diagonal y otro en la diagonal:

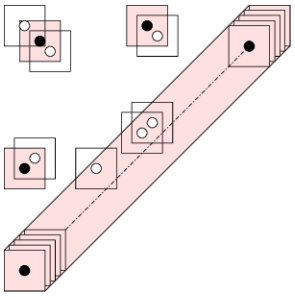
$$\delta_f = \min\{\|p-q\|_\infty \mid (\mathrm{Dgm}(f) \setminus \Delta) \ni p \neq q \in \mathrm{Dgm}(f)\}\,.$$

Si dibujamos cuadrados de radio $\epsilon = \delta_f/2$ centrados en los puntos de $\mathrm{Dgm}(f)$, obtenemos una colección de cuadrados disjuntos entre ellos y disjuntos de la diagonal engordada; ver figura 2.28a. Añadiremos otra función tame $g : \mathbb{X} \to \mathbb{R}$ que *es muy cercana a* f; ver figura 2.28b. Lo que significa que f y g satisfacen que $\|f-g\|_\infty < \delta_f/2$.

Así pues, probaremos el teorema de estabilidad para la distancia bottleneck añadiendo la condición que las funciones tame tienen que *estar muy cerca*.



(a) Cuadrados de radio ϵ centrados en los puntos de $\mathrm{Dgm}(f)$. Fuente: [12]



(b) Cuadrados de radio ϵ centrados en los puntos de $\mathrm{Dgm}(f)$ y $\mathrm{Dgm}(g)$, con g muy cercana a f. Fuente: [4]

Figura 2.28

Lema 2.2.6 (Lema de la biyección sencilla)**.** *Sean $f, g : \mathbb{X} \to \mathbb{R}$ dos funciones tame, y supongamos que g es muy cercana a f. Entonces los diagramas de persistencia satisfacen*

$$W_\infty(\mathrm{Dgm}(f), \mathrm{Dgm}(g)) \leq \|f - g\|_\infty .$$

Demostración. Denotamos μ a la multiplicidad del punto $p \in (\mathrm{Dgm}(f) \setminus \Delta)$ y $\square_\epsilon$ al cuadrado de centro p y radio $\epsilon = \|f - g\|_\infty$. Entonces, aplicando el *Lema de la caja* obtenemos que

$$\mu = \#(\mathrm{Dgm}(f) \cap \square_0) \leq \#(\mathrm{Dgm}(g) \cap \square_\epsilon) \leq \#(\mathrm{Dgm}(f) \cap \square_{2\epsilon}) .$$

Como g *es muy cercana a* f entonces $\epsilon = \|f - g\|_\infty < \delta_f/2$ por lo que $2\epsilon < \delta_f$. Como consecuencia p es el único punto de $\mathrm{Dgm}(f) \cap \square_{2\epsilon}$ como se puede ver en la figura 2.29. Por tanto

$$\#(\mathrm{Dgm}(f) \cap \square_{2\epsilon}) = \mu \Rightarrow \mu \leq \#(\mathrm{Dgm}(g) \cap \square_\epsilon) \leq \mu \Rightarrow \#(\mathrm{Dgm}(g) \cap \square_\epsilon) = \mu .$$

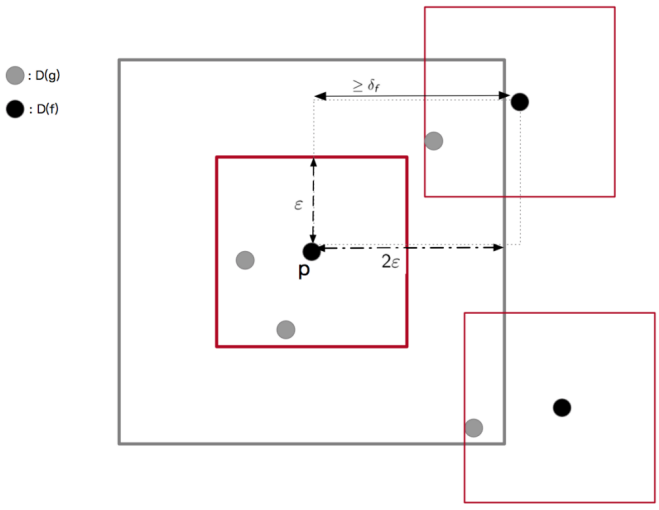


Figura 2.29: Se observa que al ser g *muy cercana a* f, entonces p es el único punto de $\mathrm{Dgm}(f) \cap \square_{2\epsilon}$. Fuente: [12]

Entonces, podemos emparejar todos los puntos de $\mathrm{Dgm}(g) \cap \square_\epsilon$ con p. Repitiendo este proceso para todos los puntos de $\mathrm{Dgm}(f)$ fuera de la diagonal, emparejaremos todos los puntos de $\mathrm{Dgm}(g)$ excepto aquellos que su distancia a $\mathrm{Dgm}(f) \setminus \Delta$ sea mayor que ϵ. Sin embargo, debido a que $H(\mathrm{Dgm}(f), \mathrm{Dgm}(g)) \leq \epsilon$, estos puntos de $\mathrm{Dgm}(g)$ deben estar a distancia menor o igual que ϵ de la diagonal. Por lo que emparejando estos puntos restantes a su proyección ortogonal sobre la diagonal obtenemos una biyección entre $\mathrm{Dgm}(f)$ y $\mathrm{Dgm}(g)$, tal y como se muestra en la figura 2.30.

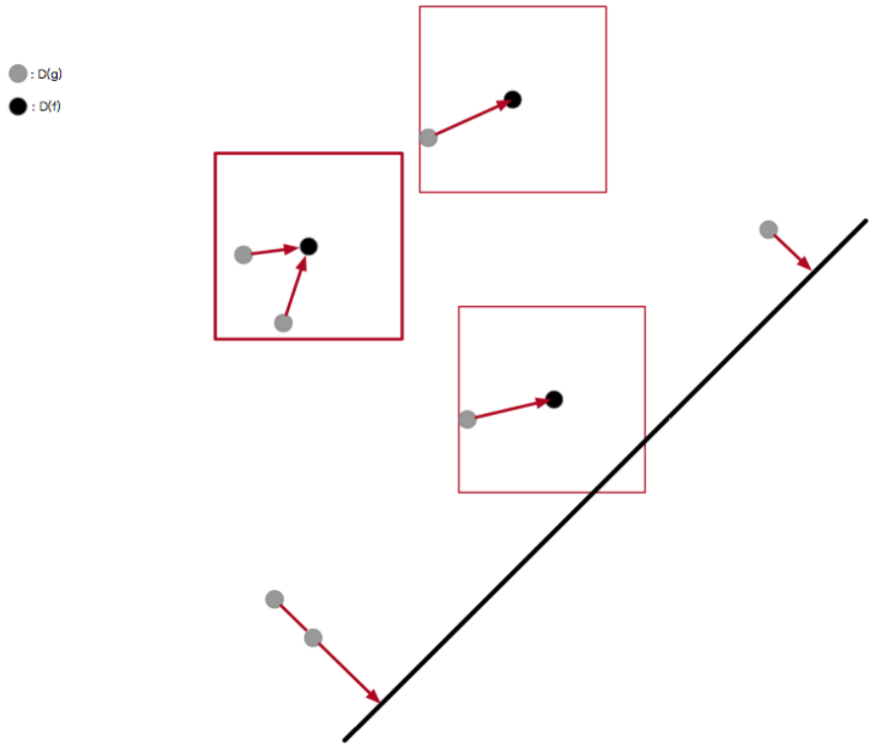


Figura 2.30: Emparejamiento de distancia menor que ϵ entre los puntos de $\mathrm{Dgm}(f)$ y $\mathrm{Dgm}(g)$, para el caso en el que g *es muy cercano a* f. Fuente: [12]

Como la biyección empareja puntos que están a distancia menor o igual que ϵ, concluimos que la distancia bottleneck entre $\mathrm{Dgm}(f)$ y $\mathrm{Dgm}(g)$ es menor o igual que ϵ. □

Estabilidad para la distancia bottleneck con funciones PL

Nos acercaremos un poco más a la demostración para el caso general, comprobando primero la estabilidad para dos funciones PL $\hat{f}$ y $\hat{g}$ definidas en un complejo simplicial K. Recordamos que vimos en la sección 2.1.4 que las funciones PL eran tame.

Definimos la *combinación convexa* de $\hat{f}$ y $\hat{g}$ como $h_\lambda : (1-\lambda)\hat{f} + \lambda\hat{g}$, para $\lambda \in [0,1]$. Esta familia uniparamétrica de combinaciones convexas forma una interpolación lineal entre las funciones $h_0 = \hat{f}$ y $h_1 = \hat{g}$.

Lema 2.2.7 (Lema de la interpolación)**.** *Sea K un complejo simplicial y sean $\hat{f}, \hat{g} : K \to \mathbb{R}$ dos funciones PL. Entonces,*

$$W_\infty(\mathrm{Dgm}(\hat{f}), \mathrm{Dgm}(\hat{g})) \leq \|\hat{f} - \hat{g}\|_\infty .$$

Demostración. Descompondremos esta interpolación lineal en suficientes pequeños pasos de forma que podamos usar el *Lema de la biyección sencilla* en cada uno de ellos. Sea $c = \|\hat{f} - \hat{g}\|_\infty$.

Podemos observar que para que para todo $\lambda \in [0,1]$, h_λ es tame y que $\delta(\lambda) = \delta_{h_\lambda} > 0$. Para garantizar que h_λ es tame veremos que h_λ es una función PL. Como $\hat{f}$ y $\hat{g}$ son PL, entonces $\hat{f}(x) = \sum_i b_i(x)\hat{f}(u_i)$ y $\hat{g}(x) = \sum_i b_i(x)\hat{g}(u_i)$, donde u_i son los vértices de K y $b_i(x)$ son las coordenadas baricéntricas de x. Luego,

$$\begin{aligned} h_\lambda(x) &= (1-\lambda)\hat{f}(x) + \lambda\hat{g}(x) \\ &= (1-\lambda)\sum_i b_i(x)\hat{f}(u_i) + \lambda\sum_i b_i(x)\hat{g}(u_i) \\ &= \sum_i b_i(x)\hat{f}(u_i) - \lambda\sum_i b_i(x)\hat{f}(u_i) + \lambda\sum_i b_i(x)\hat{g}(u_i) \\ &= \sum_i b_i(x)((1-\lambda)\hat{f}(u_i) + \lambda\hat{g}(u_i))\,, \end{aligned}$$

por lo que h_λ es PL.

Se sigue que el conjunto de los intervalos abiertos $J_\lambda = (\lambda - \delta(\lambda)/4c, \lambda + \delta(\lambda)/4c) \subset \mathbb{R}$ forman un recubrimiento abierto del intervalo $[0,1]$. Como $[0,1]$ es compacto, entonces un subrecubrimiento minimal C' de C será finito. Sean $\lambda_1 < \lambda_2 < ... < \lambda_n$ los puntos medios de los intervalos de C'. Como C' es minimal y los intervalos son abiertos, entonces cualquier par de intervalos consecutivos J_{λ_i} y $J_{\lambda_{i+1}}$ tienen intersección no vacía. Luego,

$$\lambda_{i+1} - \lambda_i \leq \frac{\delta(\lambda_{i+1}) + \delta(\lambda_i)}{4c} \leq \frac{\text{máx}\{\delta(\lambda_{i+1}), \delta(\lambda_i)\}}{2c}\,.$$

Por definición de c, se cumple que $\|h_{\lambda_i} - h_{\lambda_{i+1}}\|_\infty = c(\lambda_{i+1} - \lambda_i)$, ya que

$$\begin{aligned} \|h_{\lambda_i} - h_{\lambda_{i+1}}\|_\infty = \|(1-\lambda_i)\hat{f} + \lambda_i\hat{g} - (1-\lambda_{i+1})\hat{f} - \lambda_{i+1}\hat{g}\|_\infty = \\ = \|(\lambda_{i+1} - \lambda_i)\hat{f} - (\lambda_{i+1} - \lambda_i)\hat{g}\|_\infty = |\lambda_{i+1} - \lambda_i|\|\hat{f} - \hat{g}\|_\infty = c(\lambda_{i+1} - \lambda_i)\,. \end{aligned}$$

Como consecuencia, $\|h_{\lambda_i} - h_{\lambda_{i+1}}\|_\infty \leq \text{máx}\{\delta(\lambda_{i+1}), \delta(\lambda_i)\}/2$, lo que implica que h_{λ_i} *esta muy cercana a* $h_{\lambda_{i+1}}$ o al revés. Entonces, aplicando el *Lema de la biyección sencilla*, se sigue que $W_\infty(\text{Dgm}(h_{\lambda_i}), \text{Dgm}(h_{\lambda_i})) \leq \|h_{\lambda_i} - h_{\lambda_{i+1}}\|_\infty$, para todo $1 \leq i \leq n-1$. Siendo $\lambda_0 = 0$ y $\lambda_{n+1} = 1$, se sigue dando la desigualdad anterior para $i = 0$ e $i = n$, ya que h_0 *es muy cercana a* h_1 y h_1 *es muy cercana a* h_n.

Haciendo uso de la *desigualdad triangular de* W_∞ obtenemos el resultado,

$$\begin{aligned} W_\infty(\text{Dgm}(\hat{f}), \text{Dgm}(\hat{g})) \overset{Des.\ Triang.}{\leq} \sum_{i=0}^{n} W_\infty(\text{Dgm}(h_{\lambda_i}), \text{Dgm}(h_{\lambda_i})) \leq \sum_{i=0}^{n} \|h_{\lambda_i} - h_{\lambda_{i+1}}\|_\infty \; = \\ = c\sum_{i=0}^{n}(\lambda_{i+1} - \lambda_i) = c(\lambda_{n+1} - \lambda_0) = c(1-0) = \|\hat{f} - \hat{g}\|_\infty\,. \end{aligned}$$

□

Estabilidad para la distancia bottleneck con funciones tame

Tenemos todos los resultados necesarios para poder demostrar el *Teorema de estabilidad para la distancia bottleneck con funciones tame*, el cual recordamos a continuación:

Teorema 2.2.1 (Teorema de estabilidad para funciones tame)**.** *Sean $\mathbb{X}$ un espacio topológico triangulable y $f, g : \mathbb{X} \to \mathbb{R}$ dos funciones tame continuas. Entonces,*

$$W_\infty(\mathrm{Dgm}(f), \mathrm{Dgm}(g)) \leq \|f - g\|_\infty .$$

Adicionalmente, necesitaremos recordar un par de nociones de análisis matemático:

Definición 2.2.4. Dados dos espacios métricos (X, d_X) y (Y, d_Y), y $M \subset X$ entonces una función $f : M \to Y$ se llama *uniformemente continua* en M cuando, para cada $\epsilon > 0$, puede encontrarse $\delta > 0$ tal que, si $x_1, x_2 \in M$ verifican que $d_X(x_1, x_2) < \delta$, entonces $d_Y(f(x_1), f(x_2)) < \epsilon$.

Teorema 2.2.8 (Teorema de Heine–Cantor [14])**.** *Sean E y F espacios métricos y $f : E \to F$ una función continua. Si E es compacto, entonces f es uniformemente continua.*

Demostración del teorema 2.2.1 (Teorema de estabilidad para funciones tame).
Por la definición de espacio triangulable, existe un complejo simplicial (finito) L y un homeomorfismo $\Phi : |L| \to \mathbb{X}$. Se observa que el diagrama de persistencia es invariante por este tipo de cambio de variable, es decir, $f \circ \Phi : |L| \to \mathbb{R}$ es tame y tiene el mismo diagrama de persistencia que f.

Sea $\delta > 0$ lo suficientemente pequeño. Como f y g son continuas, L es compacto, entonces por el *Teorema de Heine-Cantor* f y g son uniformemente continuas, lo que nos garantiza que existe una subdivisión K de L tal que

$$|f \circ \Phi(u) - f \circ \Phi(v)| \leq \delta ,$$
$$|g \circ \Phi(u) - g \circ \Phi(v)| \leq \delta ,$$

cuando u y v son puntos de un mismo símplice de K, ya que podemos obtener esta subdivisión K de L con la propiedad de que el diámetro de cada símplice sea tan pequeño como sea necesario para cumplir la condición de la continuidad uniforme de f y g.

Sean $\hat{f}, \hat{g} : \mathrm{Sd}\, K \to \mathbb{R}$ funciones PL que aproximan a $f \circ \Phi$ y $g \circ \Phi$ en K. Por construcción de K, estas funciones satisfacen $\|\hat{f} - f \circ \Phi\|_\infty \leq \delta$ y $\|\hat{g} - g \circ \Phi\|_\infty \leq \delta$.

Para terminar utilizaremos la *desigualdad triangular de* W_∞ para acotar $W_\infty(\mathrm{Dgm}(f), \mathrm{Dgm}(g))$ superiormente por la suma de las distancias entre los diagramas de persistencia de las funciones adyacentes en la secuencia $f, \hat{f}, \hat{g}, g$.

Para el par $\hat{f}$ y $\hat{g}$ tenemos

$$\begin{aligned} W_\infty(\mathrm{Dgm}(\hat{f}), \mathrm{Dgm}(\hat{g})) &\leq \|\hat{f} - \hat{g}\|_\infty && (2.7)\\ &\leq \|\hat{f} - f \circ \Phi\|_\infty + \|f \circ \Phi - g \circ \Phi\|_\infty + \|\hat{g} - g \circ \Phi\|_\infty && (2.8)\\ &\leq \|f - g\|_\infty + 2\delta && (2.9) \end{aligned}$$

El punto 2.7 se debe al *Lema de la interpolación*, el 2.8 a la *desigualdad triangular de* d_∞ y por último, el punto 2.9 se debe a que $\hat{f}$ y $\hat{g}$ difieren como mucho δ de $f \circ \Phi$ y $g \circ \Phi$ respectivamente, y que $\|f - g\|_\infty = \|f \circ \Phi - g \circ \Phi\|_\infty$.

Para poder acotar la distancia bottleneck entre f y $\hat{f}$, supondremos que $\delta < \delta_f/2$, de forma que podemos aplicar el *Lema de la biyección sencilla*. Como el cambio de variables no afecta al diagrama de persistencia obtenemos que

$$W_\infty(\mathrm{Dgm}(f), \mathrm{Dgm}(\hat{f})) = W_\infty(\mathrm{Dgm}(f \circ \Phi), \mathrm{Dgm}(\hat{f})) \leq \delta\,.$$

Análogamente podemos acotar la distancia bottleneck entre g y $\hat{g}$ si asumimos que $\delta < \mín\{\delta_f/2, \delta_g/2\}$.

Luego, en total tenemos

$$\begin{aligned} W_\infty(\mathrm{Dgm}(f), \mathrm{Dgm}(g)) &\overset{Des.\ Triang.}{\leq} W_\infty(\mathrm{Dgm}(f), \mathrm{Dgm}(\hat{f})) + W_\infty(\mathrm{Dgm}(\hat{f}), \mathrm{Dgm}(\hat{g})) \\ &\quad + W_\infty(\mathrm{Dgm}(\hat{g}), \mathrm{Dgm}(g)) \\ &\leq \|f - g\|_\infty + 4\delta \underset{\delta \to 0}{\longrightarrow} \|f - g\|_\infty \end{aligned}$$

Como la condición es cierta para todo $\delta > 0$, podemos hacer δ tan pequeño como queramos. Por tanto queda demostrado el *Teorema de estabilidad*. □

2.3. Implementaciones y cálculos

En esta sección daremos algunas evidencias computacionales de la estabilidad de los diagramas de persistencias, centrándonos filtraciones de complejos simpliciales asociadas a nubes de puntos con un cierto ruido. Para ello comenzaremos estudiando posibles algoritmos para calcular tanto la *distancia Hausdorff* como la *distancia bottleneck*.

2.3.1. Cálculo de la distancia Hausdorff

Definición 2.3.1. Sea A y B dos conjuntos de puntos. Se define la *distancia Hausdorff directa* entre A y B como el máximo de las distancias entre cada punto $x \in A$ y el punto $y \in B$ más cercano a x. Es decir,

$$\check{H}(A, B) = \sup_{x \in A} \ínf_{y \in B} \|x - y\|_\infty\,.$$

Observación. $\check{H}(A, B) \neq \check{H}(B, A)$ y por tanto la distancia Hausdorff directa no es simétrica.

Luego, la distancia de Hausdorff es el máximo de las distancias Hausdorff directas en ambas direcciones, es decir

$$H(A, B) = \máx\{\check{H}(A, B), \check{H}(B, A)\}\,.$$

Sea $A = \{x_1, x_2, ..., x_m\}$ y $B = \{y_1, y_2, ..., y_m\}$ los dos conjuntos de puntos en $\mathbb{R}^k$ y sea $\|x - y\|_\infty$ la distancia infinito entre x e y. Por lo tanto, podemos calcular de manera sencilla la distancia Hausdorff directa entre A y B de la siguiendo los pasos del algoritmo 1.

Algoritmo 1 Cálculo de la distancia Hausdorff directa

Entrada: Dos conjuntos finitos de puntos A y B
Salida: Distancia Hausdorff directa entre A y B

```
1: cmax ← 0
2: for x ∈ A do
3:     cmin ← ∞
4:     for y ∈ B do                    ▷ Calculamos d_∞(x, B) = ínf_{y∈B} d_∞(x, y)
5:         d ← ‖x − y‖_∞
6:         if d < cmin then
7:             cmin ← d
8:         end if
9:     end for
10:    if cmin > cmax then             ▷ Recalculamos el supremo
11:        cmax ← cmin
12:    end if
13: end for
14: return cmax
```

Obviamente, la complejidad del algoritmo 1 es del orden de $\mathcal{O}(n * m)$, donde $m = |A|$ y $n = |B|$. La distancia Hausdorff entre A y B será el máximo de los resultados de ejecutar el algoritmo 1 en ambas direcciones, y por lo tanto la complejidad de calcular la distancia Hausdorff de este modo es de $\mathcal{O}(n * m)$.

Sin embargo, existen implementaciones del cálculo de la distancia Hausdorff que tienen complejidad del orden de $\mathcal{O}(m)$ en el mejor de los casos y $\mathcal{O}(n * m)$ en el peor de los casos [15].

2.3.2. Cálculo de la distancia bottleneck

En esta sección veremos los algoritmos propuestos en [2], donde el cálculo de la distancia bottleneck entre dos diagramas de persistencia se reduce a la obtención de un emparejamiento óptimo en un grafo bipartido.

Obtención de la distancia a partir de emparejamientos

Empezaremos viendo cómo podemos obtener la distancia bottleneck entre diagramas de persistencia a partir de emparejamientos de un grafo bipartido.

Sea X e Y dos diagramas de persistencia, para los que asumimos que están formados por un número finito de puntos fuera de la diagonal e infinitos puntos en ella. Denotamos X_0 al multiconjunto finito de los puntos fuera de la diagonal en X y X_0' a la proyección ortogonal de X_0 sobre la diagonal. Por tanto, construimos el grafo bipartido completo

$$G = (U \dot\cup V, A), \text{ con } U = X_0 \dot\cup Y_0', V = Y_0 \dot\cup X_0', \text{ y } A = U \times V\,,$$

donde $U \dot\cup V$ denota la unión disjunta de los conjuntos U y V.

En este grafo introducimos la función de coste $c : A \to \mathbb{R}$ donde a cada arista $uv \in A$

se le asigna la la distancia infinito entre los puntos u y v:

$$c(uv) = \begin{cases} \|u - v\|_\infty & \text{si } u \in X_0 \text{ ó } v \in Y_0 \\ 0 & \text{si } u \in Y_0' \text{ y } v \in X_0' \end{cases}$$

Observación. Por construcción, la arista de coste mínimo que conecta un punto u fuera de la diagonal con un punto de la diagonal es uu', donde u' es la proyección ortogonal de u sobre la diagonal. Además, el coste de esta arista es la mitad de la persistencia de u.

Definición 2.3.2. Un *emparejamiento* en G es un subconjunto $M \subseteq A$ tal que dos aristas de M no tienen un vértice en común. Diremos que

- M es *maximal* si no existe un emparejamiento M' en G con $M \subset M'$.
- M es *máximo* si no existe un emparejamiento M' en G con card $M <$ card M'.
- M es *perfecto* si todos los vértices de G son extremo de alguna arista de M.

Como G es un grafo bipartido completo, todo emparejamiento máximo es también un emparejamiento perfecto.

Definición 2.3.3. Se define $G(\epsilon) = (U \dot\cup V, A_\epsilon)$ como el subgrafo de G que se obtiene al eliminar todas las aristas $uv \in A$ con coste $c(uv) > \epsilon$.

En este caso, todo emparejamiento perfecto en $G(\epsilon)$ es máximo, sin embargo, el opuesto no siempre es cierto.

Definición 2.3.4. Un *emparejamiento de coste mínimo* es un emparejamiento máximo que minimiza la suma de los costes de las aristas del emparejamiento. Denotaremos a esta suma como el *coste total* del emparejamiento.

Lema 2.3.1 (Lema de reducción [2])**.** *Sean X e Y dos diagramas de persistencia y sea $G = (U \dot\cup V, A)$ su correspondiente grafo bipartido. Entonces la distancia bottleneck entre X e Y es el menor ϵ tal que el subgrafo $G(\epsilon)$ tiene un emparejamiento perfecto.*

Por lo tanto, el cálculo de la distancia bottleneck entre diagramas de persistencia se reduce a la obtención de emparejamientos perfectos con coste mínimo en grafos bipartidos.

Emparejamientos en grafos bipartidos

Comenzaremos viendo cómo podemos obtener emparejamientos máximos en el grafo bipartido $G(\epsilon) = (U \dot\cup V, A_\epsilon)$. Para ello haremos uso de algoritmos iterativos, donde en cada paso mejoraremos el emparejamiento, hasta que no sea posible aumentarlo.

Definición 2.3.5. Sea M_i el emparejamiento tras realizar i iteraciones. Se define $D_i = (P, Q)$ como el digrafo tal que

- $P = (U \dot\cup V) \cup \{s, t\}$, donde s se denota como fuente y t como sumidero.
- $Q = Q_1 \cup Q_2$, donde
  - Q_1 son las aristas $x \in A_\epsilon$ tal que x va de V a U si pertenece al emparejamiento M_i, y x va de U a V en caso contrario.

- Q_2 son las aristas que van desde s a los vértices no emparejados $u \in U$, más las aristas que van desde los vértices no emparejados $v \in V$ a t.

En la figura 2.31 podemos observar un ejemplo del digrafo D_i asociado a un emparejamiento M_i.

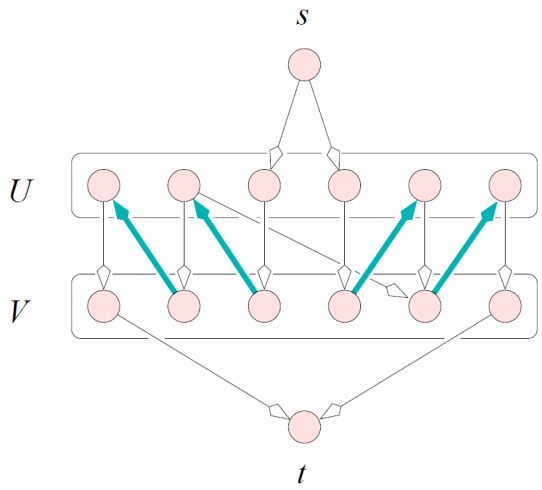


Figura 2.31: Digrafo asociado a un emparejamiento de cuatro aristas. Fuente: [2]

Definición 2.3.6. Un *camino de M_i-aumento* es un camino dirigido desde s hasta t el cual visita un vértice de D_i como máximo una vez.

Claramente, si tenemos un camino de M_i-aumento con k vértices no contenidos en M_i y $k-1$ vértices en M_i, entonces podemos mejorar el emparejamiento sustituyendo los k vértices que no estaban en M_i por los $k-1$ vértices que si estaban en M_i. Cuando hacemos esta mejora, decimos que hemos *aumentado* el emparejamiento usando el camino.

Lema 2.3.2 (Lema de Berge)**.** *M_i es un emparejamiento máximo de $G(\epsilon)$ si y sólo si $G(\epsilon)$ no contiene caminos de M_i-aumento.*

Luego, para obtener un emparejamiento máximo de $G(\epsilon)$ seguiremos los siguientes pasos:

Algoritmo 2 Obtención de emparejamientos máximos

Entrada: $G(\epsilon) = (U \dot\cup V, A_\epsilon)$ grafo bipartido
Salida: M_i es un emparejamiento máximo de $G(\epsilon)$

```
M_0 ← ∅
i ← 0
while existe un camino de M_i-aumento en D_i do
    aumentar M_i usando el camino para obtener M_{i+1}
    i ← i + 1
end while
return M_i
```

Este algoritmo terminará como mucho en n iteraciones, siendo $n = \text{card } U = \text{card } V$, ya que en cada iteración se aumenta el tamaño del emparejamiento en uno. Podemos hacer uso de la *búsqueda en anchura* y la *búsqueda en profundidad* para encontrar

caminos de M_i-aumento en un tiempo proporcional al número de aristas en A_ϵ. Por lo que la complejidad del algoritmo es del orden de $\mathcal{O}(n^3)$.

Se puede obtener una complejidad del orden de $\mathcal{O}(n^{5/2})$ implementando el algoritmo que se muestra en [2]. Este hace uso de la *búsqueda en anchura* para etiquetar los vértices con su distancia a s y después usa la *búsqueda en profundidad* para construir un conjunto maximal de múltiples caminos de M_i-aumento.

Emparejamientos de coste mínimo en grafos bipartidos

Para calcular el menor ϵ tal que $G(\epsilon)$ tiene un emparejamiento perfecto, seguiremos una variante del método húngaro, el cual se utiliza para resolver problemas de asignación [16].

Propiedad 2.3.1 ([2])**.**

A. Si el subgrafo $G(0)$, que consiste en las aristas de coste cero de G, tiene un emparejamiento perfecto, entonces es un emparejamiento de coste mínimo. Es más, su coste total es cero.

B. Restar la misma cantidad al coste de todas las aristas incidentes a un vértice de G afecta a todos los emparejamientos perfectos de la misma forma. En particular, un emparejamiento perfecto minimiza el coste total antes de las restas de la cantidades si y sólo si sigue minimizándolo tras las restas de las cantidades.

Así pues, empezaremos construyendo un emparejamiento máximo en $G(0)$. Si es un emparejamiento perfecto ya hemos acabado y por lo tanto la distancia bottleneck entre los diagramas de persistencia es 0. En otro caso, cambiaremos los costes de las aristas de G preservando el orden de los emparejamientos perfectos en G por coste total. Para ello introducimos las *funciones de reducción* $d_i : U \times V \to \mathbb{R}$. Partiendo de $d_0(x) = 0$ para todos los vértices de G, el algoritmo cambiará el valor de la función de reducción en cada iteración i.

Definición 2.3.7. Sea $c(xy)$ el coste original de la arista $xy \in G$. Se define el *coste modificado* tras i iteraciones como

$$c_i(xy) = c(xy) - d_i(x) - d_i(y) > 0\,.$$

Sea G_i el grafo G con los costes modificados por d_i, entonces el algoritmo construirá iterativamente emparejamientos máximos en $G_i(0)$, que coincide con el subgrafo resultante al eliminar las aristas con peso no nulo de G_i. Incrementando el número de aristas del emparejamiento máximo en uno por cada iteración, obtendremos el emparejamiento perfecto en n iteraciones.

Análogo al método Húngaro, iremos añadiendo aristas de coste modificado cero al emparejamiento en cada iteración y para generar ceros adicionales en los costes modificados de las aristas seleccionaremos el menor de los costes totales de los caminos de M_i-aumento como cantidad que variará la función de reducción.

Sea M_i un emparejamiento máximo en $G_i(0)$ y sea D_i el digrafo asociado al emparejamiento M_i y G_i. Si M_i no es un emparejamiento perfecto en G_i, entonces no es un emparejamiento máximo en G_i, y por lo tanto existirá un camino de M_i-aumento en D_i.

Por definición $c_i(sy) = c_i(xt) = 0$ para todo $x \in U$ e $y \in V$. Se denota como *coste total* de un camino de M_i-aumento como la suma de los costes modificados de sus aristas. Obtendremos el camino de M_i-aumento π que minimiza el coste total, a través del *algoritmo de Dijkstra* con una complejidad del orden de $\mathcal{O}(n^2)$.

Como hacíamos en el algoritmo 2, aumentamos M_i usando π para obtener M_{i+1}. Vamos a garantizar que podemos cambiar la función de reducción de forma que todas las aristas del emparejamiento M_{i+1} tienen coste modificado cero. Para ello definimos $\gamma_i(x)$ como el coste total mínimo de los caminos desde s hasta x.

De esta forma, actualizamos las funciones de reducción a

$$d_{i+1} = \begin{cases} d_i(x) - \gamma_i(x) & \text{si } x \in U \\ d_i(x) + \gamma_i(x) & \text{si } x \in V \end{cases}$$

Luego, para todos los vértices $u \in U$ y $v \in V$, el nuevo coste modificado de la arista uv es:

$$c_{i+1}(uv) = c(uv) - d_i(u) - d_i(v) + \gamma_i(u) - \gamma_i(v)\,.$$

Propiedad 2.3.2 ([2])**.** Sea M_{i+1} el emparejamiento máximo obtenido al aumentar M_i. Entonces, $c_{i+1}(uv) \geq 0$ para toda arista uv en G_i, y $c_{i+1}(uv) = 0$ para toda arista $uv \in M_{i+1}$.

La propiedad anterior garantiza que en la última iteración obtenemos el emparejamiento perfecto de coste total mínimo, y por tanto la distancia bottleneck entre los diagramas de persistencia X e Y es igual al máximo de los costes originales de las aristas de dicho emparejamiento perfecto, es decir

$$W_\infty(X,Y) = \max_{xy \in M_n} c(xy), \text{ siendo } n = \text{card } U = \text{card } V\,.$$

Como tenemos n iteraciones en las cuales cada una aplicamos el algoritmo de Dijkstra, entonces la complejidad del cálculo de la distancia bottleneck siguiendo el algoritmo comentado es del orden de $\mathcal{O}(n^3)$.

2.3.3. Pruebas

La implementación del cálculo se ha realizado en Python y el código se puede encontrar en el Anexo 1.

Subsección por hacer

Capítulo 3

Resultados y conclusiones

Sección por hacer

Resumen de resultados obtenidos en el TFG. Y conclusiones personales del estudiante sobre el trabajo realizado.

Capítulo 4

Análisis de impacto

Sección por hacer

En este capítulo se realizará un análisis del impacto potencial de los resultados obtenidos durante la realización del TFG, en los diferentes contextos para los que se aplique:

- Personal
- Empresarial
- Social
- Económico
- Medioambiental
- Cultural

En dicho análisis se destacarán los beneficios esperados, así como también los posibles efectos adversos.

Se recomienda analizar también el potencial impacto respecto a los Objetivos de Desarrollo Sostenible (ODS), de la Agenda 2030, que sean relevantes para el trabajo realizado (ver enlace)

Además, se harán notar aquellas decisiones tomadas a lo largo del trabajo que tienen como base la consideración del impacto.

Bibliografía

[1] D. Reinsel, J. Gantz, and J. Rydning. Data age 2025: The evolution of data to life-critical. Seagate. [Online]. Available: https://www.import.io/wp-content/uploads/2017/04/Seagate-WP-DataAge2025-March-2017.pdf

[2] H. Edelsbrunner and J. Harer, *Computational Topology: An Introduction*. American Mathematical Society, 01 2010.

[3] H. Edelsbrunner and J. Harer, "Persistent homology—a survey," *Discrete & Computational Geometry - DCG*, vol. 453, 01 2008.

[4] D. Cohen-Steiner, H. Edelsbrunner, and J. Harer, "Stability of persistence diagrams," *Discrete & Computational Geometry*, vol. 37, no. 1, pp. 103–120, Jan 2007. [Online]. Available: https://doi.org/10.1007/s00454-006-1276-5

[5] M. Ulmer, L. Ziegelmeier, and C. M. Topaz, "A topological approach to selecting models of biological experiments," *PLOS ONE*, vol. 14, no. 3, p. e0213679, Mar. 2019. [Online]. Available: https://doi.org/10.1371/journal.pone.0213679

[6] P. Yale, *Geometry and Symmetry*, ser. Dover books on advanced mathematics. Dover Publications, 2014. [Online]. Available: https://books.google.es/books?id=PjOlBQAAQBAJ

[7] A. Hatcher, *Algebraic topology*. Cambridge: Cambridge University Press, 2002.

[8] M. D. Crossley, *Essential Topology*. Springer London, 2005.

[9] J. Curry. Counting embedded spheres with the same persistence. University at Albany SUNY. [Online]. Available: http://www.fields.utoronto.ca/talks/Counting-Embedded-Spheres-same-Persistence

[10] J. Curry, "The fiber of the persistence map for functions on the interval," 2019.

[11] H. Ricardo, *A Modern Introduction to Linear Algebra*. Chapman and Hall/CRC, Oct. 2009. [Online]. Available: https://doi.org/10.1201/b16027

[12] X. Kong. Stability theorem. Eindhoven, university of Technology. [Online]. Available: https://www.win.tue.nl/~kbuchin/teaching/2IMA00/2018/Slides/stability.pdf

[13] B. Binegar. Lecture 14: The isomorphism theorems. Oklahoma State University. [Online]. Available: https://math.okstate.edu/people/binegar/4063-5023/4063-5023-l14.pdf

[14] W. Rudin, *Principles of mathematical analysis*, 3rd ed. McGraw-Hill New York, 1976. [Online]. Available: http://www.loc.gov/catdir/toc/mh031/75017903.html

[15] A. A. Taha and A. Hanbury, "An efficient algorithm for calculating the exact hausdorff distance," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 37, no. 11, pp. 2153–2163, Nov. 2015. [Online]. Available: https://doi.org/10.1109/tpami.2015.2408351

[16] H. W. Kuhn, "The hungarian method for the assignment problem," *Naval Research Logistics Quarterly*, vol. 2, no. 1-2, pp. 83–97, 1955. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1002/nav.3800020109

Anexos

.1. Código en Python

.1.1. Implementación de las distancias Hausdorff y bottleneck

Anexo 1: Implementación de las distancias Hausdorff y bottleneck

```
# -*- coding: utf-8 -*-
"""
Created on Tue May  4 15:45:56 2021.

@author: Alejandro
"""

import numpy as np
import networkx as nx
from networkx.algorithms import bipartite
import matplotlib.pyplot as plt
import sympy as sy
import gudhi

# variable curvas
t = sy.symbols('t', real=True)


def distanciaEuclidea(p1, p2):
    """
    Distancia euclídea entre los puntos p1 y p2.

    p1: list.
    p2: list.
    """
    p1Np = np.array(p1)
    p2Np = np.array(p2)
    return np.sqrt(np.dot(p1Np - p2Np, p1Np - p2Np))


def distanciaInf(x, y):
    """
    Calcular la distancia infiniro entre los puntos x e y.

    x: numpy.ndarray.
    y: numpy.ndarray.
    """
    res = 0
    if x[1] == float('inf') and y[1] == float('inf'):
        res = abs(x[0]-y[0])
    elif x[1] == float('inf') and y[1] != float('inf'):
        res = float('inf')
    elif x[1] != float('inf') and y[1] == float('inf'):
        res = float('inf')
    else:
        res = max(abs(x[0]-y[0]), abs(x[1]-y[1]))

```

```
48      return res
49
50
51  def hausdorffDir(A, B):
52      """
53      Calcular la distancia Hausdorff directa entre los conjuntos de puntos A y B.
54
55      A: numpy.array.
56      B: numpy.array.
57      """
58      cmax = 0
59      for x in A:
60          cmin = float('inf')
61          for y in B:
62              d = distanciaInf(x, y)
63              if d < cmin:
64                  cmin = d
65
66          if cmin > cmax:
67              cmax = cmin
68
69      return cmax
70
71
72  def hausdorff(A, B):
73      """
74      Calcular la distancia Hausdorff entre los conjuntos de puntos A y B.
75
76      A: numpy.array.
77      B: numpy.array.
78      """
79      return max(hausdorffDir(A, B),  hausdorffDir(B, A))
80
81
82  def grafoBottleneck(X, Y):
83      """
84      Obtener el grafo bipartido de para el emparejamiento de la distancia bottleneck.
85
86      X: numpy.array.
87      y: numpy.array.
88      """
89      X0 = list()
90      X0_inf = list()
91
92      Y0 = list()
93      Y0_inf = list()
94      for x in X:
95          if x[0] != x[1]:
96              if x[1] == float("inf"):
97                  X0_inf.append(x[0])
98              else:
99                  X0.append((x[0], x[1]))
100
101     for y in Y:
102         if y[0] != y[1]:
103             if y[1] == float("inf"):
104                 Y0_inf.append(y[0])
105             else:
106                 Y0.append((y[0], y[1]))
107
108     X0_ = [((x[0]+x[1])/2, (x[0]+x[1])/2) for x in X0]
109     Y0_ = [((y[0]+y[1])/2, (y[0]+y[1])/2) for y in Y0]
110     U = X0 + Y0_
111     V = Y0 + X0_
112     n = len(U)
113     G = nx.Graph()
114     G.add_nodes_from([(f"u{i}", {'coord': U[i]}) for i in range(0, n)], bipartite=0)
115     G.add_nodes_from([(f"v{i}", {'coord': V[i]}) for i in range(0, n)], bipartite=1)
116     edges = list()
117     for i in range(0, n):
```

```
118         for j in range(0, n):
119             d = 0.0
120             if U[i] in X0 or V[j] in Y0:
121                 d = distanciaInf(U[i], V[j])
122 
123             edges.append((f"u{i}", f"v{j}", d))
124 
125     G.add_weighted_edges_from(edges)
126 
127     distPinf = 0
128 
129     if len(X0_inf) != len(Y0_inf):
130         distPinf = float("inf")
131     else:
132         distPinf = max([abs(x-y) for x, y in zip(sorted(X0_inf), sorted(Y0_inf))]+[0])
133 
134     return G, U, V, distPinf
135 
136 
137 def subgrafoG(G, i):
138     """
139     Obtener el subgrafo de G formado por las aristas que tienen peso menor o igual a i.
140 
141     G: networkx.Graph.
142     i: float.
143     """
144     G0 = nx.Graph([(u, v, d) for u, v, d in G.edges(data=True) if d['weight'] <= i])
145     for n, d in G0.nodes(data=True):
146         d["bipartite"] = G.nodes[n]["bipartite"]
147 
148     return G0
149 
150 
151 def subgrafoG_todosV(G, i):
152     """
153     Obtener el subgrafo generador de G formado por las aristas que tienen peso menor o igual a
          i.
154 
155     G: networkx.Graph.
156     i: float.
157     """
158     G0 = nx.Graph()
159     G0.add_nodes_from(G.nodes(data=True))
160     G0.add_edges_from([(u, v, d) for u, v, d in G.edges(data=True) if d['weight'] <= i])
161 
162     return G0
163 
164 
165 def cambiarPesos(Gi, length):
166     """
167     Obtener los pesos de la nueva iteración a partir del diccionario de distancia obtenido de
          Dijkstra.
168 
169     Gi: networkx.Graph.
170     length: dict.
171     """
172     for u, v, d in Gi.edges(data=True):
173         d['weight'] += length[u] - length[v]
174         if d['weight'] < 0:
175             d['weight'] = 0
176 
177     return 0
178 
179 
180 def digrafoAsociado(G, M, U=None, V=None):
181     """
182     Obtener el digrafo asociado al emparejamiento M y el grafo bipartido M con vertices U+V.
183 
184     Gi: networkx.Graph.
185     M: dict.
```

```
186     U: list.
187     V: list.
188     """
189     if U is None or V is None:
190         U, V = bipartite.sets(G)
191 
192     D = nx.DiGraph()
193     keysM = M.keys()
194 
195     aristas = list()
196     for u, v, d in G.edges(data='weight'):
197         if u in U:
198             if u not in keysM or M[u] != v:
199                 aristas.append((u, v, d))
200             else:
201                 aristas.append((v, u, d))
202         else:
203             if u not in keysM or M[u] != v:
204                 aristas.append((v, u, d))
205             else:
206                 aristas.append((u, v, d))
207 
208     D.add_weighted_edges_from(aristas)
209     D.add_weighted_edges_from([("s", u, 0) for u in U if u not in keysM])
210     D.add_weighted_edges_from([(v, "t", 0) for v in V if v not in keysM])
211 
212     return D
213 
214 
215 def plotMatching(G, M, U):
216     """
217     Representar el emparejamiento M en el grafo bipartido G con vertices superiores en U.
218 
219     G: networkx.Graph.
220     M: dict.
221     U: list.
222     """
223     plt.figure(figsize=(8, 6))
224     pos = nx.drawing.layout.bipartite_layout(G, U)
225     nx.draw_networkx(G, pos=pos)
226     edgelist = [(u, M[u]) for u in U if u in M.keys()]
227     nx.draw_networkx_edges(G, pos, edgelist=edgelist, width=2.5, edge_color='blue')
228 
229 
230 def bottleneck(X, Y, plot=False, debug=False):
231     """
232     Calcular la distancia bottleneck entre los conjuntos de puntos X y Y.
233 
234     Si se quiere que se muestren los emparejamientos poner plot=True.
235     Si se quiere información de la traza debug=True.
236 
237     A: numpy.array.
238     B: numpy.array.
239     plot: boolean.
240     debug: boolean.
241     """
242     G, _, _, distPinf = grafoBottleneck(X, Y)
243     if distPinf == float("inf"):
244         return float("inf")
245     else:
246         U, V = bipartite.sets(G)
247         Gi = G.copy()
248         Mi = dict()
249 
250         if plot:
251             plotMatching(Gi, Mi, U)
252 
253         while not nx.is_perfect_matching(G, Mi):
254             Di = digrafoAsociado(Gi, Mi, U, V)
255             length, path = nx.single_source_dijkstra(Di, "s")
```

```
256             caminoAumento = path["t"][1:-1]
257
258             for i in range(0, len(caminoAumento)):
259                 x = caminoAumento[i]
260
261                 if i % 2 == 0:
262                     Mi[x] = caminoAumento[i+1]
263                 else:
264                     Mi[x] = caminoAumento[i-1]
265
266             if plot:
267                 plotMatching(Gi, Mi, U)
268
269             if debug:
270                 print("\n", {u: Mi[u] for u in U if u in Mi}, "\n")
271                 print(nx.is_perfect_matching(subgrafoG_todosV(Gi, 0), Mi), "\n")
272                 print("MAX", max([G[u][Mi[u]]["weight"] for u in U if u in Mi.keys()]+[
                        distPinf]), "\n")
273                 print(caminoAumento, "\n")
274                 print([(u, v, d) for u, v, d in Gi.edges(data=True) if d["weight"] < 0], "\n")
275
276             cambiarPesos(Gi, length)
277
278         return max([G[u][Mi[u]]["weight"] for u in U if u in Mi.keys()]+[distPinf])
279
280
281 if __name__ == "__main__":
282
283     A = np.array([(2, 4), (3, 2), (0, 0), (0, 0.8), (4, 5.2)])
284     B = np.array([(2.8, 4), (3, 3), (4.2, 5.8)])
285     C = np.array([(2, 4), (4, 2), (0, 0)])
286     D = np.array([(2.8, 4), (4.8, 2.8), (0, 0.8)])
287
288     diag1 = np.array([(2.7, 3.7), (9.6, 14.), (34.2, 34.974), (3., float('inf'))])
289     diag2 = np.array([(2.8, 4.45), (9.5, 14.1), (3.2, float('inf'))])
290
291     G, _, _, d = grafoBottleneck(diag1, diag2)
292
293     print("Hausdorff: ", hausdorffDir(diag1, diag2), "\n")
294     # print(G.nodes(data=True), "\n")
295     # print(G.edges(data=True), "\n")
296     print("Bottleneck: ", bottleneck(diag1, diag2), "=", gudhi.bottleneck_distance(diag1,
            diag2))
```

.1.2. Implementación de clase para el cálculo de la homología y persistencia de complejos simpliciales

Anexo 2: Implementación de la homología y persistencia de complejos simpliciales

```
1  # -*- coding: utf-8 -*-
2  """
3  Created on Fri Sep 18 15:24:22 2020.
4
5  @author: Alejandro
6  """
7
8  from itertools import combinations, chain
9  import networkx as nx
10 import matplotlib.pyplot as plt
11 from scipy.spatial import Delaunay, Voronoi, voronoi_plot_2d
12 import matplotlib.colors
13 import numpy as np
14 from numpy.linalg import matrix_rank
15 import imageio
16 import sympy as sy
17 import math
18 import os
```

```
19
20
21 # variable curvas
22 t = sy.symbols('t', real=True)
23
24
25 def puntosCurvaRuido(curva, t, t0, t1, numPuntos=10, mu=0, sigma=0.1):
26     """
27     Obtener conjunto discretos de puntos de una curva con ruido.
28
29     curva: list.
30     t: t sympy symbol
31     t0: float
32         Inicio intervalo.
33     t1: float
34         Final intervalo.
35     numPuntos: int. Por defecto 10.
36     mu: float. Por defecto 0
37         Media para la distribución normal.
38     sigma: float. Por defecto 0.1
39         Desviación típica para la distribución normal.
40     """
41     valores = np.linspace(t0, t1, num=numPuntos)
42     puntosCurva = np.array([[x.subs(t, v) for x in curva] for v in valores], dtype=np.float64)
43     ruido = np.random.normal(mu, sigma, [numPuntos, len(curva)])
44
45     return puntosCurva + ruido
46
47
48 def low(v):
49     """
50     Cálculo del low de una columna. Devuelve -1 si el vector es nulo.
51
52     v: np.array.
53     """
54     for i in range(len(v)-1, -1, -1):
55         if (v[i] == 1):
56             return i
57
58     return -1
59
60
61 def pivotar(M, k, m):
62     """
63     Pivota en el elemento (k,m) intercambiando filas y columnas.
64
65     M: np.array.
66     k: int.
67     m: int.
68     """
69     if M[k, m] != 1:
70         encontrado = False
71         i = k
72         j = m + 1
73         while not encontrado and i < M.shape[0] and j < M.shape[1]:
74             if M[i, j] == 1:
75                 # Intercambio columnas
76                 M[:, [m, j]] = M[:, [j, m]]
77                 # Intercambio filas
78                 M[[k, i], :] = M[[i, k], :]
79                 encontrado = True
80
81             j += 1
82             if j == M.shape[1]:
83                 j = m
84                 i += 1
85     else:
86         encontrado = True
87
88     return encontrado
```

```
89
90
91  def sumaFilZ2(M, i, j):
92      """
93      Suma: filai + filaj (sobre la j).
94
95      M: np.array.
96      i: int.
97      j: int.
98      """
99      M[j, :] = (M[i, :] + M[j, :]) % 2
100
101
102 def sumaColZ2(M, i, j):
103     """
104     Suma: coli + colj (sobre la j).
105
106     M: np.array.
107     i: int.
108     j: int.
109     """
110     M[:, j] = (M[:, i] + M[:, j]) % 2
111
112
113 def normSmithZ2(M):
114     """
115     Obtener la forma normal de Smith de una matriz con coefs en Z2.
116
117     M: np.array.
118     """
119     n = 0
120     cols = M.shape[1]
121     fils = M.shape[0]
122     while n < cols and n < fils and pivotar(M, n, n):
123         # Recorrer fila
124         for j in range(n + 1, cols):
125             if M[n, j] == 1:
126                 sumaColZ2(M, n, j)
127         # Recorrer columna
128         for i in range(n + 1, fils):
129             if M[i, n] == 1:
130                 sumaFilZ2(M, n, i)
131         n += 1
132
133     return M
134
135
136 def powerset(iterable):
137     """
138     Optiene un chain con todos los subconjuntos del iterable.
139
140     iterable: iterable.
141     """
142     # "powerset([1,2,3]) --> () (1,) (2,) (3,) (1,2) (1,3) (2,3) (1,2,3)"
143     s = list(iterable)
144     return chain.from_iterable(combinations(s, r) for r in range(len(s) + 1))
145
146
147 def ordCaras(cara):
148     """
149     Relacion de orden de las caras de una filtracion.
150
151     cara: tuple().
152     """
153     return (cara[1], len(cara[0]) - 1, cara[0])
154
155
156 def distancia(p1, p2):
157     """
158     Distancia euclídea entre los puntos p1 y p2.
```

```
159
160     p1: list.
161     p2: list.
162     """
163     p1Np = np.array(p1)
164     p2Np = np.array(p2)
165     return np.sqrt(np.dot(p1Np - p2Np, p1Np - p2Np))
166
167
168 def radioCircunscrita(p1, p2, p3):
169     """
170     Dados los vertices de un triangulo obtenemos el radio de la cirunferencia circuncentra.
171
172     p1: tuple.
173     p2: tuple.
174     p3: tuple.
175     """
176     a = distancia(p1, p2)
177     b = distancia(p1, p3)
178     c = distancia(p2, p3)
179
180     s = (a + b + c) / 2
181
182     return (a * b * c) / (4 * np.sqrt(s * (s - a) * (s - b) * (s - c)))
183
184
185 def analisisComplejo(comp, simplice):
186     """
187     Alisis de las propiedades del complejo simplicial.
188
189     comp: Complejo.
190     simplice: set(tuple)
191         simplice como ref para ejemplos.
192     """
193     # Todas las caras del complejo
194     print(f"Todas las caras: {comp.getCaras()}")
195
196     # Dimension del complejo
197     dimComp = comp.dim()
198     print(f"Dimension: {dimComp}")
199
200     # Caras de cierta dimension
201     for n in range(dimComp + 1):
202         print(f"Todas las caras de dim {n}: {comp.getCarasN(n)}")
203
204     # Característica de Euler
205     print(f"Característica de Euler: {comp.caractEuler()}")
206
207     # Estrella
208     print(f"Estrella de {simplice}: {comp.st(simplice)}")
209
210     # Link
211     print(f"Link de {simplice}: {comp.lk(simplice)}")
212
213     # Componentes conexas
214     print(f"Numero de componentes conexas: {comp.compConexas()}")
215
216     # 1-esqueleto
217     print(f"El 1-esqueleto es: {comp.k_esqueleto(1)}")
218
219
220 def drawVor(puntos):
221     """
222     Representacion de las celdas de Voronoi de una nube de puntos.
223
224     puntos: np.array.
225     """
226     vor = Voronoi(puntos)
227     voronoi_plot_2d(vor, show_vertices=False, line_width=2,
228                     line_colors='blue', line_alpha=0.6)
```

```
229     plt.plot(puntos[:, 0], puntos[:, 1], 'ko')
230     return vor
231
232
233 def delaunay(puntos):
234     """
235     Generar el triangulacion de Delaunay y su representacion junto a las celdas de Voronoi.
236
237     puntos: np.array.
238     """
239     drawVor(puntos)
240
241     Del = Delaunay(puntos)
242     c = np.ones(len(puntos))
243     cmap = matplotlib.colors.ListedColormap("limegreen")
244     plt.tripcolor(puntos[:, 0], puntos[:, 1], Del.simplices, c, edgecolor="k", lw=2,
245                   cmap=cmap)
246     plt.plot(puntos[:, 0], puntos[:, 1], 'ko')
247     plt.show()
248
249     return Complejo([tuple(sorted(triangulo)) for triangulo in Del.simplices])
250
251
252 def alfaComplejo(puntos):
253     """
254     Genera la filtracion de alfa complejos de la triangulacion de Delaunay.
255
256     puntos: np.array.
257     """
258     Del = delaunay(puntos)
259     # Introducimos los 0-simplices
260     alfa = Complejo(Del.getCarasN(0))
261
262     # Introducimos los 2-simplices
263     traingNuev = ((t, radioCircunscrita(puntos[t[0]], puntos[t[1]], puntos[t[2]]))
264                   for t in Del.getCarasN(2))
265
266     for t in traingNuev:
267         alfa.setCaras([t[0]], t[1])
268
269     # Introducimos los 1-simplices
270     for arista in Del.getCarasN(1):
271         # print(arista)
272         p1 = puntos[arista[0]]
273         p2 = puntos[arista[1]]
274         pMedio = ((p2[0] + p1[0]) / 2, (p2[1] + p1[1]) / 2)
275         d = distancia(p1, p2) / 2
276
277         pesoTriangMin = -1
278         for triang in Del.getCarasN(2):
279             # print(arista, triang)
280             difTriangArista = set(triang) - set(arista)
281
282             if len(difTriangArista) == 1 and distancia(puntos[difTriangArista.pop()], pMedio)
                    < d:
283                 pesoTriang = alfa.umbral(triang)
284                 if pesoTriangMin < 0 or pesoTriang < pesoTriangMin:
285                     pesoTriangMin = pesoTriang
286
287         alfa.setCaras([arista], d if pesoTriangMin < 0 else pesoTriangMin)
288
289     return alfa
290
291
292 def plotalpha(puntos, K):
293     """
294     Representar el alpha complejo del complejo K.
295
296     puntos: np.array.
297     K: Complejo.
```

```
298     """
299     dim = K.dim()
300 
301     if dim > 1:
302         c = np.ones(len(puntos))
303         cmap = matplotlib.colors.ListedColormap("limegreen")
304         plt.tripcolor(puntos[:, 0], puntos[:, 1], list(K.getCarasN(2)), c, edgecolor="k", lw
                =2,
305                       cmap=cmap)
306 
307     plt.plot(puntos[:, 0], puntos[:, 1], 'ko')
308 
309     if dim > 0:
310         for arista in K.getCarasN(1):
311             p1 = puntos[arista[0]]
312             p2 = puntos[arista[1]]
313             plt.plot([p1[0], p2[0]], [p1[1], p2[1]], 'k')
314 
315     # plt.show()
316 
317 
318 def vietorisRips(puntos):
319     """
320     Calculo del complejo Vietoris Rips de una nube de puntos.
321 
322     puntos: np.array.
323     """
324     nsimplex = Complejo([tuple(range(len(puntos)))])
325     VR = Complejo(list(nsimplex.getCarasN(0)))
326 
327     for arista in nsimplex.getCarasN(1):
328         VR.setCaras([arista], 0.5 * distancia(puntos[arista[0]], puntos[arista[1]]))
329 
330     for i in range(2, len(puntos)):
331         for simplex in nsimplex.getCarasN(i):
332             lista = []
333 
334             for arista in combinations(simplex, 2):
335                 lista.append(0.5 * distancia(puntos[arista[0]], puntos[arista[1]]))
336             VR.setCaras([simplex], max(lista))
337 
338     return VR
339 
340 
341 class Complejo():
342     """Clase del complejo simplicial."""
343 
344     def __init__(self, carasMaximales=[]):
345         """
346         Complejo simplicial abstracto a partir de sus caras maximales.
347 
348         carasMaximales: list(tuple). Por defecto [].
349         """
350         # Concatenamos el los conjuntos obtenidos de cada cara maximal
351         self.caras = set()
352         for cara in carasMaximales:
353             if cara not in self.caras:
354                 self.caras |= set(tuple(sorted(list(c))) for c in powerset(cara))
355         # Quitamos el conjunto vacio
356         self.caras -= {()}
357 
358         # Añadimos peso
359         self.caras = set([(cara, 0.0) for cara in self.caras])
360 
361         self.carasOrd = sorted(list(self.caras), key=ordCaras)
362 
363         self.bettiNums = [-1 for a in range(self.dim() + 1)]
364 
365     def setCaras(self, carasNuevas, peso=0.0):
366         """
```

```
367         Insertar nuevas caras y sus correspondientes subconjuntos con un peso dado.
368 
369         carasNuevas: list(tuple).
370         peso: float. Por defecto 0.0.
371         """
372         diffDim = max([len(cara) - 1 for cara in carasNuevas]) - self.dim()
373         if diffDim > 0:
374             self.bettiNums.extend(-1 for i in range(diffDim))
375 
376         for cara in carasNuevas:
377             powerCaras = set(tuple(sorted(list(c))) for c in powerset(cara))
378             for caraGen in powerCaras:
379                 if caraGen == tuple():
380                     continue
381 
382                 encontrado = False
383                 for caraAnt in self.caras:
384                     if caraGen == caraAnt[0]:
385                         encontrado = True
386                         if caraAnt[1] > peso:
387                             self.caras -= {caraAnt}
388                             self.caras |= {(caraGen, peso)}
389 
390                         break
391 
392                 if not encontrado:
393                     self.caras |= {(caraGen, peso)}
394 
395         self.carasOrd = sorted(self.caras, key=ordCaras)
396 
397     def getCaras(self):
398         """Devuelve el conjunto de todas las caras del complejo simplicial."""
399         return set([cara[0] for cara in self.caras])
400 
401     def getCarasOrd(self):
402         """Devuelve el conjunto de las caras ordenadas segun su filtracion."""
403         return [cara[0] for cara in self.carasOrd]
404 
405     def umbrales(self):
406         """Devuelve el conjunto de las umbrales ordenados segun la filtracion."""
407         return list(dict.fromkeys([cara[1] for cara in self.carasOrd]))
408 
409     def umbral(self, cara):
410         """
411         Obtiene el umbral de una cara dada.
412 
413         cara: tuple.
414         """
415         index = 0
416         encontrado = False
417         while index < len(self.carasOrd) and not encontrado:
418             encontrado = self.carasOrd[index][0] == cara
419             index += int(not encontrado)
420 
421         return self.carasOrd[index][1] if encontrado else None
422 
423     def dim(self):
424         """Devuelve la dimensión del complejo simplicial."""
425         return max([len(caras[0]) for caras in self.caras]) - 1 if self.caras != set() else 0
426 
427     def getCarasN(self, dimension):
428         """
429         Devuelve el conjunto de todas las caras de dimension dada.
430 
431         dimension: int.
432         """
433         return set(c for c in self.getCaras() if len(c) == dimension + 1)
434 
435     def st(self, v):
436         """
```

```
437         Calcular la estrella del simplice v.
438 
439         v: set.
440         """
441         return set(c for c in self.getCaras() if v.issubset(c))
442 
443     def lk(self, v):
444         """
445         Calcular el de un simplice v.
446 
447         v: set.
448         """
449         # Calculamos la estrella de v
450         st = self.st(v)
451 
452         # Calculamos la estrella cerrada de v
453         st_ = set()
454         for cara in st:
455             if cara not in st_:
456                 st_ |= set(powerset(cara))
457         # Quitamos el conjunto vacio
458         st_ -= {()}
459 
460         # Devolvemos el link de v
461         return st_ - st
462 
463     def compConexas(self):
464         """Comprobar la conexion de un complejo simplicial."""
465         # Para ello comprobamos que su 1-esqueleto sea conexo
466         k1Graph = nx.Graph()
467         k1Graph.add_nodes_from([vertice[0] for vertice in self.getCarasN(0)])
468         k1Graph.add_edges_from(self.getCarasN(1))
469         return nx.number_connected_components(k1Graph)
470 
471     def k_esqueleto(self, k):
472         """
473         Calcular el k-esqueleto de un complejo simplicial.
474 
475         k: int.
476         """
477         return set(c for c in self.getCaras() if len(c) <= k + 1)
478 
479     def drawK1(self):
480         """Representación gráfica del 1-esqueleto."""
481         k1Graph = nx.Graph()
482         k1Graph.add_nodes_from([vertice[0] for vertice in self.getCarasN(0)])
483         k1Graph.add_edges_from(self.getCarasN(1))
484         plt.figure().add_subplot(111)
485         nx.draw_networkx(k1Graph, with_labels=True)
486 
487     def caractEuler(self):
488         """Obtención de la característica de Euler."""
489         return sum([(-1)**k * len(self.getCarasN(k)) for k in range(self.dim() + 1)])
490 
491     def filtracion(self, a):
492         """
493         Obtener las caras con peso menor o igual que un valor.
494 
495         a: float.
496         """
497         i = 0
498         caras = list()
499         while i < len(self.carasOrd) and self.carasOrd[i][1] <= a:
500             caras.append(self.carasOrd[i])
501             i += 1
502 
503         result = Complejo()
504         for cara, peso in caras:
505             result.setCaras([cara], peso)
506 
```

```
507         return result
508
509     def borde(self):
510         """Funcion borde."""
511         d = self.dim()
512         return list(chain.from_iterable(combinations(s, d) for s in self.getCarasN(d)))
513
514     def matrizBorde(self, p):
515         """
516         Calculo de la matriz borde de dimensión dada.
517
518         p: int.
519         """
520         if p < 0:
521             return None
522
523         carasP = sorted(list(self.getCarasN(p)))
524
525         if p == 0:
526             m = np.zeros((1, len(carasP)), dtype=int)
527         else:
528             carasP_1 = sorted(list(self.getCarasN(p - 1)))
529             d = self.dim()
530             if p == d + 1:
531                 m = np.zeros((len(carasP_1), 1), dtype=int)
532             elif p > d:
533                 m = None
534             else:
535                 m = np.zeros((len(carasP_1), len(carasP)), dtype=int)
536                 for j in range(len(carasP)):
537                     caraP = set(carasP[j])
538                     for i in range(len(carasP_1)):
539                         m[i, j] = int(set(carasP_1[i]).issubset(caraP))
540
541         return m
542
543     def matrizBordeGeneralizada(self):
544         """Calculo de la matriz borde generalizada."""
545         caras = self.getCarasOrd()
546         caras1 = caras.copy()
547
548         m = np.zeros((len(caras), len(caras1)), dtype=int)
549
550         for j in range(len(caras)):
551             cara = set(caras[j])
552             for i in range(len(caras1)):
553                 m[i, j] = int(len(cara) - len(caras1[i]) == 1 and set(caras1[i]) != cara and
                        set(caras1[i]).issubset(cara))
554
555         return m
556
557     def algoritmoPersistencia(self):
558         """Realiza el algoritmo de persistencia sobre la matriz borde generalizada."""
559         M = self.matrizBordeGeneralizada()
560         lowsArray = [-1 for i in range(len(M))]
561
562         for j in range(len(M)):
563             lowsArray[j] = low(M[:, j])
564             # Comportamiento do-while
565             mismoLow = True
566             while mismoLow and lowsArray[j] >= 0:
567                 mismoLow = False
568                 for k in range(j-1, -1, -1):
569                     if lowsArray[k] == lowsArray[j]:
570                         sumaColZ2(M, k, j)
571                         mismoLow = True
572                         lowsArray[j] = low(M[:, j])
573                         break
574
575         return M, lowsArray
```

```
576
577     def persistencia(self):
578         """Cálculo de los puntos del diagrama de persistencia."""
579         _, lowsArray = self.algoritmoPersistencia()
580         dgm = list()
581         carasVisitadas = []
582         for i in range(0, self.dim()):
583             dgmi = list()
584             numCaras = len(self.getCarasN(i))
585             j = 0
586             while j < len(lowsArray) and numCaras > 0:
587                 if j not in carasVisitadas and lowsArray[j] >= 0 and len(self.carasOrd[
                        lowsArray[j]][0]) == i+1:
588                     dgmi.append((self.carasOrd[lowsArray[j]][1], self.carasOrd[j][1]))
589                     numCaras = numCaras - 1
590                     carasVisitadas.append(j)
591                     # Marca de que ya se ha muerto su clase de equivalencia
592                     lowsArray[lowsArray[j]] = -2
593
594                 j = j + 1
595
596             j = 0
597             while j < len(lowsArray) and numCaras > 0:
598                 if j not in carasVisitadas and lowsArray[j] == -1 and len(self.carasOrd[j][0])
                         == i+1:
599                     dgmi.append((self.carasOrd[j][1], float('inf')))
600                     numCaras = numCaras - 1
601                     carasVisitadas.append(j)
602                     # Marca de que ya se ha anadido su persistencia
603                     lowsArray[j] = -2
604                 j = j + 1
605
606             dgm.append(dgmi)
607
608         return dgm
609
610     def diagramaPersistencia(self):
611         """Representación del diagrama de persistencia."""
612         dmg = self.persistencia()
613         fig, ax = plt.subplots(dpi=300)
614         maxDeath = -1
615         infinity = list()
616         birth = list()
617         death = list()
618         for i in range(len(dmg)):
619             dmgi = dmg[i]
620             birthI = np.array([c[0] for c in dmgi if c[1] != float('inf')])
621             deathI = np.array([c[1] for c in dmgi if c[1] != float('inf')])
622             infinity.append([c[0] for c in dmgi if c[1] == float('inf')])
623             maxDeath = max(maxDeath, int(np.amax(deathI))*1.1 + 1)
624             birth.append(birthI)
625             death.append(deathI)
626
627         for i in range(len(infinity)):
628             if infinity[i] != []:
629                 birth[i] = np.append(birth[i], np.array(infinity[i]))
630                 death[i] = np.append(death[i], np.array([maxDeath for j in range(len(infinity[
                        i]))]))
631
632             ax.scatter(x=birth[i], y=death[i], alpha=0.90, label=r"$H_{}$".format(i), zorder
                    =10)
633
634         lims = [
635                 np.min([ax.get_xlim(), ax.get_ylim()]),  # min of both axes
636                 np.max([ax.get_xlim(), ax.get_ylim()]),  # max of both axes
637                 ]
638         ax.set_xlabel("Birth Time")
639         ax.set_ylabel("Death Time")
640         ax.plot([lims[0], lims[1]], [lims[0], lims[1]], "--",  color=(0.3, 0.3, 0.3), zorder
                =0)
```

```
641         ax.plot([lims[0], lims[1]], [maxDeath, maxDeath], "k--", label=r"$\infty$", zorder=0)
642         ax.legend()
643         ax.set_xlim(lims)
644         ax.set_ylim(ymin=lims[0])
645 
646         if not os.path.exists("persistencia/"):
647             os.makedirs("persistencia/")
648 
649         fig.savefig("persistencia/perDiag.png", dpi=300)
650 
651     def codigoBarrasPers(self):
652         """Representación de la persistencia en formato de código de barras."""
653         dmg = self.persistencia()
654         fig, ax = plt.subplots(nrows=len(dmg), sharex=True, dpi=300)
655         ax = ax[::-1]
656         maxDeath = -1
657         infinity = list()
658         birth = list()
659         death = list()
660         for i in range(len(dmg)):
661             dmgi = dmg[i]
662             birthI = np.array([c[0] for c in dmgi if c[1] != float('inf')])
663             deathI = np.array([c[1] for c in dmgi if c[1] != float('inf')])
664             infinity.append([c[0] for c in dmgi if c[1] == float('inf')])
665             maxDeath = max(maxDeath, int(np.amax(deathI))*1.1 + 1)
666             birth.append(birthI)
667             death.append(deathI)
668 
669         for i in range(len(infinity)):
670             if infinity[i] != []:
671                 birth[i] = np.append(birth[i], np.array(infinity[i]))
672                 death[i] = np.append(death[i], np.array([maxDeath for j in range(len(infinity[
                        i]))]))
673 
674             # Elimina las parejas que nacen y mueren a la vez
675             n = 0
676             while n < len(birth[i]):
677                 if birth[i][n] == death[i][n]:
678                     birth[i] = np.delete(birth[i], n)
679                     death[i] = np.delete(death[i], n)
680                     n = n-1
681                 n = n+1
682 
683             diff = death[i] - birth[i]
684             # diff[diff<=0] = 0.005
685             ax[i].barh(y=np.arange(len(birth[i])),
686                        width=diff,
687                        height=0.2,
688                        align="center",
689                        left=birth[i],
690                        label=r"$H_{}$".format(i),
691                        color=f"C{i}",
692                        linewidth=0)
693 
694             ax[i].get_yaxis().set_ticks([])
695             ax[i].set_ylabel(r"$H_{}$".format(i), rotation="horizontal")
696             ax[i].get_yaxis().set_label_coords(-0.035, 0.5)
697 
698         if not os.path.exists("persistencia/"):
699             os.makedirs("persistencia/")
700 
701         fig.savefig("persistencia/perBarras.png", dpi=300)
702 
703     def betti(self, p, incremental=False):
704         """
705         Calculo del número de p de Betti.
706 
707         p: int.
708         """
709         if incremental and self.dim() == 2:
```

```
710             # Algoritmo incremental
711             b = self.allBettis(incremental=True)[p]
712 
713         else:
714             if p == 0:
715                 Zp = len(self.getCarasN(0))
716             else:
717                 Mp = normSmithZ2(self.matrizBorde(p))
718                 Zp = Mp.shape[1] - matrix_rank(Mp)
719 
720             Bp = matrix_rank(normSmithZ2(self.matrizBorde(p + 1)))
721 
722             b = Zp - Bp
723 
724             self.bettiNums[p] = b
725 
726         return b
727 
728     def allBettis(self, incremental=False):
729         """Calculo de todos los números de Betti."""
730         if incremental and self.dim() == 2:
731             # Puede que el resultado sea erroneo si el complejo no está contenido en R2
732             # Algoritmo incremental
733             k1Graph = nx.Graph()
734             nodos = self.getCarasN(0)
735             k1Graph.add_nodes_from([vertice[0] for vertice in nodos])
736 
737             self.bettiNums[0] = len(nodos)
738             self.bettiNums[1] = 0
739             numCompConexas = self.bettiNums[0]
740 
741             for arista in self.getCarasN(1):
742                 k1Graph.add_edge(*arista)
743                 newNumCompConexas = nx.number_connected_components(k1Graph)
744                 if nx.number_connected_components(k1Graph) < numCompConexas:
745                     numCompConexas = newNumCompConexas
746                     self.bettiNums[0] -= 1
747                 else:
748                     self.bettiNums[1] += 1
749 
750             self.bettiNums[1] -= len(self.getCarasN(2))
751             self.bettiNums[2] = 0
752 
753         elif -1 in self.bettiNums:
754             # Calculo con las matrices borde
755             Zps = np.array([len(self.getCarasN(0))], dtype=int)
756             Bps = np.array([], dtype=int)
757             d = self.dim()
758             for p in range(1, d + 2):
759                 Mp = normSmithZ2(self.matrizBorde(p))
760 
761                 if p <= d:
762                     Zps = np.append(Zps, Mp.shape[1] - matrix_rank(Mp))
763 
764                 Bps = np.append(Bps, matrix_rank(Mp))
765 
766             self.bettiNums = list(Zps - Bps)
767 
768         return self.bettiNums
769 
770     def __str__(self):
771         """El toString del complejo."""
772         return "Caras: " + str(self.caras)
773 
774 
775 if __name__ == "__main__":
776     """
777     comp1 = Complejo([(0, 1, 2, 3)])
778     print("-------------COMP1-------------")
779     analisisComplejo(comp1, set((0, 1)))
```

```
780
781     comp2 = Complejo(list(comp1.k_esqueleto(2)))
782     print("\n-------------COMP2-------------")
783     analisisComplejo(comp2, set((0,)))
784
785     comp3 = Complejo([(0, 1), (1, 2, 3, 4), (4, 5), (5, 6), (4, 6), (6, 7, 8), (8, 9)])
786     print("\n-------------COMP3-------------")
787     analisisComplejo(comp3, set((4,)))
788
789     comp4 = Complejo(list(comp3.k_esqueleto(1)))
790     print("\n-------------COMP4-------------")
791     analisisComplejo(comp4, set((4,)))
792
793     comp5 = Complejo([(0, 1, 2), (2, 3), (3, 4)])
794     print("\n-------------COMP5-------------")
795     analisisComplejo(comp5, set((2,)))
796
797     comp6 = Complejo([(1, 2, 4), (1, 3, 6), (1, 4, 6), (2, 3, 5), (2, 4, 5), (3, 5, 6)])
798     print("\n-------------COMP6-------------")
799     analisisComplejo(comp6, set((1, 4)))
800
801     comp7 = Complejo(list(comp6.k_esqueleto(1)))
802     print("\n-------------COMP7-------------")
803     analisisComplejo(comp7, set((1, 4)))
804
805     comp8 = Complejo([(1, 2, 4), (2, 4, 5), (2, 3, 5), (3, 5, 6), (1, 3, 6), (1, 4, 6),
806                       (4, 5, 7), (5, 7, 8), (5, 6, 8), (6, 8, 9), (4, 6, 9), (4, 7, 9),
807                       (1, 7, 8), (1, 2, 8), (2, 8, 9), (2, 3, 9), (3, 7, 9), (1, 3, 7)])
808     print("\n-------------COMP8-------------")
809     analisisComplejo(comp8, set((1,)))
810
811     comp9 = Complejo(list(comp8.k_esqueleto(1)))
812     print("\n-------------COMP9-------------")
813     analisisComplejo(comp9, set((1,)))
814
815     comp10 = Complejo([(1, 2, 6), (2, 3, 4), (1, 3, 4), (1, 2, 5), (2, 3, 5), (1, 3, 6),
816                        (2, 4, 6), (1, 4, 5), (3, 5, 6), (4, 5, 6)])
817     print("\n-------------COMP10-------------")
818     analisisComplejo(comp10, set((1,)))
819
820     comp11 = Complejo(list(comp10.k_esqueleto(1)))
821     print("\n-------------COMP11-------------")
822     analisisComplejo(comp11, set((1,)))
823
824     comp12 = Complejo([(0,), (1,), (2, 3), (4, 5), (5, 6), (4, 6), (6, 7, 8, 9)])
825     print("\n-------------COMP12-------------")
826     analisisComplejo(comp12, set((6,)))
827
828     # Ejemplo filtracion
829     print("\n-------------COMP13-------------")
830     comp13 = Complejo()
831     comp13.setCaras([(0, 1)], 1.0)
832
833     comp13.setCaras([(1, 2), (2, 3), (2, 4)], 2.0)
834     comp13.setCaras([(3, 4)], 3.0)
835     comp13.setCaras([(2, 3, 4)], 4.0)
836
837     # Todas las caras del complejo
838     print(f"Todas las caras: {comp13.getCaras()}")
839
840     # Umbral
841     print(f"Umbral de {{3}}: {comp13.umbral((3,))}")
842
843     # Filtraciones
844     K1 = comp13.filtracion(1.0)
845     K2 = comp13.filtracion(2.0)
846     K3 = comp13.filtracion(3.0)
847     K4 = comp13.filtracion(4.0)
848
849     # Todas las caras de las filtraciones
```

```
850     print(f"Todas las caras de K1: {K1.getCaras()}")
851     print(f"Todas las caras de K2: {K2.getCaras()}")
852     print(f"Todas las caras de K3: {K3.getCaras()}")
853     print(f"Todas las caras de K4: {K4.getCaras()}")
854
855     # Caras ordenedas por filtracion
856     print(f"Caras ordenadas segun las filtraciones: {comp13.getCarasOrd()}")
857
858
859
860
861     points = np.array([(0.38021546727456423, 0.46419202339598786),
862                        (0.7951628297672293, 0.49263630135869474),
863                        (0.566623772375203, 0.038325621649018426),
864                        (0.3369306814864865, 0.7103735061134965),
865                        (0.08272837815822842, 0.2263273314352896),
866                        (0.5180166301873989, 0.6271769943824689),
867                        (0.33691411899985035, 0.8402045183219995),
868                        (0.33244488399729255, 0.4524636520475205),
869                        (0.11778991601260325, 0.6657734204021165),
870                        (0.9384303415747769, 0.2313873874340855)])
871
872     points = np.array([[0.8957641450573793, 0.2950833519989374],
873                        [0.028621391963087994, 0.9440875759025237],
874                        [0.517621505875702, 0.1236620161847416],
875                        [0.7871047164191424, 0.7777474116014623],
876                        [0.21869796914805273, 0.7233589914276723],
877                        [0.9891035292480995, 0.6032186214942837],
878                        [0.30113764052453484, 0.613321425324272],
879                        [0.18407448222466916, 0.7868606964403773],
880                        [0.4496777667376678, 0.874366215574117],
881                        [0.08225571534539433, 0.616710205071694]])
882
883     curva1 = [4 * sy.sin(t), 9 * sy.cos(t)]
884     points = puntosCurvaRuido(curva1, t, 0, 2*np.pi, numPuntos=30)
885
886     curva2 = [1 + 3 * t**2, t**3 - 2 * t]
887
888     points = puntosCurvaRuido(curva2, t, -2, 2, numPuntos=30)
889
890     print(points)
891
892     plt.plot(points[:, 0], points[:, 1], 'ko')
893     plt.show()
894
895     vor = drawVor(points)
896
897     delaunay(points)
898
899     alpha = alfaComplejo(points)
900     print(alpha)
901     i = 0
902     images = []
903     for valor in alpha.umbrales():
904         # print(valor)
905         K = alpha.filtracion(valor)
906         fig = voronoi_plot_2d(vor, show_vertices=False, line_width=2, line_colors='blue',
                lines_alpha=0.6)
907         plotalpha(points, K)
908         plt.title(r"$r={}$".format(str(valor)))
909         fig.savefig(f"imgTemp/im{i}.png")
910         images.append(imageio.imread(f"imgTemp/im{i}.png"))
911         i += 1
912         plt.show()
913
914     imageio.mimsave('alphaGif/alpha.gif', images)
915
916
917     """
918     comp1 = Complejo([(0, 1, 2, 3)])
```

```
919     print(f"Los num de Betti del tetraedro son: {comp1.allBettis()}")
920     print(f"Los num de Betti del borde del tetraedro son: {Complejo(comp1.borde()).allBettis()
            }")
921
922     toro1 = Complejo([(1, 7, 8), (1, 2, 8), (2, 8, 9), (2, 3, 9), (3, 9, 7), (3, 1, 7),
923                       (4, 1, 2), (4, 5, 2), (5, 2, 3), (5, 6, 3), (6, 3, 1), (6, 4, 1),
924                       (7, 4, 5), (7, 8, 5), (8, 5, 6), (8, 9, 6), (9, 6, 4), (9, 7, 4)])
925     print(f"Los num de Betti del toro son: {toro1.allBettis()}")
926     toro2 = Complejo([(1, 7, 3), (3, 4, 6), (6, 4, 7), (1, 2, 3), (2, 3, 6), (6, 7, 1),
927                       (2, 5, 6), (5, 6, 1), (7, 2, 5), (7, 3, 5), (3, 4, 5), (5, 4, 1),
928                       (1, 4, 2), (2, 4, 7)])
929     print(f"Los num de Betti del toro con triang minimal son: {toro2.allBettis()}")
930
931     klein = Complejo([(1, 7, 8), (1, 2, 8), (2, 8, 9), (2, 3, 9), (3, 9, 7), (3, 4, 7),
932                       (1, 4, 2), (4, 2, 5), (2, 3, 5), (3, 5, 6), (3, 4, 6), (1, 4, 6),
933                       (4, 5, 7), (7, 5, 8), (5, 6, 8), (6, 8, 9), (6, 1, 9), (1, 9, 7)])
934     print(f"Los num de Betti de la botella de Klein son: {klein.allBettis()}")
935
936     anillo = Complejo([(0, 1, 3), (1, 3, 4), (1, 2, 4), (2, 4, 5), (0, 2, 5), (0, 3, 5)])
937     print(f"Los num de Betti del anillo son: {anillo.allBettis()}")
938     print(f"Los num de Betti del anillo son (algoritmo incremental): {anillo.allBettis(
            incremental=True)}")
939
940     planoProy = Complejo([(1, 2, 10), (2, 3, 10), (3, 9, 10), (3, 4, 9), (4, 8, 9), (4, 5, 8),
941                           (2, 3, 5), (3, 5, 6), (3, 6, 4), (4, 6, 7), (4, 5, 7), (2, 5, 7),
942                           (5, 6, 8), (6, 8, 9), (6, 7, 9), (7, 9, 10), (2, 7, 10), (1, 2, 10)
                                ])
943     print(f"Los num de Betti del plano proyectivo son: {planoProy.allBettis()}")
944
945     asno = Complejo([(1, 3, 5), (1, 5, 6), (1, 3, 6), (2, 3, 5), (2, 4, 5), (4, 5, 6),
946                      (3, 6, 7), (2, 3, 7), (6, 7, 8), (6, 4, 8), (1, 2, 4), (1, 3, 4),
947                      (3, 4, 8), (2, 3, 8), (1, 2, 8), (1, 7, 8), (1, 2, 7)])
948     print(f"Los num de Betti del sombrero del asno son: {asno.allBettis()}")
949     print(f"Los num de Betti del sombrero del asno son (algoritmo incremental): {asno.
            allBettis(incremental=True)}")
950
951     dobeToro = Complejo([(1, 9, 7), (1, 7, 3), (1, 4, 3), (4, 6, 3), (6, 3, 5),
952                          (6, 8, 5), (8, 5, 7), (8, 10, 7), (10, 7, 9), (7, 3, 11),
953                          (11, 3, 9), (3, 5, 9), (5, 9, 1), (1, 5, 11), (5, 7, 11),
954                          (10, 9, 0), (0, 9, 11), (0, 11, 2), (2, 11, 1), (1, 2, 4),
955                          (2, 10, 4), (10, 8, 4), (2, 6, 10), (2, 6, 8), (2, 0, 8),
956                          (0, 4, 8), (0, 4, 6), (0, 6, 10)])
957     print(f"Los num de Betti del doble toro son: {dobeToro.allBettis()}")
958
959     comp3 = Complejo([(0, 1), (1, 2, 3, 4), (4, 5), (5, 6), (4, 6), (6, 7, 8), (8, 9)])
960     print(f"Los num de Betti del siguiente complejo son: {comp3.allBettis()}")
961     """
962     curva1 = [4 * sy.sin(t), 9 * sy.cos(t)]
963     points = puntosCurvaRuido(curva1, t, 0, 2*np.pi, numPuntos=30)
964
965     alpha = alfaComplejo(points)
966     K = alpha.filtracion(3.6)
967     print(f"Los num de Betti del siguiente alpha complejo son: {K.allBettis()}")
968     print(f"Los num de Betti del siguiente alpha complejo son (algoritmo incremental): {K.
            allBettis(incremental=True)}")
969
970
971     points = np.array([(-2, 2),
972                        (1.5, 2.2),
973                        (2.5, -0.5),
974                        (-1.4, -0.7),
975                        (1.2, -1.87)])
976     alpha = alfaComplejo(points)
977
978     vor = drawVor(points)
979
980     i = 0
981     images = []
982     if not os.path.exists("imgTemp/"):
983         os.makedirs("imgTemp/")
```

```
984
985     for valor in alpha.umbrales():
986         K = alpha.filtracion(valor)
987         fig = voronoi_plot_2d(vor, show_vertices=False, line_width=2, line_colors='blue',
                lines_alpha=0.6)
988         plotalpha(points, K)
989         plt.title(r"$r={}$".format(str(valor)))
990         fig.savefig(f"imgTemp/im{i}.png", dpi=300)
991         images.append(imageio.imread(f"imgTemp/im{i}.png"))
992         i += 1
993         plt.show()
994
995     if not os.path.exists("alphaGif/"):
996         os.makedirs("alphaGif/")
997     imageio.mimsave('alphaGif/alpha.gif', images)
998     """
```