

# Memoria de seguimiento

Alejandro García Castellanos

z17m008

30 de abril de 2021

## 1. Resumen del trabajo realizado

Como comenté en las tareas del plan de trabajo comencé repasando los contenidos de la asignatura de Topología Computacional. Una vez terminado, estudié el siguiente artículo: *Cohen-Steiner, David & Edelsbrunner, Herbert & Harer, John. (2005). Stability of Persistence Diagrams. Discrete & Computational Geometry - DCG. 37. 263-271. 10.1007/s00454-006-1276-5*. Profundizando en las cuestiones y pasos de la demostración que más dificultad me suponían.

Tras comprender el resultado y la demostración del teorema, escribí la sección de conocimientos previos necesarios para hacer autocontenida el trabajo. Para ello he hecho uso de libros, artículos y conferencias para tomar sus explicaciones y definiciones como referencia. Por último he introducido el teorema de estabilidad tanto para la distancia Hausdorff como para la distancia Bottleneck, con el fin de poner en contexto los algoritmos necesarios para la implementación.

Para la sección de implementación, hice una búsqueda de los diversos métodos de calcular las distancias y escribí su funcionamiento explicado en la memoria. Además he estado investigando sobre el funcionamiento de diversas librerías de Python que me pueden ser de utilidad a la hora de implementar los algoritmos propuestos.

## 2. Modificaciones al Plan de Trabajo

### 2.1. Revisión de la lista de objetivos del trabajo

Se ha añadido la implementación de la distancia Hausdorff a los objetivos del trabajo, ya que la demostración propuesta en el artículo previamente mencionado, demuestra primero la estabilidad para esta distancia. Por tanto es conveniente implementarla para ilustrar este resultado.

Luego, la lista de objetivos del trabajo es:

- Buscar referencias que contengan el enunciado y la demostración del Teorema de estabilidad.
- Estudiar y entender estas referencias.
- Implementar el cálculo de las distancias de Bottleneck y Hausdorff entre diagramas de persistencia.
- Ilustrar el Teorema de estabilidad sobre distintos conjuntos de datos.
- Redactar la memoria y preparar la presentación

## 2.2. Revisión de la lista de tareas

La lista de tareas se ve modifica en dos puntos. El primero en el hecho de añadir la implementación de la distancia Hausdorff y el segundo en la redacción de los algoritmos para el cálculo de las distancias, la cual no se había considerado en el Plan de Trabajo original.

Es por ello que la lista tareas para la realización del trabajo es:

### ■ Preparación

- Buscar referencias que contengan el enunciado y la demostración del Teorema de estabilidad.
- Repasar los contenidos de la asignatura de Topología Computacional.

### ■ Realización de los objetivos principales

- Estudiar y entender estas referencias.
- Implementar el cálculo de las distancias Bottleneck y Hausdorff entre diagramas de persistencia
  1. Búsqueda de los algoritmos.
  2. Implementación sobre clase de complejos simpliciales programada en la asignatura de Topología Computacional.
  3. Testing de las nuevas funcionalidades.

### ■ Elaboración de la Memoria y Presentación

#### ● Memoria

- Redactar conocimientos previos requeridos para la comprensión del teorema.
- Redactar Teorema de estabilidad y su demostración.

- Redactar algoritmos para el cálculo de las distancias.
- Ilustrar el teorema a través de su implementación.
- Elaboración y preparación de la presentación.

### **3. Revisión del Diagrama de Gantt**

Debido a un error de estimación del tiempo requerido para la realización de la parte de “Conocimientos previos”, se ve desplazada la planificación de las tareas: “Redactar Teorema y demostración”, “Implementar distancias” e “Ilustrar conocimientos”. Y se ha incorporado en el Diagrama de Gantt la tarea “Redactar algoritmos”.

Además, no se tuvo en consideración el tiempo de estudio de los algoritmos de las distancias en la planificación de la tarea “Estudiar y entender las referencias”.

## TFG replanificación

### Preparación

- Buscar referencias
- Repasar Topología Computacional

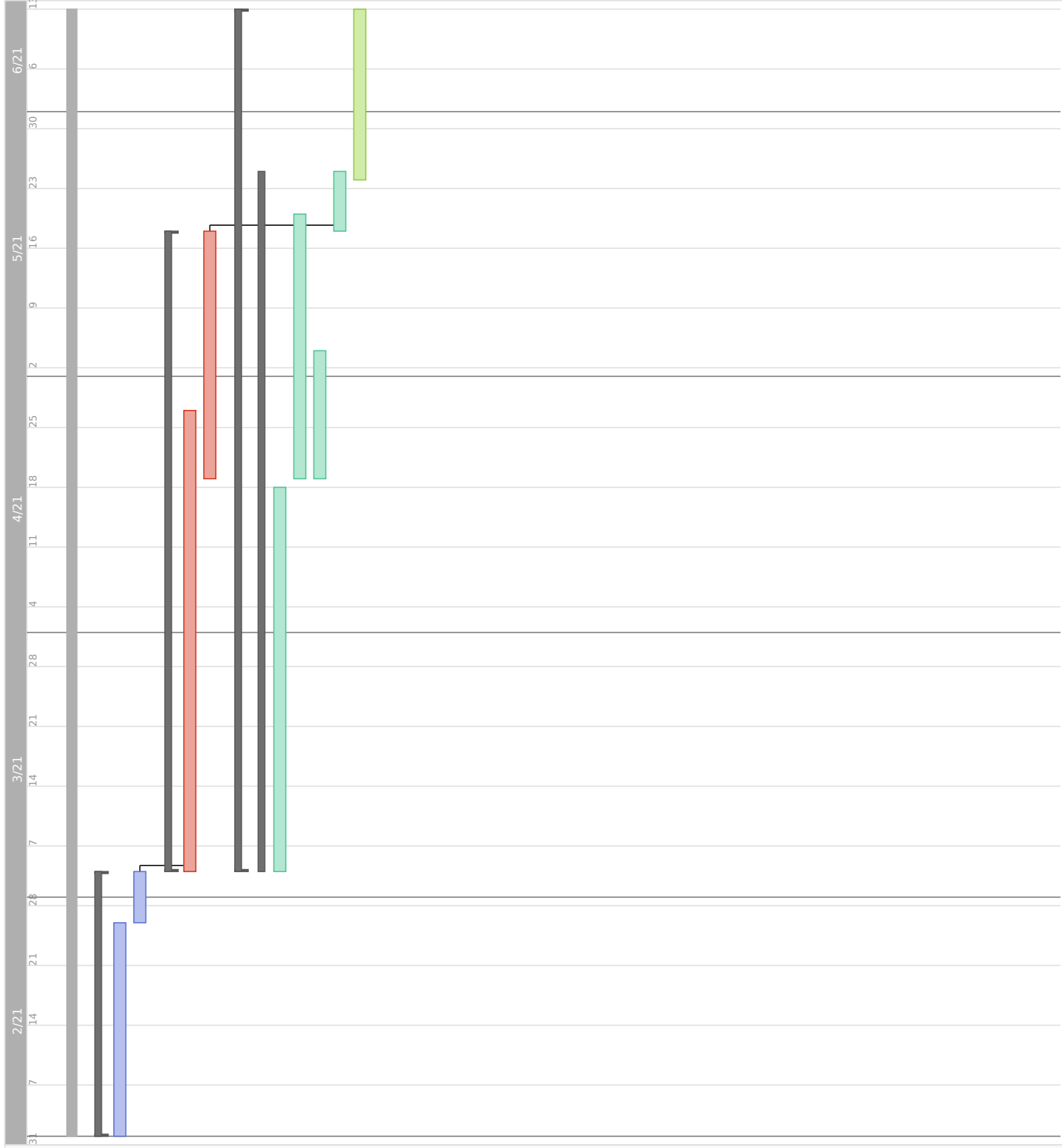
### Realización de los objetivos

- Estudiar y entender las referencias
- Implementar distancias

### Memoria y presentación

#### Memoria

- Redactar conocimientos previos
- Redactar Teorema y dem
- Redactar algoritmos
- Ilustrar con ej.
- Presentación





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

<b>1. Introducción</b>	<b>1</b>
1.1. Motivación . . . . .	1
1.2. Descripción general del trabajo . . . . .	1
1.3. Estructura del trabajo . . . . .	2
<b>2. Desarrollo</b>	<b>3</b>
2.1. Conocimientos previos y definiciones . . . . .	3
2.1.1. Complejos Simpliciales . . . . .	3
2.1.2. Complejos simpliciales de nubes de puntos . . . . .	9
2.1.3. Homología . . . . .	13
2.1.4. Persistencia . . . . .	19
2.2. Teorema de estabilidad . . . . .	27
2.2.1. Proposición del teorema . . . . .	27
2.2.2. Estabilidad para la distancia Hausdorff . . . . .	28
2.2.3. Estabilidad para la distancia bottleneck . . . . .	28
2.3. Implementaciones y cálculos . . . . .	29
2.3.1. Cálculo de la distancia Hausdorff . . . . .	29
2.3.2. Cálculo de la distancia bottleneck . . . . .	30
2.3.3. Pruebas . . . . .	34
<b>3. Resultados y conclusiones</b>	<b>35</b>
<b>4. Análisis de impacto</b>	<b>37</b>
<b>Bibliografía</b>	<b>39</b>
<b>Anexo</b>	<b>40</b>



# 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 resolver estas preguntas se suele hacer uso de la *homología persistente*, y más concretamente en su representación en los denominados *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, el cual, a grandes rasgos, establece que pequeñas perturbaciones en los datos implican pequeñas perturbaciones en la homología persistente.

Para ello he buscado y estudiado diversas referencias que contengan el enunciado y la demostración del Teorema de estabilidad, como [2] [3] [4].

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

Las pruebas se sustentan en el *pipeline* usual en el 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 secuencia de espacios topológicos sobre los obtendremos la homología persistente que será representada en el diagrama

de persistencia. Para poder observar la estabilidad, se introducirán perturbaciones en conjunto de puntos inicial y se contrastará los diagramas de persistencia.

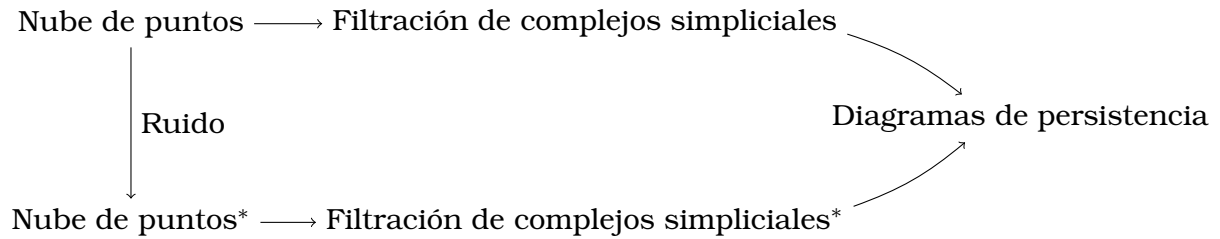


Figura 1.1: Pipeline para la comprobación del Teorema de estabilidad

### 1.3. Estructura del trabajo

En la sección 2.1 se introducirán las nociones topológicas básicas para hacer auto-contenido 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  $\{u_0, u_1, \dots, u_k\}$  de  $\mathbb{R}^d$  es *afinmente independiente* si los vectores  $\{\overrightarrow{u_0 u_1}, \dots, \overrightarrow{u_0 u_k}\}$  son linealmente independientes.

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

**Definición 2.1.3.** Llamaremos *envolvente convexa* de  $u_0, u_1, \dots, u_k$ , denotado por  $\text{conv}\{u_0, u_1, \dots, u_k\}$ , 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*  $\sigma$  en  $\mathbb{R}^d$  con  $d \geq k$  es la envolvente convexa de  $k + 1$  puntos afínmente independientes  $u_0, u_1, \dots, u_k \in \mathbb{R}^d$ , es decir,  $\sigma := \text{conv}\{u_0, u_1, \dots, u_k\}$ .

## 2.1. Conocimientos previos y definiciones

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

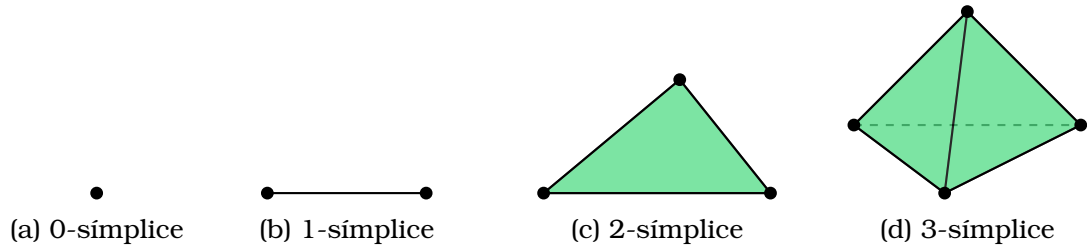


Figura 2.1: Representación de los símlices de dimensión 0, 1, 2 y 3

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

Haciendo uso de la definición de caras de un símplex  $\sigma$  podemos definir el borde y el interior de  $\sigma$ .

**Definición 2.1.5.** Sea  $\sigma$  un símplex. Entonces

- Se define el borde de  $\sigma$  como

$$\text{bd } \sigma = \bigcup_{\tau < \sigma} \tau.$$

- Se define el interior de  $\sigma$  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  $\sigma$  si y sólo si todos sus coeficientes  $\lambda_i$  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  $\lambda_i$  positivos.

Una vez que ya conocemos las piezas de nuestra descomposición vamos a ver como tenemos que unirlos 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 símlices con otros por sus caras.

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

- Si  $\sigma \in K$  y  $\tau \leq \sigma$  entonces  $\tau \in K$ .
- 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  $i = 0, 1$ .

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



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.

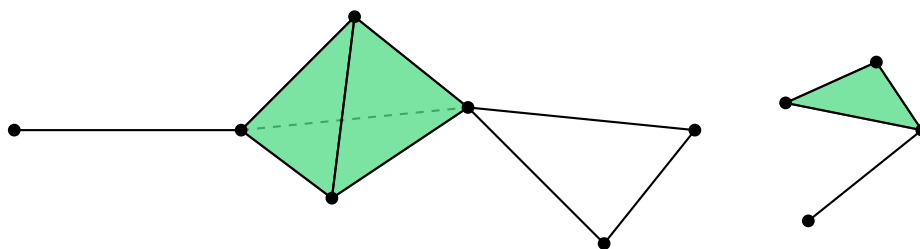


Figura 2.2: Ejemplo de complejo simplicial

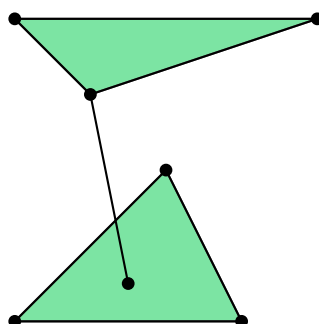


Figura 2.3: Ejemplo de conjunto de símlices que no cumplen las condiciones de complejo simplicial

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

Como se puede observar, el espacio subyacente de un complejo simplicial es compacto, siendo unión finita de símlices. 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  $h : X \rightarrow |K|$  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 símlices que nos será de gran ayuda más adelante es la *estrella de un símlice*  $\tau$ , la cual consiste de las cocaras de  $\tau$ , denotado por  $\text{St } \tau$ . 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  $\text{St } \tau$ . Adicionalmente, se define el *link* de  $\tau$  como:  $\text{Lk } \tau = \{v \in \overline{\text{St}} \tau \mid v \cap \tau = \emptyset\}$ .

### 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 *símlices abstractos*.
- La *dimensión* de un símlice abstracto  $\alpha \in A$  es  $\dim \alpha = \text{card}(\alpha) - 1$ . Y la dimensión del complejo es el máximo de las dimensiones de sus símlices.
- Una *cara* de  $\alpha \in A$  es cualquier subconjunto no vacío de  $\beta \subset \alpha$ .
- El *conjunto de vértices* de  $A$ , denotado por  $\text{Vert } A$ , es la unión de todos sus símlices.
- 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:  $\text{Vert } A = \{0, 1, 2, 3, 4, 5, 6\}$ .

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

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

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 símlices 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  $u_0, u_1, \dots, u_n$ , pertenece al interior de uno de los símlices de  $K$ . Si  $\sigma = \text{conv}\{u_0, u_1, \dots, u_k\}$  es dicho símplex, entonces  $x = \sum_{i=0}^k b_i(x)u_i$ , 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  $\varphi : \text{Vert } K \rightarrow \text{Vert } L$  una aplicación. Diremos que  $\varphi$  es una *aplicación de vértices* si satisface que para cada  $\sigma \in K$  su imagen  $\varphi(\sigma) \in L$ .

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

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

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

### Subdivisiones

Veremos que hay ocasiones que nos interesará controlar el tamaño de los símlices 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 símplex de  $L$  está contenido en un símplex de  $K$ .

## 2.1. Conocimientos previos y definiciones

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 símplece y el *cono* de un símplece de vértice  $v$ .

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

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

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

**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 símplexes de la forma  $b_\sigma * \tau$ , donde  $\sigma$  es un  $j$ -símplece y  $\tau$  es cualquier símplece de  $\text{Sd}K^{(j-1)}$  contenido en una cara de  $\sigma$ .

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

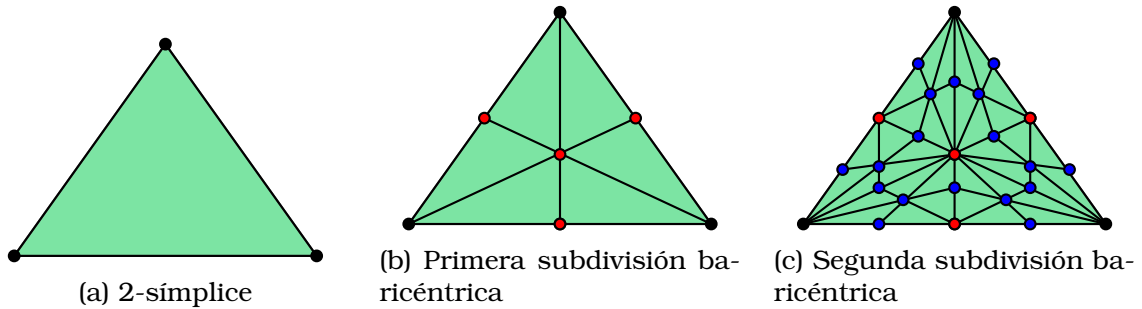


Figura 2.4: Primera y segunda subdivisión baricéntrica de un 2-símplece

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  $\sigma$  es un  $k$ -símplece, entonces el diámetro de cada símplece en la subdivisión baricéntrica de  $\sigma$  es como máximo  $\frac{k}{k+1} \text{diam } \sigma$ .

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

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

## 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

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

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

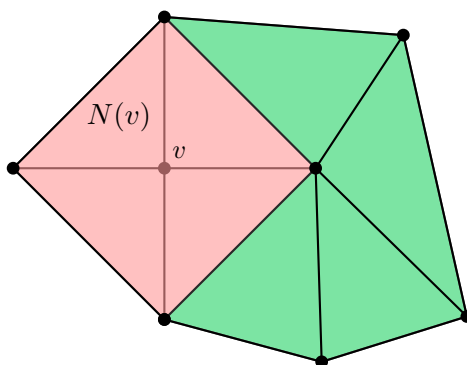


Figura 2.5: Entorno estrellado de  $v$  marcado en color rojo

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

**Definición 2.1.20.** Sean  $K$  y  $L$  complejos simpliciales,  $g : |K| \rightarrow |L|$  una aplicación continua y  $f : K \rightarrow L$  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,  $g : |K| \rightarrow |L|$  una aplicación continua que satisface la condición de estrella. Entonces  $g$  tiene una aproximación simplicial  $f : K \rightarrow L$ .

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,  $g : |K| \rightarrow |L|$  una aplicación continua. Entonces existe  $n \in \mathbb{N}$  tal que  $g$  tiene una aproximación simplicial  $f : \text{Sd}^n K \rightarrow L$ .

### 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.

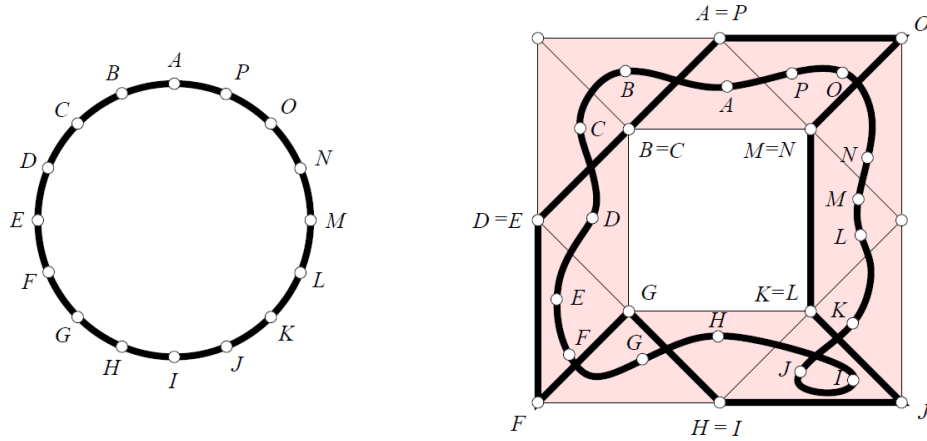


Figura 2.6: Aplicación continua del círculo en una corona circular y una aproximación simplicial de dicha aplicación. Fuente: [2]

### 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  $\mathbb{R}^d$ .

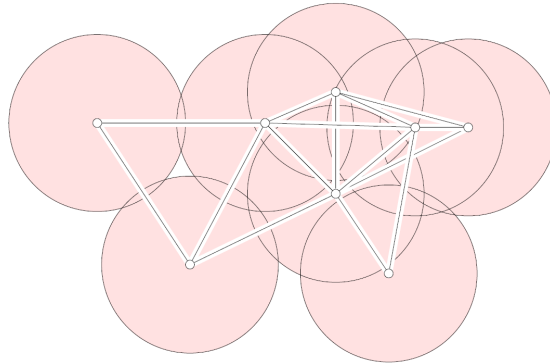


Figura 2.7: Complejo de Čech para un conjunto de nueve puntos y un radio  $r$ . Fuente: [2]

**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

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

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,  $\check{Cech}(r)$  es un símple de dimensión  $\text{card}(S) - 1$ , 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  $\text{diam } \sigma$  denota el diámetro del subconjunto  $\sigma$ .

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

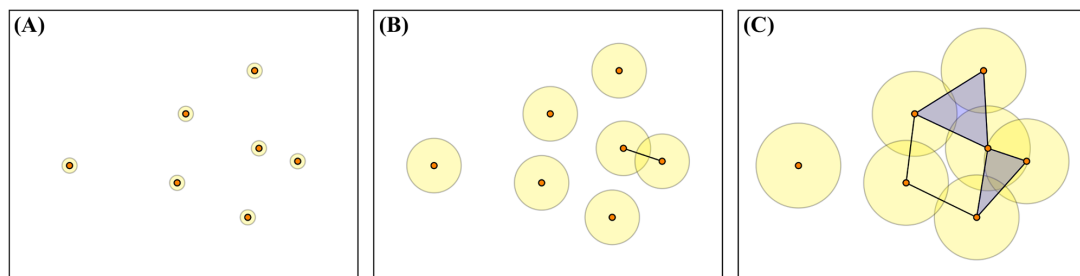


Figura 2.8: Complejos de Vietoris-Rips para un conjunto de siete puntos a medida que aumentamos el radio de izquierda a derecha. Fuente: [5]

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  $\text{VR}(r)$ . Dicho de otro modo,  $\text{VR}(r)$  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,

$$\check{Cech}(r) \subset \text{VR}(r) \subset \check{Cech}(\sqrt{2}r).$$

### Complejo de Delaunay

En esta sección introduciremos construcciones geométricas que nos limitarán la dimensión de los símplexes 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  $k = 2, 3, \dots, d + 1$ .

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 Delaunay en  $\mathbb{R}^d$  considerando envolventes convexas de los símplexes abstractos. Esta realización geométrica se denomina *triangulación de Delaunay*.

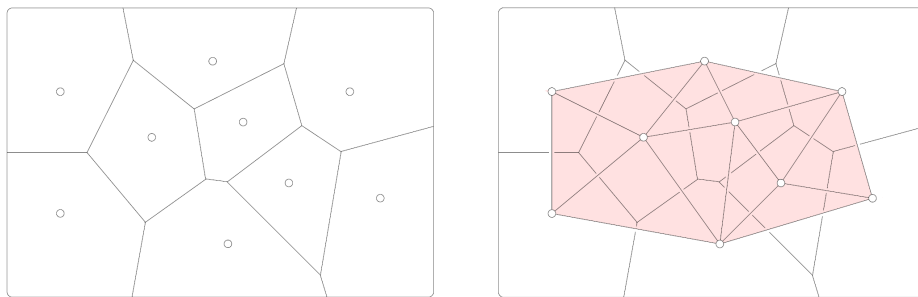


Figura 2.9: A la izquierda tenemos el Diagrama de Voronoi de un conjunto de nueve puntos en el plano, y a la derecha triangulación de Delaunay superpuesta al diagrama de Voronoi. Fuente: [2]

### 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  $R_u(r)$ .

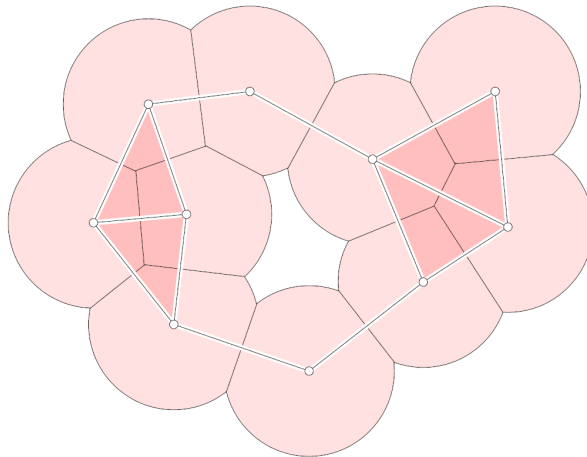


Figura 2.10: Unión de las regiones  $R_u(r)$  asociadas a un radio  $r$  y un conjunto finito de puntos  $S$ . El correspondiente alfa complejo es superpuesto a esta unión de regiones. Fuente: [2]

Puesto que  $R_u(r) \subset \overline{B}_r(u)$  para cada  $u \in S$ , se tiene que  $\text{Alpha}(r) \subset \check{\text{Cech}}(r)$ . 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 simplices 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  $p$ -simplices en  $K$ . Más concretamente,  $c$  es una  $p$ -cadena en

$K$  si

$$c = \sum a_i \sigma_i$$

con  $\sigma_i$  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,  $\sigma = [u_0, u_1, \dots, u_p]$ .

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

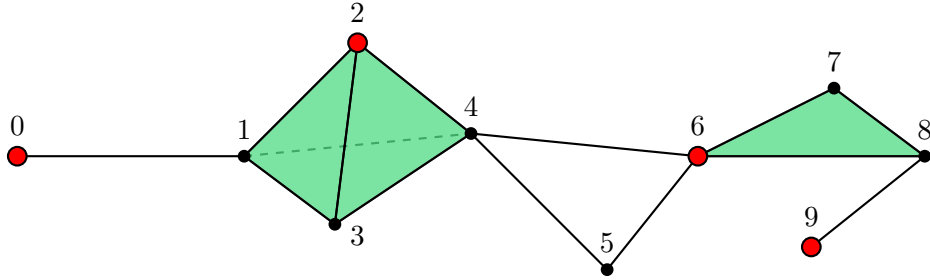


Figura 2.11: Ejemplo de 0-cadena

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

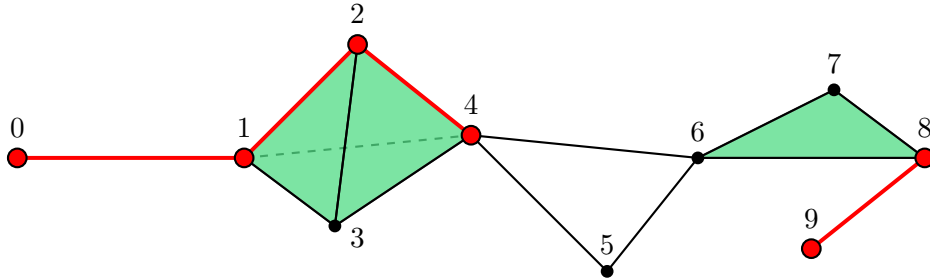


Figura 2.12: Ejemplo de 1-cadena

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

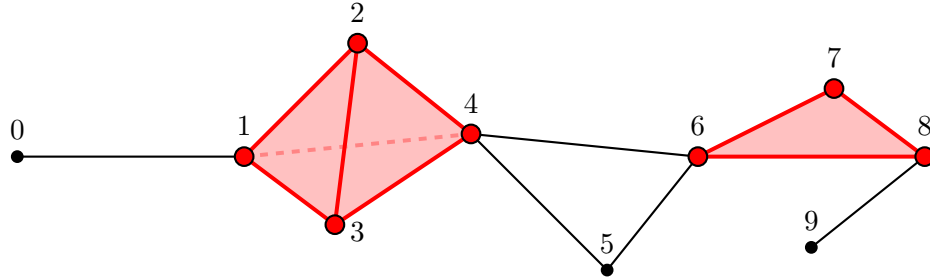


Figura 2.13: Ejemplo de 2-cadena

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

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

## Desarrollo

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

Este grupo es un grupo abeliano, y como en nuestro caso los coeficientes están tomados en el cuerpo  $\mathbb{Z}_2$ ,  $C_p(K)$  es un espacio vectorial sobre  $\mathbb{Z}_2$ . Fijado  $p \in \mathbb{Z}$ , una base del espacio vectorial  $C_p(K)$  es el conjunto  $\{\sigma_i^p \mid i = 1, \dots, s_p\}$  formado por los símlices de dimensión  $p$  de  $K$ . Como consecuencia  $C_p(K) = \{0\}$ , 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 símplex.

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

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

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  $\partial_p : C_p \rightarrow C_{p-1}$  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  $\partial$  conmuta con la suma  $+$ , el conjunto de  $p$ -ciclos  $Z_p = \ker \partial_p$  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  $c \in \operatorname{im} \partial_{p+1}$ .

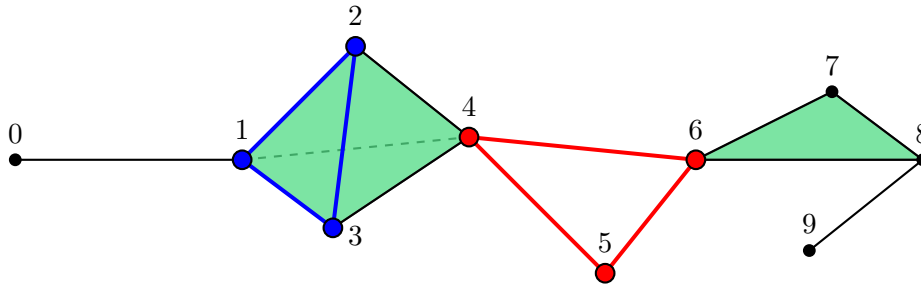


Figura 2.14: Ejemplos de 1-ciclos

Debido a que  $\partial$  conmuta con la suma  $+$ , el conjunto de  $p$ -bordes  $B_p = \text{im } \partial_{p+1}$  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  $B_p \subset Z_p$ . 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

$$\dots \xrightarrow{\partial_{p+2}} C_{p+1} \xrightarrow{\partial_{p+1}} C_p \xrightarrow{\partial_p} C_{p-1} \xrightarrow{\partial_{p-1}} \dots$$

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.

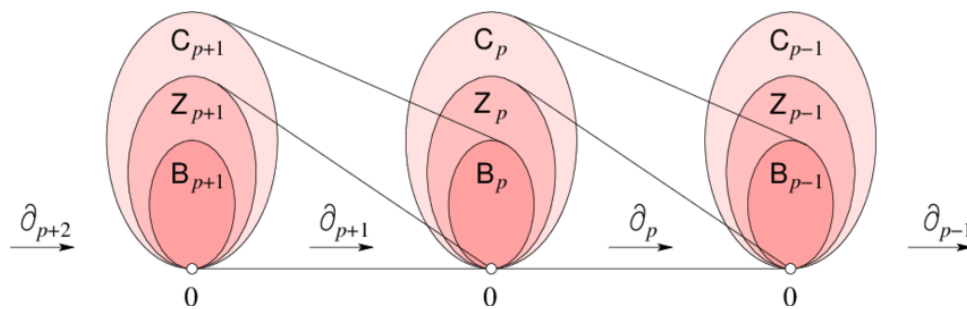


Figura 2.15: Complejo de cadenas representando el grupo de cadenas, el grupo de ciclos y el grupo de bordes. Fuente: [2]

### 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

$$H_p(K) = \frac{Z_p}{B_p}.$$

El *número de Betti  $p$ -dimensional*  $\beta_p(K)$  como la dimensión de  $H_p(K)$ .

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

**Definición 2.1.32.** Diremos que dos ciclos  $c_1, c_2 \in Z_p$  son *homólogos* si existe  $b \in B_p$  tal que

$$c_1 = c_2 + b.$$

Como  $H_p(K)$  es un grupo finito, por el *teorema de Lagrange* sabemos que el número de clases de homología es

$$\text{ord } H_p(K) = \frac{\text{ord } Z_p}{\text{ord } B_p}.$$

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

$$\beta_p = \dim H_p = \dim Z_p - \dim B_p.$$

### 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  $f : K \rightarrow L$  una aplicación simplicial. Para cada  $p$ -símplice  $\sigma^p$  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 símlices forman una base de los espacios vectoriales  $C_p(K)$  y  $C_p(L)$ , mediante una extensión lineal se obtiene una aplicación lineal  $f_{\#} : C_p(K) \rightarrow C_p(L)$ .

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

La propiedad anterior garantiza que  $f_{\#}(Z_p(K)) \subset Z_p(L)$  y  $f_{\#}(B_p(K)) \subset B_p(L)$ . Por tanto  $f_{\#}$  induce una aplicación lineal  $f_* : H_p(K) \rightarrow H_p(L)$ , 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  $\lambda : C_p(K) \rightarrow C_p(\text{Sd}^n K)$  definida sobre los  $p$ -símlices como

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

La aplicación  $\lambda_p$  se denomina *operador subdivisión*.

## 2.1. Conocimientos previos y definiciones

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

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

Donde  $g_*$  es el homomorfismo inducido por  $g$  y  $\lambda_{p*} : H_p(K) \rightarrow H_p(\text{Sd}^n K)$  es el isomorfismo inducido por  $\lambda_p$ .

**Teorema 2.1.7.** Sean  $K$  y  $L$  dos complejos simpliciales y  $f : |K| \rightarrow |L|$  un homeomorfismo. Entonces  $f_* : H_p(K) \rightarrow H_p(L)$  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  $s_p = \dim C_p(K)$ .

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 seguiremos [3] como referencia.

#### Funciones reales de una variable

Sea  $f : \mathbb{R} \rightarrow \mathbb{R}$  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 constante 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.

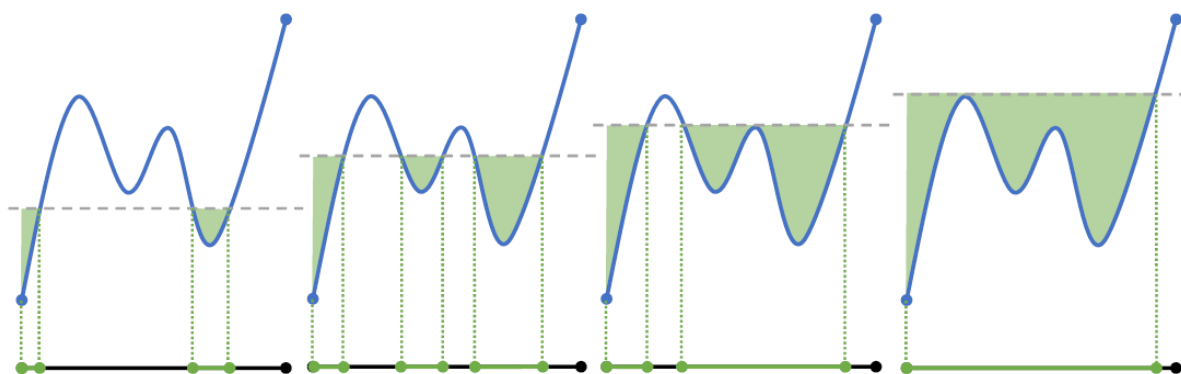


Figura 2.16: Componentes conexas en  $\mathbb{R}$  en las diferentes filtraciones. Fuente: [9]

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

## 2.1. Conocimientos previos y definiciones

- 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.

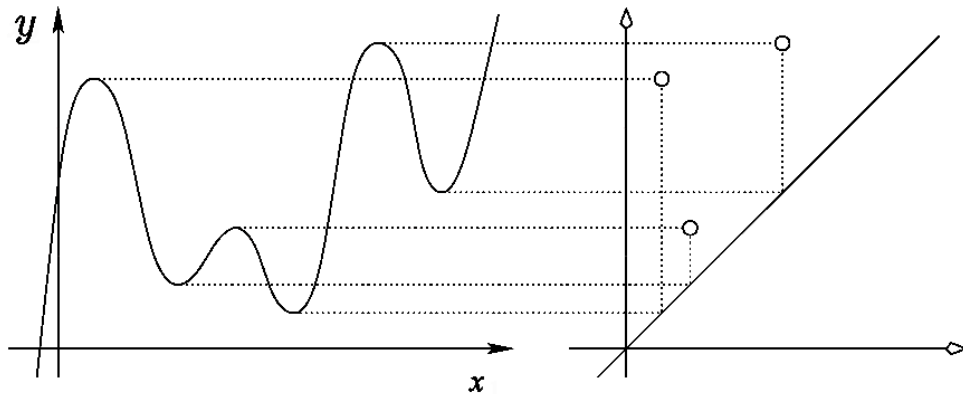


Figura 2.17: Emparejamiento de los puntos críticos de la función de la función de la izquierda representados como puntos en el diagrama de persistencia de la derecha. Fuente: [3]

### 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  $f : \mathbb{M} \rightarrow \mathbb{R}$  una aplicación suave. En este caso, un *punto crítico* es un punto  $p \in \mathbb{M}$  tal que  $\frac{\partial f}{\partial x_i}(p) = 0$  para  $i = 1, \dots, n$ . 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  $p$ .

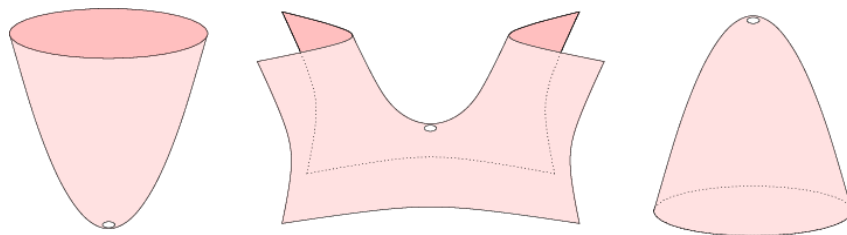


Figura 2.18: De izquierda a derecha tenemos: un punto crítico no degenerado de índice 0, 1 y 2. Fuente: [3]

**Definición 2.1.36.** Sea  $f : \mathbb{M} \rightarrow \mathbb{R}$  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  $t_0 < t_1 < \dots < t_m$  tal que existe un único punto crítico  $p_i \in (t_i, t_{i+1})$  para todo  $i = 0, \dots, m-1$ . 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  $\alpha$  *nace* en  $\mathbb{M}_i$  si no existe en  $\mathbb{M}_{i-1}$ .
- Una clase de homología  $\alpha$  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  $\alpha$  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  $\alpha$  con una clase que ya existía en  $\mathbb{M}_{i-1}$ .

Si  $\alpha$  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*,  $\text{Dgm}_p(f)$ , 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  $f : \mathbb{X} \rightarrow \mathbb{R}$ , 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  $H_k(\mathbb{X}_{a-\epsilon}) \rightarrow H_k(\mathbb{X}_{a+\epsilon})$ <sup>1</sup> 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  $f : \mathbb{X} \rightarrow \mathbb{R}$  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  $F_x = H_k(f^{-1}(-\infty, x])$ , y para  $x < y$ , denotamos como  $f_x^y : F_x \rightarrow F_y$  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:

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

Repetiendo 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

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

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  $\epsilon > 0$ .

<sup>1</sup>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.  $\square$

**Definición 2.1.39.** Sea  $f_x^y : F_x \rightarrow F_y$  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  $\gamma$ ,  $c \in B_k(\mathbb{X}_y)$ . Como consecuencia

$$\ker f_x^y = \frac{Z_k(\mathbb{X}_x) \cap B_k(\mathbb{X}_y)}{B_k(\mathbb{X}_x)}$$

para cada dimensión  $k$  fija.

Sea  $f : \mathbb{X} \rightarrow \mathbb{R}$  una función tame,  $(a_i)_{i=1..n}$  sus valores críticos homológicos y se considera la secuencia 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,  $\mu_i^j$ , como se muestra en la figura 2.19. Donde, considerando  $\beta_x^y$  como una función sobre el plano real extendido  $\overline{\mathbb{R}}^2$ , donde  $\overline{\mathbb{R}} = \mathbb{R} \cup \{-\infty, +\infty\}$ ; entonces,  $\mu_i^j$  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 de en 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*  $\text{Dgm}(f) \subset \overline{\mathbb{R}}^2$  de  $f$  es el multiconjunto de puntos  $(a_i, a_j)$  con multiplicidad  $\mu_i^j$  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

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

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

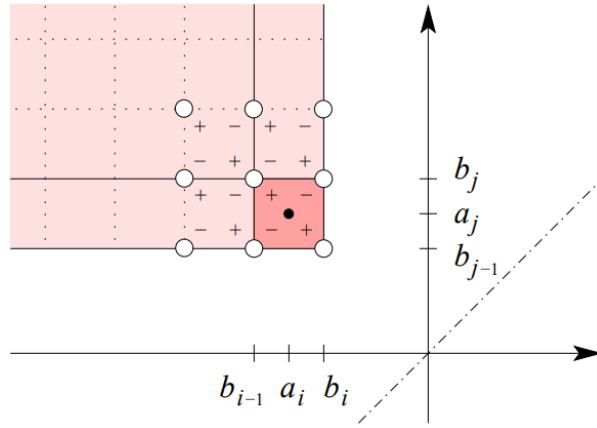


Figura 2.19: La multiplicidad del punto  $(a_i, a_j)$  es la suma alternada de los números de Betti persistentes en las esquinas del cuadrado  $[b_{i-1}, b_i] \times [b_{j-1}, b_j]$ . Fuente: [4]

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*

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

*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

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

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:

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

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.  $\square$

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

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

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

Estas filtraciones pueden ser obtenidas de diversas formas. Por un lado, podremos obtener filtraciones variando el radio de los complejos de Čech, Vietoris-Rips y alfa complejos; y más adelante veremos cómo obtener filtraciones de complejos simpliciales más generales a partir de la filtración por las estrellas inferiores de una función PL.

De esta forma, al igual que vimos en las funciones Morse, una clase de homología  $\alpha$  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  $\alpha$  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  $\alpha$ , 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  $f_i^j(\gamma) \in F_{i-1}^j$ .

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

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

Donde a  $\beta_i^{j-1}$  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,  $\mu_i^j$ , 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  $\mu_i^j$  clases de homología independientes cuya persistencia coincide con la distancia del punto  $(a_i, a_j)$  a su proyección vertical sobre la diagonal  $\Delta$ . Por razones técnicas, los puntos de la diagonal se añaden al diagrama de persistencia con multiplicidad infinito.

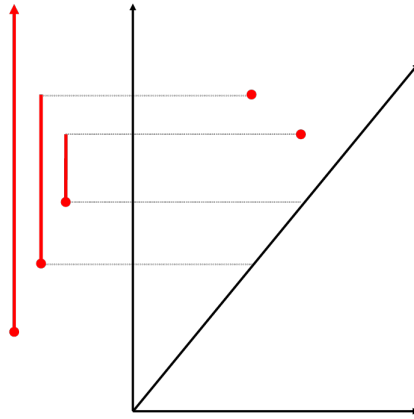


Figura 2.20: Código de barras asociado a un diagrama de persistencia. Fuente: [10]

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  $\mu_i^j$  intervalos semiabiertos  $[a_i, a_j)$ , como se muestra en la figura 2.20.

### Funciones PL

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

**Definición 2.1.45.** Sea  $K$  un complejo simplicial con valores reales asignados en todos sus vértices. Se define la función lineal a trozos  $f : |K| \rightarrow \mathbb{R}$  como la extensión lineal de los valores de los vértices sobre los símlices, 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  $f(u_1) < f(u_2) < \dots < f(u_n)$ , 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 símlices 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 símlices en  $\text{St}_- u_i$ , obtenemos la *estrella inferior cerrada*  $\overline{\text{St}_- u_i}$ , que es el menor subcomplejo de  $K$  que contiene a  $\text{St}_- u_i$ . Como  $f$  es inyectiva en sus vértices, cada símlice 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  $|K| \rightarrow \mathbb{R}$  una función PL. Se define la *filtración de  $K$  por las estrellas inferiores de  $f$*  como la filtración de subcomplejos  $\emptyset = K_0 \subseteq K_1 \subseteq \dots \subseteq K_n = K$ , donde  $K_i = K_{i-1} \cup \overline{\text{St}}_{-} u_i$ .

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 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 introduciremos el *teorema de estabilidad de los diagramas de persistencia*, y profundizaremos en su demostración 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 “cercaños”. 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  $f, g : \mathbb{X} \rightarrow \mathbb{R}$  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

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

$$W_\infty(X, Y) = \inf_{\eta: X \rightarrow Y} \sup_{x \in X} \|x - \eta(x)\|_\infty$$

siendo  $\eta : X \rightarrow Y$  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 determinan en mayor escala la distancia bottleneck son los del primer tipo, y los que menor importancia tienen son los del último tipo, ya que completarán el emparejamiento sin afectar en la distancia.

*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  $f, g : \mathbb{X} \rightarrow \mathbb{R}$  dos funciones tame continuas. Entonces, para cada dimensión  $k$ , la distancia bottleneck entre los diagramas de persistencia esta acotada por la distancia  $L_\infty$  entre las funciones, es decir,*

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

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.

### 2.2.2. Estabilidad para la distancia Hausdorff

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

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

### 2.2.3. Estabilidad para la distancia bottleneck

Subsección por hacer



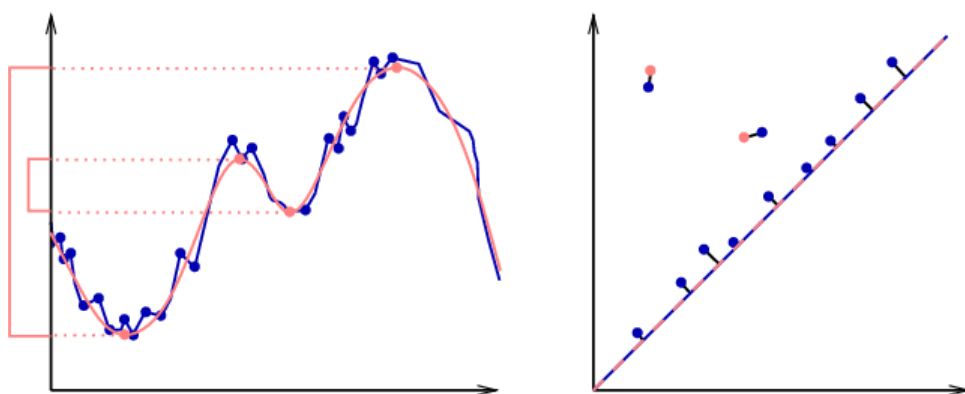


Figura 2.21: A la izquierda se muestran dos funciones cercanas, una con muchos valores críticos y otra con cuatro. A la derecha se muestran los diagramas de persistencia superpuestos, con la biyección que da lugar a la distancia bottleneck. Fuente: [4]

## 2.3. 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,

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

*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

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

Sea  $A = \{x_1, x_2, \dots, x_m\}$  y  $B = \{y_1, y_2, \dots, y_m\}$  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.

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  $\mathcal{O}(n * m)$ .

---

**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:  $cmax \leftarrow 0$ 
2: for  $x \in A$  do
3:    $cmin \leftarrow \infty$ 
4:   for  $y \in B$  do                                      $\triangleright$  Calculamos  $d_\infty(x, B) = \inf_{y \in B} d_\infty(x, y)$ 
5:      $d \leftarrow \|x - y\|_\infty$ 
6:     if  $d < cmin$  then
7:        $cmin \leftarrow d$ 
8:     end if
9:   end for
10:  if  $cmin > cmax$  then                                    $\triangleright$  Recalculamos el supremo
11:     $cmax \leftarrow cmin$ 
12:  end if
13: end for
14: return  $cmax$ 

```

---

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 [11].

### 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 en 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  $c : A \rightarrow \mathbb{R}$  donde a cada arista  $uv \in A$  se le asigna la distancia  $L_\infty$  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  $\text{card } M < \text{card } M'$ .
- $M$  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  $\epsilon$  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 2.22 podemos observar un ejemplo del digrafo  $D_i$  asociado a un emparejamiento  $M_i$ .

**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.

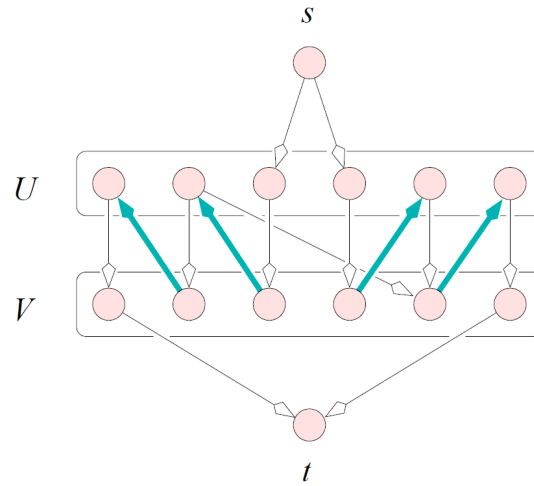


Figura 2.22: Digrafo asociado a un emparejamiento de cuatro aristas. Fuente: [2]

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 sí 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:**  $G(\epsilon) = (U \cup V, A_\epsilon)$  grafo bipartido

**Salida:**  $M_i$  es un emparejamiento máximo de  $G(\epsilon)$

- 1:  $M_0 \leftarrow \emptyset$
  - 2:  $i \leftarrow 0$
  - 3: **while** existe un camino de  $M_i$ -aumento en  $D_i$  **do**
  - 4:     aumentar  $M_i$  usando el camino para obtener  $M_{i+1}$
  - 5:      $i \leftarrow i + 1$
  - 6: **end while**
  - 7: **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  $A_\epsilon$ . 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  $\epsilon$  tal que  $G(\epsilon)$  tiene un emparejamiento perfecto, seguiremos una variante del método húngaro, el cual se utiliza para resolver problemas de asignación [12].

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

- A. Si el subgrafo  $G(0)$ , el cual 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*  $d_i : U \times V \rightarrow \mathbb{R}$ . 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  $G_i(0)$ , el cual es el subgrafo resultante de 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  $\pi$  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  $\pi$  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

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] —, “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] —, “The fiber of the persistence map for functions on the interval,” 2019.
- [11] 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>
- [12] 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

Este capítulo es opcional, y se escribirá de acuerdo con las indicaciones del Tutor.

## Anexo 1: 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)
283             < d:
284             pesoTriang = alfa.umbral(triang)
285             if pesoTriangMin < 0 or pesoTriang < pesoTriangMin:
286                 pesoTriangMin = pesoTriang
287
288         alfa.setCaras([arista], d if pesoTriangMin < 0 else pesoTriangMin)
289
290     return alfa
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
305                     =2,
306                     cmap=cmap)
307
308     plt.plot(puntos[:, 0], puntos[:, 1], 'ko')
309
310     if dim > 0:
311         for arista in K.getCarasN(1):
312             p1 = puntos[arista[0]]
313             p2 = puntos[arista[1]]
314             plt.plot([p1[0], p2[0]], [p1[1], p2[1]], 'k')
315
316     # plt.show()
317
318 def vieterisRips(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 simple 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 simple 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_l[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
554                               set(caras1[i]).issubset(cara))
555
556         return m
557
558     def algoritmoPersistencia(self):
559         """Realiza el algoritmo de persistencia sobre la matriz borde generalizada."""
560         M = self.matrizBordeGeneralizada()
561         lowsArray = [-1 for i in range(len(M))]
562
563         for j in range(len(M)):
564             lowsArray[j] = low(M[:, j])
565             # Comportamiento do-while
566             mismoLow = True
567             while mismoLow and lowsArray[j] >= 0:
568                 mismoLow = False
569                 for k in range(j-1, -1, -1):
570                     if lowsArray[k] == lowsArray[j]:
571                         sumaColZ2(M, k, j)
572                         mismoLow = True
573                         lowsArray[j] = low(M[:, j])
574                         break
575
576         return M, lowsArray
577
578     def persistencia(self):
579         """Cálculo de los puntos del diagrama de persistencia."""
580         _, lowsArray = self.algoritmoPersistencia()
581         dgm = list()
582         carasVisitadas = []
583         for i in range(0, self.dim()):
584             dgm_i = list()
585             numCaras = len(self.getCarasN(i))
586             j = 0
587             while j < len(lowsArray) and numCaras > 0:
588                 if j not in carasVisitadas and lowsArray[j] >= 0 and len(self.carasOrd[
589                     lowsArray[j]][0]) == i+1:
590                     dgm_i.append((self.carasOrd[lowsArray[j]][1], self.carasOrd[j][1]))
591                     numCaras = numCaras - 1
592                     carasVisitadas.append(j)
593                     # Marca de que ya se ha muerto su clase de equivalencia
594                     lowsArray[lowsArray[j]] = -2
595
596             j = j + 1
597
598             j = 0
599             while j < len(lowsArray) and numCaras > 0:
600                 if j not in carasVisitadas and lowsArray[j] == -1 and len(self.carasOrd[j][0])
601                     == i+1:
602                     dgm_i.append((self.carasOrd[j][1], math.inf))
603                     numCaras = numCaras - 1
604                     carasVisitadas.append(j)
605                     # Marca de que ya se ha anadido su persistencia
606                     lowsArray[j] = -2
607
608             j = j + 1

```

```

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] != math.inf])
621         deathI = np.array([c[1] for c in dmgi if c[1] != math.inf])
622         infinity.append([c[0] for c in dmgi if c[1] == math.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[
631                 i]))]))
632
633     ax.scatter(x=birth[i], y=death[i], alpha=0.90, label=r"$H_{\{i\}}$.format(i), zorder
634               =10)
635
636     lims = [
637         np.min([ax.get_xlim(), ax.get_ylim()]), # min of both axes
638         np.max([ax.get_xlim(), ax.get_ylim()]), # max of both axes
639     ]
640     ax.set_xlabel("Birth Time")
641     ax.set_ylabel("Death Time")
642     ax.plot([lims[0], lims[1]], [lims[0], lims[1]], "--", color=(0.3, 0.3, 0.3), zorder
643             =0)
644     ax.plot([lims[0], lims[1]], [maxDeath, maxDeath], "k--", label=r"$\infty$", zorder=0)
645     ax.legend()
646     ax.set_xlim(lims)
647     ax.set_ylim(ymin=lims[0])
648
649     if not os.path.exists("persistencia/"):
650         os.makedirs("persistencia/")
651
652     fig.savefig("persistencia/perDiag.png", dpi=300)
653
654 def codigoBarrasPers(self):
655     """Representación de la persistencia en formato de código de barras."""
656     dmg = self.persistencia()
657     fig, ax = plt.subplots(nrows=len(dmg), sharex=True, dpi=300)
658     ax = ax[:-1]
659     maxDeath = -1
660     infinity = list()
661     birth = list()
662     death = list()
663     for i in range(len(dmg)):
664         dmgi = dmg[i]
665         birthI = np.array([c[0] for c in dmgi if c[1] != math.inf])
666         deathI = np.array([c[1] for c in dmgi if c[1] != math.inf])
667         infinity.append([c[0] for c in dmgi if c[1] == math.inf])
668         maxDeath = max(maxDeath, int(np.amax(deathI))*1.1 + 1)
669         birth.append(birthI)
670         death.append(deathI)
671
672     for i in range(len(infinity)):
673         if infinity[i] != []:
674             birth[i] = np.append(birth[i], np.array(infinity[i]))
675             death[i] = np.append(death[i], np.array([maxDeath for j in range(len(infinity[

```

```

        i))))))
673
        # Elimina las parejas que nacen y mueren a la vez
674
        n = 0
675
        while n < len(birth[i]):
676
            if birth[i][n] == death[i][n]:
677
                birth[i] = np.delete(birth[i], n)
678
                death[i] = np.delete(death[i], n)
679
                n = n-1
680
                n = n+1
681
682
        diff = death[i] - birth[i]
683
        # diff[diff<=0] = 0.005
684
        ax[i].barh(y=np.arange(len(birth[i])),
685
                    width=diff,
686
                    height=0.2,
687
                    align="center",
688
                    left=birth[i],
689
                    label=r"$H_{\}$".format(i),
690
                    color=f"C{i}",
691
                    linewidth=0)
692
693
        ax[i].get_yaxis().set_ticks([])
694
        ax[i].set_ylabel(r"$H_{\}$".format(i), rotation="horizontal")
695
        ax[i].get_yaxis().set_label_coords(-0.035, 0.5)
696
697
        if not os.path.exists("persistencia/"):
698
            os.makedirs("persistencia/")
699
700
        fig.savefig("persistencia/perBarras.png", dpi=300)
701
702
def betti(self, p, incremental=False):
703
    """
704
    Calculo del número de p de Betti.
705
706
    p: int.
707
    """
708
    if incremental and self.dim() == 2:
709
        # Algoritmo incremental
710
        b = self.allBettis(incremental=True)[p]
711
712
    else:
713
        if p == 0:
714
            Zp = len(self.getCarasN(0))
715
        else:
716
            Mp = normSmithZ2(self.matrizBorde(p))
717
            Zp = Mp.shape[1] - matrix_rank(Mp)
718
719
            Bp = matrix_rank(normSmithZ2(self.matrizBorde(p + 1)))
720
721
            b = Zp - Bp
722
723
            self.bettiNums[p] = b
724
725
    return b
726
727
def allBettis(self, incremental=False):
728
    """Calculo de todos los números de Betti."""
729
    if incremental and self.dim() == 2:
730
        # Puede que el resultado sea erróneo si el complejo no está contenido en R2
731
        # Algoritmo incremental
732
        k1Graph = nx.Graph()
733
        nodos = self.getCarasN(0)
734
        k1Graph.add_nodes_from([vertice[0] for vertice in nodos])
735
736
        self.bettiNums[0] = len(nodos)
737
        self.bettiNums[1] = 0
738
        numCompConexas = self.bettiNums[0]
739
740
        for arista in self.getCarasN(1):
741

```

```

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 if __name__ == "__main__":
775     """
776     comp1 = Complejo([(0, 1, 2, 3)])
777     print("-----COMP1-----")
778     analisisComplejo(comp1, set((0, 1)))
779
780     comp2 = Complejo(list(comp1.k_esqueleto(2)))
781     print("\n-----COMP2-----")
782     analisisComplejo(comp2, set((0,)))
783
784     comp3 = Complejo([(0, 1), (1, 2, 3, 4), (4, 5), (5, 6), (4, 6), (6, 7, 8), (8, 9)])
785     print("\n-----COMP3-----")
786     analisisComplejo(comp3, set((4,)))
787
788     comp4 = Complejo(list(comp3.k_esqueleto(1)))
789     print("\n-----COMP4-----")
790     analisisComplejo(comp4, set((4,)))
791
792     comp5 = Complejo([(0, 1, 2), (2, 3), (3, 4)])
793     print("\n-----COMP5-----")
794     analisisComplejo(comp5, set((2,)))
795
796     comp6 = Complejo([(1, 2, 4), (1, 3, 6), (1, 4, 6), (2, 3, 5), (2, 4, 5), (3, 5, 6)])
797     print("\n-----COMP6-----")
798     analisisComplejo(comp6, set((1, 4)))
799
800     comp7 = Complejo(list(comp6.k_esqueleto(1)))
801     print("\n-----COMP7-----")
802     analisisComplejo(comp7, set((1, 4)))
803
804     comp8 = Complejo([(1, 2, 4), (2, 4, 5), (2, 3, 5), (3, 5, 6), (1, 3, 6), (1, 4, 6),
805                       (4, 5, 7), (5, 7, 8), (5, 6, 8), (6, 8, 9), (4, 6, 9), (4, 7, 9),
806                       (1, 7, 8), (1, 2, 8), (2, 8, 9), (2, 3, 9), (3, 7, 9), (1, 3, 7)])
807     print("\n-----COMP8-----")
808     analisisComplejo(comp8, set((1,)))
809
810     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 ordenadas 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                   [0.1840744822466916, 0.7868606964403773],
880                   [0.4496777667376678, 0.874366215574117],
881                   [0.08225571534539433, 0.616710205071694]])
```

```

882
883     curval = [4 * sy.sin(t), 9 * sy.cos(t)]
884     puntos = puntosCurvaRuido(curval, t, 0, 2*np.pi, numPuntos=30)
885
886     curva2 = [1 + 3 * t**2, t**3 - 2 * t]
887
888     puntos = 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',
907                               lines_alpha=0.6)
908         plotalpha(points, K)
909         plt.title(r"$r={}$".format(str(valor)))
910         fig.savefig(f"imgTemp/im{i}.png")
911         images.append(imageio.imread(f"imgTemp/im{i}.png"))
912         i += 1
913         plt.show()
914
915     imageio.mimsave('alphaGif/alpha.gif', images)
916
917     """
918     compl = Complejo([(0, 1, 2, 3)])
919     print(f"Los num de Betti del tetraedro son: {compl.allBettis()}")
920     print(f"Los num de Betti del borde del tetraedro son: {Complejo(compl.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(
939         incremental=True)}")
940
941     planoProy = Complejo([(1, 2, 10), (2, 3, 10), (3, 9, 10), (3, 4, 9), (4, 8, 9), (4, 5, 8),
942                          (2, 3, 5), (3, 5, 6), (3, 6, 4), (4, 6, 7), (4, 5, 7), (2, 5, 7),
943                          (5, 6, 8), (6, 8, 9), (6, 7, 9), (7, 9, 10), (2, 7, 10), (1, 2, 10)
944                          ])
945     print(f"Los num de Betti del plano proyectivo son: {planoProy.allBettis()}")
946
947     asno = Complejo([(1, 3, 5), (1, 5, 6), (1, 3, 6), (2, 3, 5), (2, 4, 5), (4, 5, 6),
948                     (3, 6, 7), (2, 3, 7), (6, 7, 8), (6, 4, 8), (1, 2, 4), (1, 3, 4),
949                     (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 curval = [4 * sy.sin(t), 9 * sy.cos(t)]
963 points = puntosCurvaRuido(curval, 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 """

```