Análisis topológico de datos: teoría y práctica

Aldo Guzmán Sáenz

Índice general

Capítulo 1

Introducción

Consideremos una colección de puntos $X \subseteq \mathbb{R}^2$ como en la figura 1.1, y tratemos de inferir información sobre dichos datos, partiendo de la suposición de que, en efecto, no son únicamente ruido. A primera vista, parece un círculo.

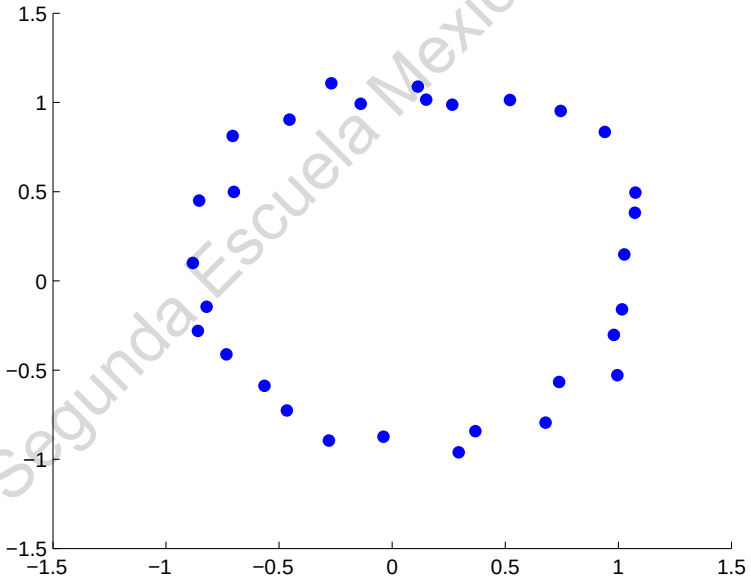


Figura 1.1: X parece un círculo.

Un intento inicial de estudiar la nube con herramientas usuales de topología no provee de mucho, pues estamos trabajando con un conjunto discreto. Consideremos ahora el resultado de "engordar" (considerar un conjunto que contenga a X) de una manera el conjunto X:

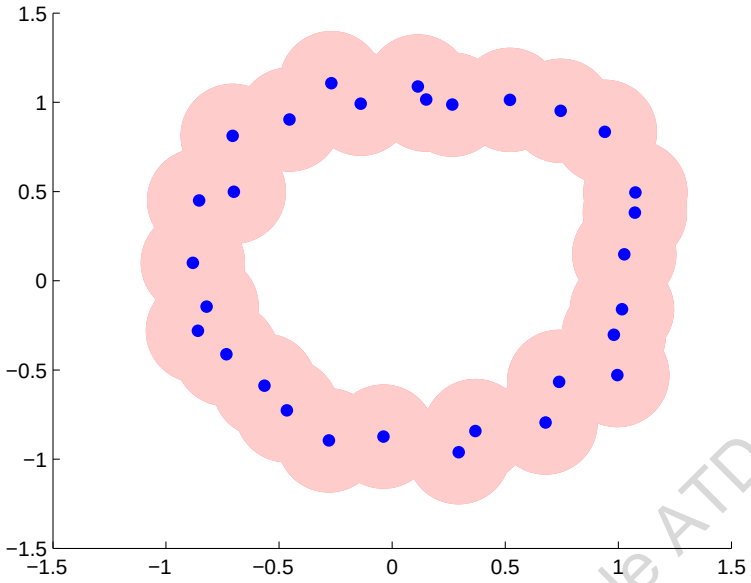


Figura 1.2: X' es, homotópicamente, un círculo.

Ahora podemos inferir ciertas cosas del conjunto X'. Sin embargo, la elección del proceso de engorda de X parece algo arbitrario: ¿qué pasaría si lo engordáramos de otra manera?

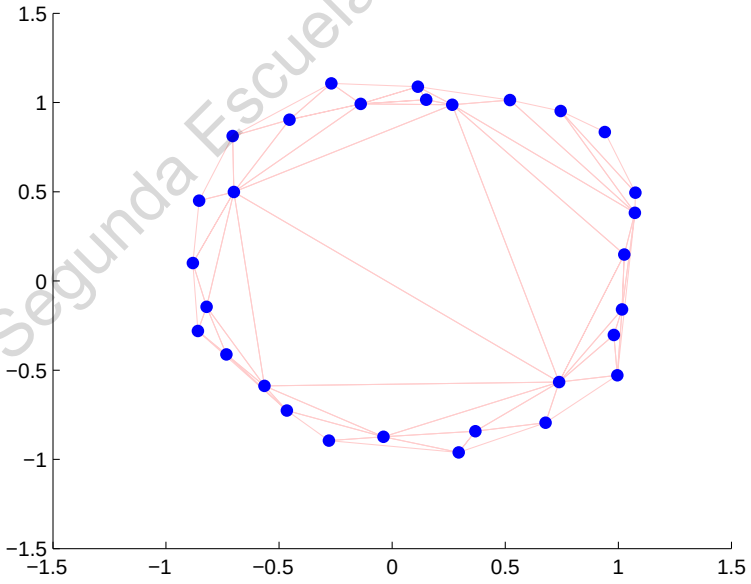


Figura 1.3: X'' es, homotópicamente, muchos círculos con un punto en común.

Para dar conclusiones significativas acerca de la información que contienen los datos, es

necesaria que la elección de estos modelos para un conjunto de puntos dado sea adecuada. Tomando en cuenta esta ambigüedad inherente a cómo están presentados los datos, una posibilidad es elegir una manera parametrizada de engordar a nuestro conjunto, y estudiar las propiedades topológicas que aparecen, o dejan de aparecer, en determinados intervalos de nuestro parámetro. Volviendo a nuestro ejemplo, podríamos tomar bolas de radio $t > 0$ alrededor de los puntos de X, formar su unión, y denotarla por $X(t)$.

Notemos además lo siguiente: los diversos espacios $X(t)$ satisfacen que $X(t_0)$ es un subconjunto de $X(t_1)$ si $t_0 < t_1$. Esto es deseable para que haya alguna noción de buen comportamiento entre el parámetro t, su correspondiente espacio $X(t)$, y eventualmente cualquier construcción algebraica sobre $X(t)$.

Podemos ahora, para cada t, considerar algún complejo simplicial asociado a $X(t)$, calcular algún invariante de topología algebraica, y observar los cambios en estructura que hay cuando variamos la t. El invariante que elegiremos durante estas notas será la homología, puesto que es posible calcularla con una computadora para complejos simpliciales finitos (una pregunta interesante es qué pasa con la homotopía. Un artículo que usa esta noción en el contexto de complejos anudados es [7]).

Más allá del interés teórico acerca de fenómenos de persistencia, para el aspecto práctico es importante el desarrollar ejemplos explícitos en los que se muestre cómo se realiza el análisis de alguna base de datos, y un tal ejemplo se encuentra en el Capítulo 5, en el que se reproduce una porción del trabajo encontrado en el artículo *On the local behavior of spaces of natural images*, de Carlsson, et al. [2], incluyendo código fuente para su implementación en diversos sistemas de cómputo.

Nota (08/Nov/2015): Estas notas son un trabajo en progreso. En esta versión está inicialmente solo homología persistente, e incluso se puede ampliar más lo dicho sobre ésta (la estabilidad de los códigos de barras es en particular un tema que falta, y que está siendo redactado).

Capítulo 2

Preliminares de Topología Algebraica

2.1. Complejos Simpliciales Abstractos

Definición 2.1.1. Un complejo simplicial abstracto es un conjunto V de vértices junto con un subconjunto R del conjunto de subconjuntos no vacíos de V que cumple las siguientes propiedades:

(1) Todos los subconjuntos de un elemento de V están en R.

(2) Si $\sigma \in R$ y $\tau \subseteq \sigma$, $\tau \neq \emptyset$, entonces $\tau \in R$.

Llamamos a los elementos de R símplices, o simplejos, y si $\sigma \in R$ es tal que $\#(\sigma) = n + 1$, entonces llamamos a σ un n-simplejo. Si $\sigma \in R$ y $\tau \subseteq \sigma$, entonces decimos que τ es una cara de σ y que σ es una cocara de τ. Decimos que un complejo simplicial $X = (V, R)$ es finito si V es un conjunto finito o decimos que es localmente finito si cada vértice es elemento de a lo más una cantidad finita de símplices.

Definición 2.1.2. Un mapeo de complejos simpliciales (V, R) a (V', R') es una función $f : V \longrightarrow V'$ tal que para cada $\sigma \in R$, $f(\sigma) \in R'$.

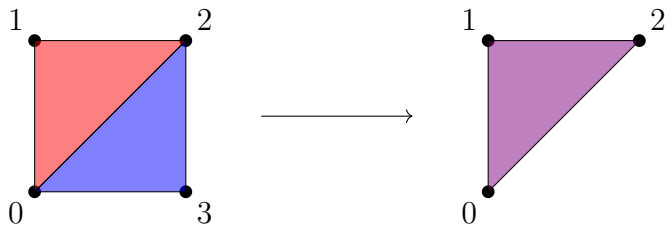


Figura 2.1: Mapeo simplicial: $f(i) = i \ \forall \ i \leq 2$, $f(3) = 1$.

Proposición 2.1.3. (i) La composición de mapeos de complejos simpliciales es un mapeo de complejos simpliciales.

(ii) Para cada complejo simplicial K, id_K es un mapeo de complejos simpliciales.

Demostración. Inmediata de las definiciones □

Definición 2.1.4. Un subcomplejo de un complejo simplicial (V, R) es un complejo simplicial (U, S) tal que $U \subseteq V$ y $S \subseteq R$.

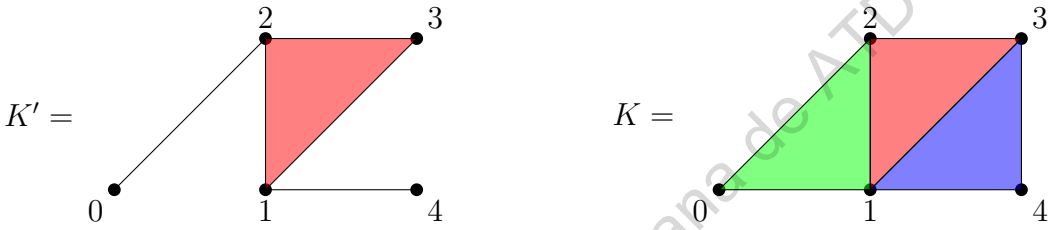
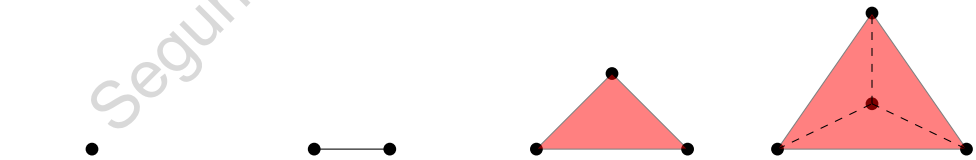


Figura 2.2: K' es subcomplejo de K.

Ejemplo 2.1.5. Para cada n, tenemos el simplejo estándar $\Delta[n]$ con vértices $\{0, \ldots, n\}$ y con conjunto de símplices el conjunto de todos los subconjuntos no vacíos de $\{0, \ldots, n\}$.



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

Figura 2.3: Los primeros cuatro n-simplejos estándares.

Ejemplo 2.1.6 (Gráficas no-dirigidas simples)**.** Para cada n, consideramos el complejo simplicial $G = (V, R)$ donde $V = \{0, \ldots, n\}$ y R es el conjunto obtenido al considerar la unión de un subconjunto de la familia de todos los subconjuntos de exactamente dos elementos de V con la familia de todos los conjuntos con exactamente un elemento de V.

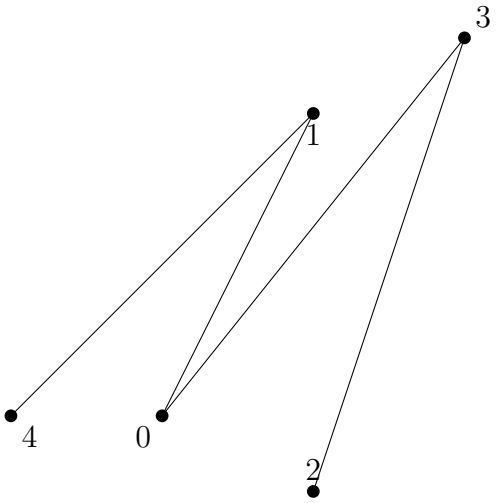


Figura 2.4: Una gráfica simple no-dirigida como complejo sólo consta de 1 y 0 simplejos.

Ejemplo 2.1.7. Podemos modelar la orilla de un cuadrado como un complejo simplicial de la siguiente manera: ponemos $V = \{0, 1, 2, 3\}$ y $R = \{\{0, 1\}, \{1, 2\}, \{2, 3\}, \{3, 0\}, \{0\}, \{1\}, \{2\}, \{3\}\}$.

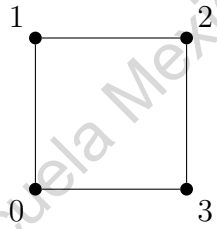


Figura 2.5: Un cuadrado como complejo simplicial

Ejemplo 2.1.8. Expandiendo un poco el ejemplo anterior, ponemos $V = \{0, 1, 2, 3\}$ y $R = \{\{0, 1, 2\}, \{2, 3, 0\}, \{0, 1\}, \{0, 2\}, \{1, 2\}, \{2, 3\}, \{3, 0\}, \{0\}, \{1\}, \{2\}, \{3\}\}$.

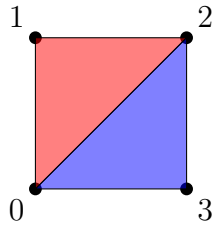


Figura 2.6: Un cuadro relleno, modelado como complejo simplicial

Observación 2.1.9 (No-ejemplo)**.** No podemos modelar el cuadro relleno del ejemplo 2.1.8 como un complejo simplicial poniendo $V = \{0, 1, 2, 3\}$ y

$$R = \{\{0, 1, 2, 3\}, \{0, 1\}, \{1, 2\}, \{2, 3\}, \{3, 0\}, \{0\}, \{1\}, \{2\}, \{3\}\},$$

puesto que faltan simplejos en R (como por ejemplo $\{0,1,2\}$). También implícito aquí está el hecho de que $\{0,1,2,3\}$, al ser un 3-simplejo, no es de la "dimensión adecuada"para formar una cara del cuadro.

Observación 2.1.10. En los ejemplos anteriores hemos usado números naturales para denotar a los elementos del conjunto de vértices, pero nada nos impide denotarlos con dibujos de perritos, o estrellitas, o como nos venga en gana.

Ejemplo 2.1.11 (Nervio de una cubierta)**.** Dado un espacio topológico X y una cubierta $\mathcal{U} = \{U_\alpha\}_{\alpha \in A}$ de X, definimos el nervio de $\mathcal{U}$, denotado por $N\mathcal{U}$, como el complejo simplicial que tiene vértices A, y en el que dado un subconjunto $\{\alpha_0, \ldots, \alpha_k\} \subseteq A$ es un simplejo si y sólo si $U_{\alpha_0} \cap \cdots \cap U_{\alpha_k} \neq \emptyset$.

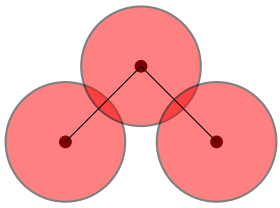


Figura 2.7: Consideremos 3 puntos en $\mathbb{R}^2$ cubiertos por bolas. Por cada intersección de dos conjuntos en la cubierta, agregamos un 1-simplejo.

Definición 2.1.12. Una cubierta buena de un espacio X es una cubierta $\mathcal{U} = \{U_\alpha\}_{\alpha \in A}$ tal que dada cualquier colección finita de elementos de la cubierta su intersección es vacía o contraíble.

A partir de este momento, a menos que se indique lo contrario, trabajaremos únicamente con cubiertas buenas.

Ejemplo 2.1.13 (Complejo de Čech)**.** Dado un conjunto finito de puntos S en $\mathbb{R}^d$, podemos considerar, para cada $r > 0$, la cubierta $\mathcal{U}_r$ de S formada por las bolas cerradas con centro en $s \in S$ y radio r. Definimos el complejo de Čech $\check{C}(S,r)$ como $N\mathcal{U}_r$. Esto es,

$$\check{C}(S,r) = \{\sigma \subseteq S \,|\, \bigcap_{s \in \sigma} \overline{B(s,r)} \neq \emptyset\} \tag{2.1}$$

Ejemplo 2.1.14 (Complejo de Vietoris-Rips)**.** Dado un conjunto finito de puntos S en $\mathbb{R}^d$, podemos considerar, para cada $r > 0$, el complejo simplicial de Vietoris-Rips, definido por

$$VR(S,r) = \{\sigma \subseteq S \,|\, d(x,y) < r \ \forall\, x, y \in \sigma,\ \sigma \neq \emptyset\} \tag{2.2}$$

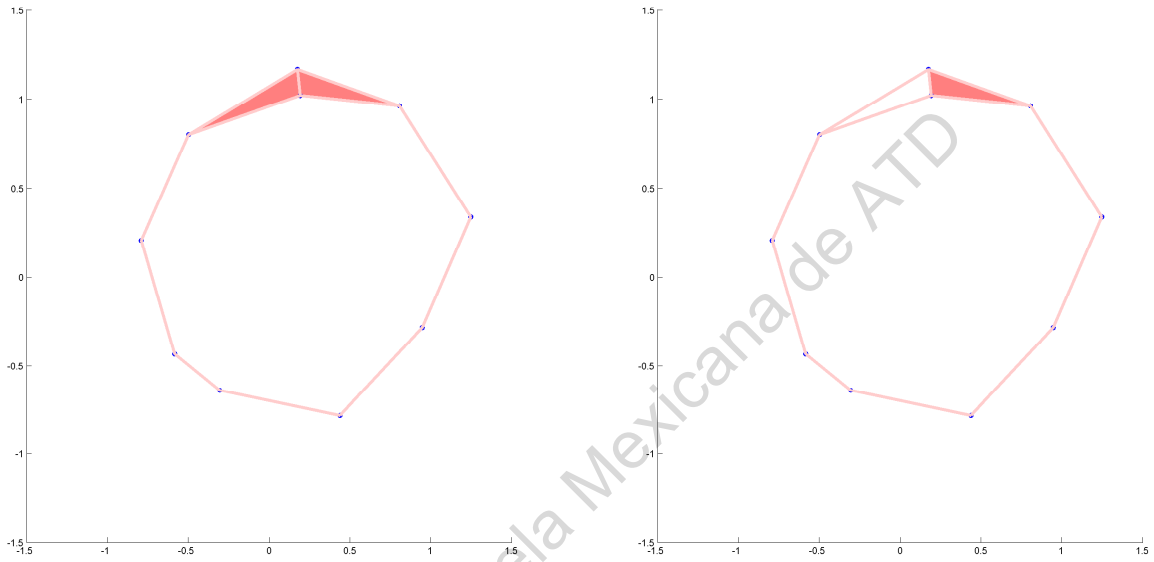
Observación 2.1.15. El complejo de Vietoris-Rips está caracterizado por la propiedad de que un simplice está en el complejo si y sólo si todas sus 1-caras lo están.

Observación 2.1.16. Los complejos de Vietoris-Rips y de Čech están relacionados de la siguiente manera, para cada $r > 0$ tenemos

$$\check{C}(S, r) \subseteq VR(S, 2r) \subseteq \check{C}(S, 2r)$$

Demostración.

□



(a) r=0.77 (b) r=0.77

Figura 2.8: A la izquierda $VR(S, r/2)$ y a la derecha $\check{C}(S, r)$, con S el conjunto de puntos azules en el plano.

Definición 2.1.17. Una orientación de un k-simplejo $\{v_0, \ldots, v_k\}$ es una clase de equivalencia de orden de los vértices de σ, con la relación $(v_0, \ldots, v_k) \sim (v_{\tau(0)}, \ldots, v_{\tau(k)})$ si y sólo si $\operatorname{sgn} \tau = 1$. Dado un simplejo $\{v_0, \ldots, v_k\}$, denotamos al correspondiente simplejo orientado por $[v_0, \ldots, v_k]$.

Observación 2.1.18. Una orientación en un k-simplejo induce una orientación en sus $k-1$ caras: si tenemos $\sigma = [v_0, \ldots, v_k]$ un simplejo orientado, entonces tenemos la i-ésima cara orientada $(-1)^i[v_0, \ldots, v_{i-1}, v_{i+1}, \ldots, v_k]$.

Ejemplo 2.1.19. De manera similar al Ejemplo 2.1.8, ahora consideramos el complejo simplicial con $V = \{0, 1, 2, 3\}$ y simplejos orientados $[0, 1, 2]$, $[0, 2, 3]$, $[0, 1]$, $[0, 2]$, $[1, 2]$, $[2, 3]$, $[3, 0]$, $[0]$, $[1]$, $[2]$, $[3]$.

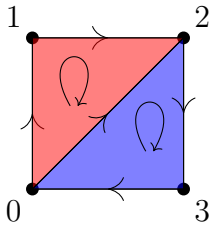


Figura 2.9: Un cuadro relleno, modelado como complejo simplicial con simplejos orientados.

Definición 2.1.20. Sea $X = (V, R)$ un complejo simplicial y pongamos $\mathbb{R}^V$ al espacio vectorial sobre $\mathbb{R}$ generado por los elementos de V, con la topología límite (o de la unión): Un subconjunto $U \subseteq \mathbb{R}^V$ es abierto si y sólo si la intersección con cualquier subespacio de dimensión finita es abierto con la topología estándar.

Definimos la realización geométrica de X como

$$|X| = \{t_0 v_0 + \cdots + t_n v_n \in \mathbb{R}^V \mid t_i \geq 0 \ \forall i = 0, \ldots, n, \quad \sum t_i = 1, \quad \{v_0, \ldots, v_n\} \in S\} \quad (2.3)$$

con la topología de subespacio.

Definición 2.1.21. Para cada n, definimos el n-simplejo geométrico estándar $\Delta^n = |\Delta[n]|$.

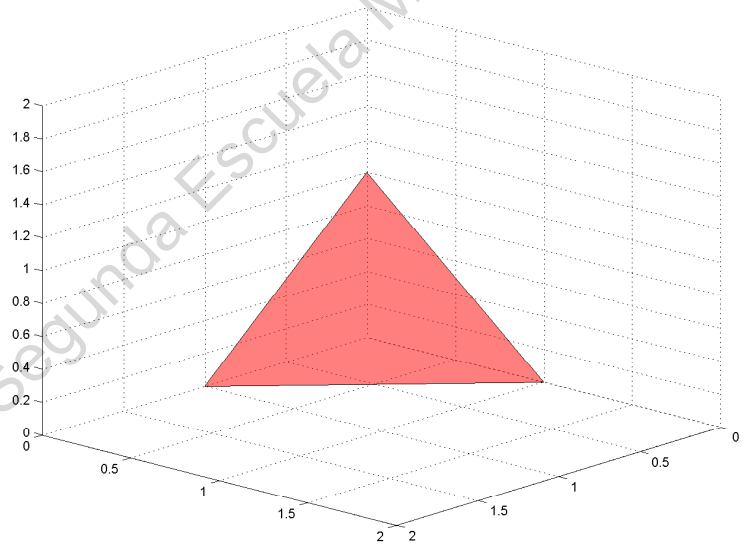


Figura 2.10: El 2-simplejo estándar Δ^2 en $\mathbb{R}^3$.

Definición 2.1.22. Más generalmente, para cada n, definimos un n-simplejo geométrico Δ^n como la cerradura convexa de $n + 1$ puntos independientes en el sentido afín en $\mathbb{R}^d$, con $d \geq n$.

Definición 2.1.23. Para cada complejo simplicial abstracto $X = (V, R)$, definimos la dimensión de $\sigma \in R$ como $\dim \sigma = \#(\sigma) - 1$ y

$$\dim X = \sup_{\sigma \in R}\{\dim \sigma\}$$

Observación 2.1.24. Cuando $K = (V, R)$ es finito, podemos encajar V en posición general en $\mathbb{R}^{2\dim K+1}$, y así podemos interpretar a $|K|$ como un subespacio de $\mathbb{R}^{2\dim K+1}$.

Proposición 2.1.25. Un mapeo de complejos $X \longrightarrow X'$ induce un mapeo $|X| \longrightarrow |X'|$.

Teorema 2.1.26 (Teorema del nervio)**.** Sea $\mathcal{U}$ una familia finita de subconjuntos de $\mathbb{R}^n$ cerrados, convexos tal que $\bigcup \mathcal{U}$ es homeomorfa a la realización de un complejo simplicial abstracto. Entonces $\bigcup \mathcal{U}$ es del tipo de homotopía de $|N\mathcal{U}|$.

2.2. Homología Simplicial

En esta sección, supondremos que K es un complejo simplicial abstracto, y trabajaremos con sus simplejos.

Definición 2.2.1. Sea $k \geq 0$. Para cada σ simplejo de K, escogemos y fijamos una orientación. Definimos el k-ésimo grupo de cadenas simpliciales C_k como el grupo libre abeliano generado por el conjunto de k-simplejos orientados, y poniendo $[\sigma] = -[\tau]$ si $\sigma = \tau$ y $[\sigma]$ y $[\tau]$ tienen diferentes orientaciones. Los elementos de C_k son llamados k-cadenas simpliciales.

Observación 2.2.2. De manera alternativa, podemos definir las k-cadenas simpliciales con coeficientes enteros como funciones del conjunto de k-simplejos orientados a los enteros tales que

1. $c(\sigma) = -c(\sigma')$ si σ y σ' son orientaciones opuestas del mismo simplejo.
2. $c(\sigma) = 0$ para todos salvo una cantidad finita de p-simplejos orientados σ.

Definición 2.2.3. Sea $k \geq 1$. Definimos el operador de frontera $\partial_k : C_k \longrightarrow C_{k-1}$ como el homomorfismo inducido por

$$\partial_k \sigma = \sum_i (-1)^i [v_0, v_1, \ldots, \hat{v}_i, \ldots, v_k],$$

donde $\sigma = [v_0, \ldots, v_k]$ es un k-simplejo y $\hat{v}_i$ indica que v_i no aparece en el simplejo. Ponemos $\partial_0 = 0$.

Proposición 2.2.4. Para $k \geq 1$, $\partial_{k-1}\partial_k = 0$.

Definición 2.2.5. Para $k \geq 0$, definimos $Z_k = \ker \partial_k \subseteq C_k$ y $B_k = \operatorname{im} \partial_{k+1} \subseteq C_k$, llamados los grupos de ciclos y fronteras, de manera respectiva, y definimos el k-ésimo grupo de homología simplicial de K con coeficientes enteros como

$$H_k(K) = H_k(K; \mathbb{Z}) = H_k(C_*) = \frac{Z_k}{B_k}. \tag{2.4}$$

Definición 2.2.6. Dado M un módulo sobre los enteros, definimos la homología simplicial de K con coeficientes en M como

$$H_k(K; M) = H_k(C_* \otimes M). \tag{2.5}$$

En particular, si M es un anillo conmutativo con identidad, le podemos dar a $H(K; M)$ estructura de M-módulo.

Observación 2.2.7. Si M es un campo, podemos definir la homología simplicial de K con coeficientes en M simplemente como en la definición 2.2.1 reemplazando "grupo libre abeliano" por "espacio vectorial sobre M".

Proposición 2.2.8. Para K finito, $H_k(K; M)$ es un M-módulo finitamente generado.

Proposición 2.2.9. Si M es un dominio de ideales principales, $H_k(K; M)$ visto como M-módulo se descompone como la suma directa de una parte libre de torsión (y por lo tanto libre sobre M, pues M es un DIP) y una parte de torsión.

Definición 2.2.10. Denotamos por β_k al rango de la parte libre de torsión de $H_k(K; M)$ como M-módulo y le llamamos el k-ésimo número de Betti de K.

Ejemplo 2.2.11 (Homología del toro sobre $\mathbb{R}$)**.** Consideremos la siguiente triangulación K del toro:

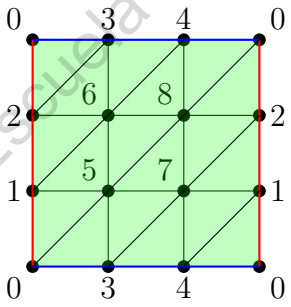


Figura 2.11: Una triangulación del toro.

En este caso, las orientaciones que escogeremos para cada simplejo al generar las cadenas serán las que llevan los vértices en orden ascendente, y las bases estándares consistirán de los simplejos ordenados lexicográficamente. Determinaremos primero $\operatorname{Im} \partial_1$. Tenemos que ∂_1 está representado matricialmente como:

$$\partial_1 = \begin{pmatrix}
-1 & -1 & -1 & -1 & -1 & -1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
1 & 0 & 0 & 0 & 0 & 0 & -1 & -1 & -1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & -1 & -1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & -1 & -1 & -1 & -1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & -1 & -1 & -1 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & -1 & -1 & -1 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & -1 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 1 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 1
\end{pmatrix}$$

La forma escalonada reducida por filas de la transpuesta de esta matriz es

$$RREF(\partial_1^t) = \begin{pmatrix}
1&0&0&0&0&0&0&0&0&-1\\
0&1&0&0&0&0&0&0&0&-1\\
0&0&1&0&0&0&0&0&0&-1\\
0&0&0&1&0&0&0&0&0&-1\\
0&0&0&0&1&0&0&0&0&-1\\
0&0&0&0&0&1&0&0&0&-1\\
0&0&0&0&0&0&1&0&0&-1\\
0&0&0&0&0&0&0&1&0&-1\\
0&0&0&0&0&0&0&0&1&-1\\
0&0&0&0&0&0&0&0&0&0\\
0&0&0&0&0&0&0&0&0&0\\
0&0&0&0&0&0&0&0&0&0\\
0&0&0&0&0&0&0&0&0&0\\
0&0&0&0&0&0&0&0&0&0\\
0&0&0&0&0&0&0&0&0&0\\
0&0&0&0&0&0&0&0&0&0\\
0&0&0&0&0&0&0&0&0&0\\
0&0&0&0&0&0&0&0&0&0\\
0&0&0&0&0&0&0&0&0&0\\
0&0&0&0&0&0&0&0&0&0\\
0&0&0&0&0&0&0&0&0&0\\
0&0&0&0&0&0&0&0&0&0
\end{pmatrix}$$

por lo que su imagen está generada por los 8 vectores fila no cero. Por lo tanto

$$H_0(K) = Z_0/B_0 = C_0/B_0 = \mathbb{R}^9/\operatorname{Im}\partial_1 = \mathbb{R}.$$

Ahora calculamos $H_1(K)$. Primero determinamos $\ker\partial_1$. Una base de este espacio consiste de los 19 vectores fila

$$\begin{pmatrix}
1&0&0&0&0&-1&0&0&0&0&1&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&1\\
0&1&0&0&0&-1&0&0&0&0&0&0&0&0&1&0&0&0&0&0&0&0&0&0&0&0&0\\
0&0&1&0&0&-1&0&0&0&0&0&0&0&0&0&0&0&0&1&0&0&0&0&0&0&0&1\\
0&0&0&1&0&-1&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&1&0&0&0&0&0\\
0&0&0&0&1&-1&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&1&0&0\\
0&0&0&0&0&0&1&0&0&0&-1&0&0&0&1&0&0&0&0&0&0&0&0&0&0&0&-1\\
0&0&0&0&0&0&0&1&0&0&-1&0&0&0&0&0&0&0&1&0&0&0&0&0&0&0&-1\\
0&0&0&0&0&0&0&0&1&0&-1&0&0&0&0&0&0&0&0&0&0&0&1&0&0&0&-1\\
0&0&0&0&0&0&0&0&0&1&-1&0&0&0&0&0&0&0&0&0&0&0&0&1&0&0&-1\\
0&0&0&0&0&0&0&0&0&0&0&1&0&0&-1&0&0&0&1&0&0&0&0&0&0&0&1\\
0&0&0&0&0&0&0&0&0&0&0&0&1&0&-1&0&0&0&0&0&0&0&0&0&1&0&0\\
0&0&0&0&0&0&0&0&0&0&0&0&0&1&-1&0&0&0&0&0&0&0&0&0&0&0&1\\
0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&1&0&0&-1&0&0&1&0&0&0&0&-1\\
0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&1&0&-1&0&0&0&0&0&1&0&-1\\
0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&1&-1&0&0&0&0&0&0&1&-1\\
0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&1&0&-1&0&0&0&1&0\\
0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&1&0&0&-1&0&0&0&1\\
0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&1&0&0&-1&1&0&0\\
0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&1&0&0&-1&0&1&-1
\end{pmatrix}$$

Ahora determinamos $\operatorname{Im}\partial_2$. La matriz asociada es

$$\partial_2 = \begin{pmatrix}
1&1&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0\\
0&0&1&1&0&0&0&0&0&0&0&0&0&0&0&0&0&0\\
0&0&-1&0&1&0&0&0&0&0&0&0&0&0&0&0&0&0\\
-1&0&0&0&0&1&0&0&0&0&0&0&0&0&0&0&0&0\\
0&-1&0&0&-1&0&0&0&0&0&0&0&0&0&0&0&0&0\\
0&0&0&-1&0&-1&0&0&0&0&0&0&0&0&0&0&0&0\\
0&0&0&0&0&0&1&1&0&0&0&0&0&0&0&0&0&0\\
1&0&0&0&0&0&0&0&1&0&0&0&0&0&0&0&0&0\\
0&1&0&0&0&0&0&0&0&1&0&0&0&0&0&0&0&0\\
0&0&0&0&0&0&-1&0&0&-1&0&0&0&0&0&0&0&0\\
0&0&0&0&0&0&0&-1&-1&0&0&0&0&0&0&0&0&0\\
0&0&1&0&0&0&0&0&0&0&1&0&0&0&0&0&0&0\\
0&0&0&0&0&0&1&0&0&0&-1&0&0&0&0&0&0&0\\
0&0&0&0&0&0&0&1&0&0&0&1&0&0&0&0&0&0\\
0&0&0&1&0&0&0&0&0&0&0&-1&0&0&0&0&0&0\\
0&0&0&0&0&0&0&0&0&0&0&0&1&1&0&0&0&0\\
0&0&0&1&0&0&0&0&0&0&0&0&0&0&1&0&0&0\\
0&0&0&0&0&0&0&0&0&0&1&0&-1&0&0&0&0&0\\
0&0&0&0&0&0&0&0&0&0&0&0&0&-1&-1&0&0&0\\
0&0&0&0&0&0&0&0&0&0&0&0&1&0&0&1&0&0\\
0&0&0&0&0&0&0&0&1&0&0&0&0&1&0&0&0&0\\
0&0&0&0&0&1&0&0&0&0&0&0&0&0&0&-1&0&0\\
0&0&0&0&0&0&0&0&0&1&0&0&0&0&0&0&1&0\\
0&0&0&0&0&0&0&0&0&0&0&0&0&0&1&0&0&1\\
0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&-1&-1\\
0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&1&1&0\\
0&0&0&0&0&0&0&0&0&0&0&1&0&0&0&0&0&1
\end{pmatrix}$$

Y la forma escalonada reducida por filas de su transpuesta es

$$RREF(\partial_2^t) = \left(\begin{array}{cccccccccccccccccccccccccccc}
1&0&0&0&0&-1&0&0&0&0&1&0&0&0&0&0&0&0&0&-1&1&0&0&0&0&0\\
0&1&0&0&0&-1&0&0&0&0&0&0&0&0&1&0&0&0&0&0&0&0&0&0&0&0\\
0&0&1&0&0&-1&0&0&0&0&0&0&-1&0&1&0&0&0&1&0&-1&1&0&0&0&-1&0\\
0&0&0&1&0&-1&0&0&0&0&0&0&0&0&0&0&0&0&0&0&1&0&0&0&0&0\\
0&0&0&0&1&-1&0&0&0&0&0&0&-1&0&1&0&0&0&0&0&-1&1&0&0&1&-1&-1\\
0&0&0&0&0&0&1&0&0&0&-1&0&0&0&1&0&0&0&0&0&0&0&0&0&0&-1\\
0&0&0&0&0&0&0&1&0&0&-1&0&0&0&0&0&0&0&0&1&0&0&0&0&0&0\\
0&0&0&0&0&0&0&0&1&0&-1&0&-1&0&1&0&0&0&0&0&0&0&0&1&-1&-1\\
0&0&0&0&0&0&0&0&0&1&-1&0&-1&0&1&0&0&0&0&0&0&0&0&0&0&-1\\
0&0&0&0&0&0&0&0&0&0&1&-1&0&0&0&0&0&0&1&0&-1&1&0&0&0&-1&0\\
0&0&0&0&0&0&0&0&0&0&0&1&-1&0&0&0&0&0&0&0&0&0&0&0&0&0&1\\
0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&1&0&0&-1&0&1&0&0&0&0&0&0\\
0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&1&0&-1&0&0&1&0&0&0&0&-1\\
0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&1&-1&0&1&-1&0&0&0&1&0\\
0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&1&0&-1&0&0&0&1&0\\
0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&1&0&-1&1&0&0\\
0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&1&-1&0&1&0
\end{array}\right),$$

luego

$$H_1(K) = Z_1/B_1 = \ker \partial_1 / \operatorname{Im} \partial_2 = \mathbb{R}^2.$$

Por último, se sigue de un cálculo directo que $\ker \partial_2$ está generado por el vector

$$(1, -1, 1, -1, 1, 1, -1, 1, -1, 1, -1, -1, -1, 1, -1, 1, -1, 1),$$

por lo que

$$H_2(K) = \mathbb{R}.$$

Más adelante usaremos maneras más eficientes de determinar la homología de complejos simpliciales, incluyendo algoritmos para calcular homología sobre los enteros.

Un k-ciclo que también es una frontera corresponde a 0 en homología. Estos k-ciclos no ven huecos en el complejo y decimos que son triviales en homología. Los k-ciclos no triviales en homología, por otro lado, detectan huecos en el complejo simplicial en el que estamos trabajando. Esto no quiere decir que cada ciclo no trivial en Z_k induzca una clase distinta en homología, ya que puede haber muchos ciclos no triviales en Z_k que sean homólogos.

¿Qué quiere decir que dos ciclos sean homólogos? De la definición de homología, esto quiere decir que difieren por una frontera. Una manera de visualizar esto es como sigue:

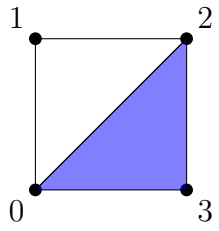


Figura 2.12: El triángulo sin relleno y la orilla del cuadrado son ciclos homólogos. La orilla del triángulo relleno es un ciclo que también es una frontera.

En la Figura 2.12, tenemos que los ciclos

$$[0, 1] + [1, 2] - [0, 2]$$

y

$$[0,1]+[1,2]+[2,3]-[0,3]$$

son homólogos, puesto que

$$[0,1]+[1,2]+[2,3]-[0,3]-([0,1]+[1,2]-[0,2])=[0,2]+[2,3]-[0,3].$$

y $[0,2]+[2,3]-[0,3]$ es frontera del 2-simplejo $[0,2,3]$.

Proposición 2.2.12. Cada mapeo de complejos simpliciales $f : K \longrightarrow K'$ induce de manera canónica un mapeo en homología $H_i(f) : H_i(K) \longrightarrow H_i(K')$. Si tenemos $K \xrightarrow{f} K' \xrightarrow{g} K''$ entonces $H_i(g \circ f) = H_i(g) \circ H_i(f)$, y $H_i(\mathrm{id}) = \mathrm{id}$.

Capítulo 3

Homología persistente

3.1. Propiedades básicas

En lo que sigue, la notación $\{K^i\}_{i\in I}$ representa una correspondencia entre índices en I y complejos simpliciales, y decir que K pertenece a o es un elemento de $\{K^i\}_{i\in I}$, quiere decir que K está en la imagen de dicha correspondencia. Además, los superíndices no son exponentes ni gradación en cohomología, sino que denotan índices en filtraciones o en familias de conjuntos.

Definición 3.1.1. Dado I un conjunto totalmente ordenado, decimos que una familia de complejos simpliciales $\{K^i\}_{i\in I}$ que satisfacen $K^i \subseteq K^j$ si $i \leq j$ es un complejo simplicial filtrado (ascendente) sobre I.

Definición 3.1.2. Sea K un complejo simplicial. Decimos que $\{K^i\}_{i\in I}$ es una filtración de K si $\{K^i\}_{i\in I}$ es un complejo simplicial filtrado tal que $\bigcup K^i = K$.

Usualmente se toma a I como $\mathbb{R}$, $\mathbb{Z}$, $\mathbb{Z}^+$ o $\mathbb{Z}^+ \cup \{0\}$. Nosotros trabajaremos en esta sección con filtraciones de complejos sobre $\mathbb{Z}^+ \cup \{0\}$. De ahora en adelante, y a menos que se indique lo contrario, en estas notas "filtración" querrá decir "filtración finita", esto es, una filtración en la que el número de complejos distintos K^i es finito. Usualmente omitiremos el conjunto de índices en la notación. También abusaremos de notación para denotar por K a un complejo simplicial filtrado o a la unión de todos los elementos de nuestra filtración.

Podemos representar cada una de estas filtraciones como

$$K^0 \subsetneq K^1 \subsetneq \cdots \subsetneq K^m = K = K^{m+1} = K^{m+2} = \cdots$$

poniendo $K^i = K \ \forall \ i \geq m$. Una visualización de un caso particular aparece en la figura 3.1.

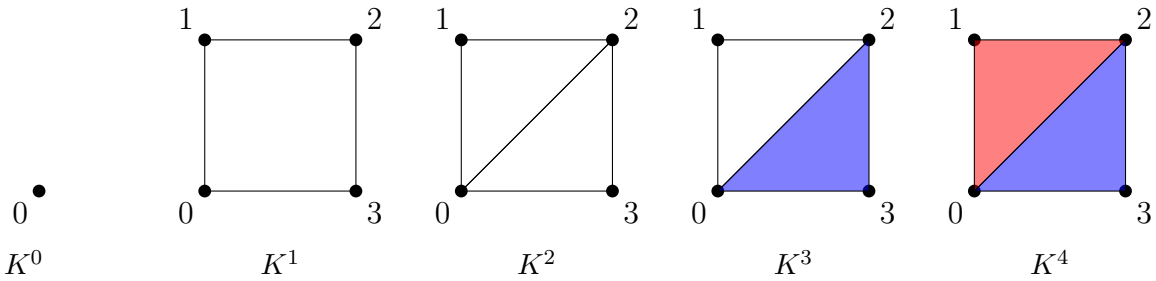


Figura 3.1: Un complejo filtrado.

Observación 3.1.3. Las inclusiones en un complejo simplicial filtrado K

$$K^0 \subsetneq K^1 \subsetneq \cdots \subsetneq K^m = K$$

inducen una familia de complejos de cadenas y morfismos entre ellos

$$C_*(K^0) \xrightarrow{i_\#} C_*(K^1) \xrightarrow{i_\#} \cdots \xrightarrow{i_\#} C_*(K^m),$$

lo que a su vez, para cada l, induce una familia de morfismos y grupos de homología

$$H_l(K^0) \xrightarrow{i_*} H_l(K^1) \xrightarrow{i_*} \cdots \xrightarrow{i_*} H_l(K^m). \tag{3.1}$$

Partiendo de esta observación, damos la siguiente definición.

Definición 3.1.4. Para cada i, p, l con $p \geq 0$, definimos el l-ésimo grupo de homología p-persistente de K^i como

$$H_l^{i,p}(K) = \mathrm{im}(H_l(K^i) \xrightarrow{i_*} H_l(K^{i+p})).$$

Si tomamos coeficientes en un DIP, decimos que el rango de la parte libre de $H_l^{i,p}$ es el l-ésimo número de Betti p-persistente de K^i.

Notemos que $H_l^{i,0}(K) = H_l(K^i)$.

Definición 3.1.5. Sea z un l-ciclo que no es frontera en K^i y que no está en la imagen de $i_\#$ (es decir, aparece por primera vez en K^i). Si z se vuelve trivial en homología para algún K^{j_0}, tomamos j como el mínimo entero tal que z se vuelve trivial en homología y decimos que z y $[z]$ tienen persistencia $j - i - 1$. En otro caso decimos que z y $[z]$ tienen persistencia ∞.

Con esta última definición, podemos darle una interpretación a la Definición 3.1.4: generadores de $H_l^{i,p}(K)$ provienen de ciclos no triviales en homología nacidos antes de o en el tiempo i con persistencia al menos p.

Nota 3.1.6. Algunos autores definen $H_l^{i,j}$ como $\mathrm{im}(H_l(K^i) \xrightarrow{i_*} H_l(K^j))$. Nosotros usamos la notación empleada por Zomorodian en Topology for Computing.

Ejemplo 3.1.7. Tomaremos como ejemplo el complejo filtrado K de la figura 3.1. Tenemos que el ciclo

$$[0,1]+[1,2]+[2,3]-[0,3]$$

aparece en K^1 y se vuelve trivial en homología en K^4, por lo que su persistencia es de 2. De hecho tenemos que

$$H_1^{1,2}(K)=\mathbb{Z},$$

y que

$$H_1^{1,3}(K)=0.$$

Por otro lado, la persistencia del ciclo $[0,1]+[1,2]-[0,2]$, que aparece en K^2 y que se vuelve trivial en homología en K^4, es 1. De hecho tenemos que

$$H_1^{2,1}(K)=\mathbb{Z},$$

y que

$$H_1^{2,2}(K)=0.$$

Ahora que tenemos definida la persistencia para ciclos, podemos enfocarnos en la estructura global que tiene (3.1) para cada l, siguiendo "Computing Persistent Homology", de Carlsson y Zomorodian.

Teorema 3.1.8 (Descomposición de A-módulos f.g.)**.** Sea A un DIP. Si M es un A-módulo finitamente generado, entonces existe un isomorfismo de A-módulos

$$M\cong A^{\oplus m_1}\oplus\left(\bigoplus_{i=1}^{m_2}A/d_iA\right)$$

donde $d_i\in A$ para $i=1,\dots m_2$ es no cero y no es una unidad, y $d_1|d_2|\cdots|d_{m_2}$. Esta descomposición es única salvo orden en los factores y multiplicación por unidades.

Existe una versión análoga para módulos graduados:

Teorema 3.1.9 (Versión graduada del Teorema 3.1.8)**.** Sea A un DIP graduado sobre $\mathbb{Z}^+\cup\{0\}$. Si M es un A-módulo graduado sobre $\mathbb{Z}^+\cup\{0\}$ finitamente generado, entonces existe un isomorfismo de A-módulos graduados

$$M\cong\left(\bigoplus_{i=1}^{m_1}\Sigma^{s_i}A\right)\oplus\left(\bigoplus_{i=1}^{m_2}\Sigma^{r_i}A/d_iA\right) \tag{3.2}$$

donde Σ^l denota aumentar en l el grado, s_i y r_i son enteros no negativos para toda i, $d_i\in A$ es homogéneo, no cero y no es una unidad, para $i=1,\dots m_2$ y $d_1|d_2|\cdots|d_{m_2}$. Esta descomposición es única salvo orden en los factores y multiplicación por unidades.

Definición 3.1.10. Sea R un campo. En este caso, $R[t]$ es un DIP graduado. Le damos una estructura a (3.1) de módulo graduado sobre $R[t]$ poniendo

$$M = \bigoplus_{i=0}^{\infty} H_l(K^i),$$

definiendo la acción de elementos de R sobre M que ya se tenía coordenada a coordenada, y poniendo

$$t \cdot (m_1, m_2 \ldots) = (0, i_*(m_1), i_*(m_2), \ldots)$$

Llamamos a este módulo sobre $R[t]$ el módulo graduado correspondiente a la homología persistente de K.

Lo que deseamos es una manera sistemática de obtener, a partir del módulo M, la forma explícita de la parte derecha del isomorfismo (3.2) tomando $A = R[t]$. ¿Para qué queremos el isomorfismo, y para qué queremos la forma explícita mencionada?

Pongamos por ejemplo que estamos trabajando con un complejo que cuenta con un l-ciclo z en K^2 con persistencia infinita. Haciendo la construcción anterior, tenemos que esto corresponde al elemento

$$z' = (0, 0, [z], 0, \ldots) \in M.$$

Al considerar la acción del anillo $R[t]$ sobre z', tenemos que $t^n z' \neq 0 \ \forall \ n \geq 0$. Esto es, el elemento z' no tiene torsión respecto a $R[t]$, y está en la parte libre de M.

Por otro lado, supongamos que en este complejo hay un l-ciclo w en K^1 con persistencia 3. Ponemos

$$w' = (0, [w], 0, \ldots) \in M.$$

Entonces tenemos que $t^1 w'$, $t^2 w'$ y $t^3 w'$ son no cero y $t^n w' = 0$ si $n \geq 4$, por lo que w' tiene torsión respecto a $R[t]$, y consecuentemente está en la parte de torsión de M y en la parte derecha del isomorfismo (3.2).

Lo que estamos ilustrando con este ejemplo es el hecho de que, respecto a $R[t]$, generadores en H_* con persistencia infinita corresponden a elementos en la parte libre y elementos con persistencia finita corresponden a elementos con torsión.

Ejemplo 3.1.11. Consideremos el complejo de la Figura 3.2.

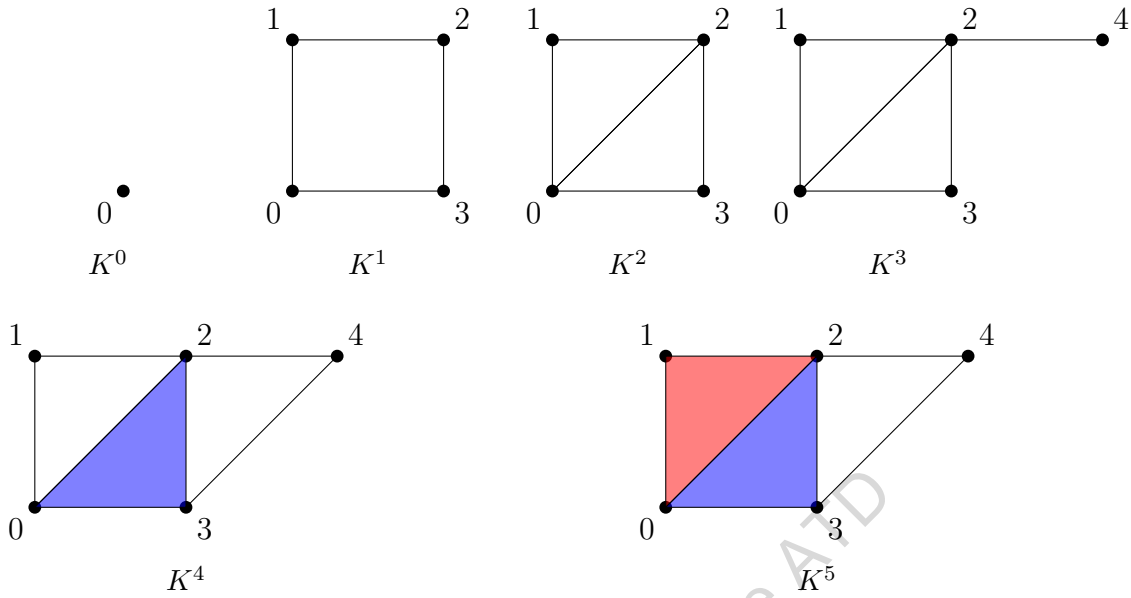


Figura 3.2: Un complejo filtrado con 6 etapas.

Aunque aún no tenemos una forma sistemática de determinar en general la forma que nos interesa de la homología de esta filtración, podemos hacerlo de la manera siguiente: Tomamos el módulo graduado M sobre $R[t]$ correspondiente al complejo filtrado K. Para cada l-ciclo z de K^i, denotaremos por z' a la imagen de $[z]$ en M.

Consideremos ahora los ciclos

$$z_1 = [0,1] + [1,2] + [2,3] - [0,3],$$

$$z_2 = [0,2] + [2,3] - [0,3],$$

y

$$z_3 = [2,4] - [3,4] - [2,3].$$

Es claro que z_1 nace en la etapa 1 y tiene persistencia 3, que z_2 nace en la etapa 2 y su persistencia es 1 y que la persistencia de z_3, nacido en la etapa 4, es infinita. Como habíamos observado anteriormente, esto quiere decir que z_1' tiene torsión t^3 y z_2' tiene torsión t. Por otro lado, z_3' no tiene torsión.

Hay más ciclos en este complejo, que aparecen en diversas etapas de la filtración, pero una verificación rutinaria nos dice que en homología son combinaciones lineales de z_1, z_2 y z_3 al considerarlos en las etapas adecuadas. Para efectos de una descripción como la del isomorfismo (3.2), sólo estos últimos son necesarios. De manera explícita, tenemos:

$$M \cong \Sigma^4 R[t] \oplus \left(\Sigma^1 R[t]/(t^4)\right) \oplus \left(\Sigma^2 R[t]/(t^2)\right).$$

Ahora veremos una manera algorítmica de darle solución a este problema.

3.2. Cálculo de Homología Persistente

Antes de continuar, necesitamos citar resultados acerca de las conexiones entre matrices y la estructura de módulos f.g. sobre DIPs.

Teorema 3.2.1 (Forma Normal de Smith)**.** Sea X una matriz de tamaño $m \times n$ con coeficientes en un DIP A. Entonces existen matrices P y Q invertibles de tamaños $m \times m$ y $n \times n$ respectivamente tales que

$$PXQ = \begin{pmatrix} d_1 & 0 & 0 & & \dots & & 0 \\ 0 & d_2 & 0 & & \dots & & 0 \\ 0 & 0 & \ddots & & & & 0 \\ \vdots & & & d_r & & & \vdots \\ & & & & 0 & & \\ & & & & & \ddots & \\ 0 & & & \dots & & & 0 \end{pmatrix}$$

con $d_1|d_2|\cdots|d_r$. Estos elementos son únicos salvo multiplicación por unidades en A. Llamamos a la matriz PXQ la Forma Normal de Smith de X. La denotaremos por $\mathrm{SNF}(X)$. Notemos que $\mathrm{SNF}(X)$ es de tamaño $m \times n$, al igual que X.

Si X representa un morfismo $M \xrightarrow{f} M'$, podemos interpretar a $\mathrm{SNF}(X)$ como la representación de f respecto a bases obtenidas al aplicar los cambios de base P y Q a las bases tomadas en M' y M para la representación original, de manera respectiva.

La importancia de este teorema en este contexto consiste en que existen algoritmos para calcular $\mathrm{SNF}(X)$, y esta tiene una conexión con la representación del Teorema 3.1.8.

Proposición 3.2.2. Sea A un DIP. Todo A-módulo M finitamente generado es finitamente presentado. Esto es, existe una sucesión exacta

$$A^m \xrightarrow{f} A^n \longrightarrow M \longrightarrow 0,$$

con $m, n \in \mathbb{N}$.

Podemos ahora aplicar el Teorema 3.2.1 a una representación de f y tenemos:

Teorema 3.2.3. En el Teorema 3.1.8, los elementos d_i corresponden con los elementos de la diagonal de la Forma Normal de Smith que no son unidades al considerar la presentación del módulo M sobre A. Cada sumando en la parte libre corresponde con un elemento cero en la diagonal.

Habiendo establecido todo lo anterior, veamos ahora un algoritmo para, dada $X \neq 0$ de tamaño $m \times n$ representando algún ∂_k, obtener $\mathrm{SNF}(X)$ cuando trabajamos con coeficientes módulo 2. Este algoritmo se encuentra en [3].

Algoritmo. (Reducción módulo 2)

```
void Reducir(t)
Si existen k ≥ t, l ≥ t con X(k,l) == 1

     intercambia filas t y k;
     intercambia columnas t y l;
     De i = t + 1 hasta n
          Si X(i,t) == 1 entonces
               suma fila t a fila i;
          Fin Si
     Fin De
     De j = t + 1 a m
          Si X(t,j) == 1 entonces
               suma columna t a columna j
          Fin Si
     Fin De
     Reducir(t + 1)

Fin Si
```


A continuación presentamos un algoritmo para calcular SNF(X) para X de tamaño $m \times n$ con coeficientes en un anillo de polinomios en una variable sobre un campo $\mathbb{F}[t]$. Este algoritmo se encuentra en [5], y se compone de varias funciones. Usamos evaluación de expresiones antes de asignaciones, paso de parámetros por copia (usual), y por referencia (indicadas por * y &). La notación $M_{i,j}$ denota la entrada i, j de la matriz M, y la notación C_i denota la i-ésima entrada de la fila o columna C.

La función principal, SNF, lo que hace es tomar una matriz, hacer que en su esquina superior izquierda quede un elemento tal que tenga sólo ceros a la derecha y hacia abajo y que divida a todos los elementos del menor $1, 1$ de la matriz argumento. Después se vuelve a llamar a sí misma, pero ahora el parámetro es el menor $1, 1$ de la matriz argumento para repetir este proceso hasta que la matriz original esté en la Forma Normal de Smith. Las demás funciones son las que realizan la "carga fuerte". El que el algoritmo alcance tras un número finito de pasos su final es una consecuencia del teorema del algoritmo de la división para polinomios de una variable sobre un campo, y la justificación completa puede ser encontrada en [5].

Sin embargo, cabe notar que este algoritmo no está particularmente optimizado para calcular los códigos de barras. Existen implementaciones de algoritmos en javaplex ([9]), por ejemplo, que están hechas para ese propósito específico.

Algoritmo (Sobre $\mathbb{F}[t]$). Funciones: cerocolumna, cerofila, ceropivote, SNF.

```
matriz cerocolumna(matriz M)

    C = Primera columna de M;
    l(C) = mínimo de los grados de las entradas de C;
    k = índice de la primera entrada de C con deg(C_k) == l(C);
    M = intercambia filas 1 y k de M;
    De i = 2 hasta número de filas de M

        q = cociente de dividir C_i entre C_1;
        M = suma fila i multiplicada por -q a fila 1 de M;

    Fin De
    C = Primera columna de M;
    Si C_i == 0 para toda i ≥ 2 entonces

        Salida = M;

    Si no

        Salida = cerocolumna(M);

    Fin Si

Fin cerocolumna
```

```
matriz cerofila(matriz M)

    C = Primera fila de M;
    l(C) = mínimo de los grados de las entradas de C;
    k = índice de la primera entrada de C con deg(C_k) == l(C);
    M = intercambia columnas 1 y k de M;
    De i = 2 hasta número de columnas de M

        q = cociente de dividir C_i entre C_1;
        M = suma columna i multiplicada por -q a columna 1 de M;

    Fin De
    C = Primera fila de M;
    Si C_i == 0 ∀ i ≥ 2 entonces

        Salida = M;

    Si no

        Salida = cerofila(M);

    Fin Si
```

```
Fin cerofila

matriz ceropivote(matriz M)

    l(M) = mínimo de los grados de las entradas de M_{i,k};
    k = columna de la primera entrada de M con deg(M_{i,k}) == l(M);
    M = intercambia columnas 1 y k de M;
    Mientras M_{1,i} ≠ 0 o M_{j,1} ≠ 0 para algunos i, j ≥ 2
        M = cerocolumna(M);
        M = cerofila(M);
    Fin Mientras
    Salida = M;

Fin ceropivote
```

```
void SNF(matriz *N)

    Si *N == 0
        Romper SNF;
    Fin Si
    M =* N;
    M = ceropivote(M);
    Para M_{i,j} tal que M_{1,1} ∤ M_{i,j}
        M = suma columna j a la columna 1 de M;
        M = ceropivote(M);
    Fin Para
    Si número de filas de M > 1 y número de columnas de M > 1
        SNF(&(Menor 1,1 de M));
    Fin Si

Fin SNF
```

Capítulo 4

Categorificación de persistencia

Habiendo visto ya una presentación de la homología persistente, con ejemplos explícitos y algoritmos, podemos ahora pensar en un marco de referencia en el cual muchas de las definiciones admiten generalizaciones, y donde se pueden considerar más modos de persistencia: El marco de la teoría de categorías.

4.1. Definiciones de teoría de categorías

Definición 4.1.1. Una categoría $\mathbf{C}$ es una colección de objetos $\mathrm{Obj}(\mathbf{C})$ y, para cada pareja (A, B) de objetos en $\mathbf{C}$ una colección de flechas $\mathbf{C}(A, B)$ sujeta a las siguientes condiciones:

1. La composición de flechas está definida: para $f \in \mathbf{C}(A, B)$ y $g \in \mathbf{C}(B, C)$ existe una única flecha $gf \in \mathbf{C}(A, C)$.
2. La composición de flechas es asociativa: para todas $f \in \mathbf{C}(A, B)$, $g \in \mathbf{C}(B, C)$, $h \in \mathbf{C}(C, D)$, $h(gf) = (hg)f$.
3. Para cada objeto $A \in \mathrm{Obj}(\mathbf{C})$, existe un elemento $1_A \in \mathbf{C}(A, A)$ tal que para todo objeto $X \in \mathrm{Obj}(\mathbf{C})$ tenemos
   a) Para todo $f \in \mathbf{C}(A, X)$, $f1_A = f$.
   b) Para todo $f \in \mathbf{C}(X, A)$, $1_A f = f$.

Las flechas son usualmente llamadas morfismos. Abusaremos de notación escribiendo $A \in \mathbf{C}$ para indicar que A es objeto de $\mathbf{C}$. Otra notación para $\mathbf{C}(A, B)$ es $\mathrm{Hom}_{\mathbf{C}}(A, B)$.

En el estudio de las matemáticas, uno se encuentra frecuentemente con objetos que forman categorías. Ponemos algunos ejemplos usuales a continuación, y ejemplos que ya hemos encontrado en estas notas.

Ejemplo 4.1.2. Set, la categoría de conjuntos. Los objetos son conjuntos y los morfismos son funciones entre ellos.

Ejemplo 4.1.3. $\mathbf{Vect}_{\mathbb{F}}$, la categoría de espacios vectoriales sobre un campo $\mathbb{F}$. Los objetos son espacios vectoriales sobre $\mathbb{F}$ y los morfismos son transformaciones lineales. Una generalización de esto es cuando R es un anillo conmutativo con identidad y consideramos la categoría $\mathbf{Mod}_R$ de módulos unitarios sobre R y morfismos de módulos.

Ejemplo 4.1.4. FinVect$_{\mathbb{F}}$, la categoría de espacios vectoriales de dimensión finita sobre un campo $\mathbb{F}$. Los objetos son espacios vectoriales sobre $\mathbb{F}$ y los morfismos son transformaciones lineales.

Ejemplo 4.1.5. Top, la categoría de espacios topológicos. Los objetos son espacios topológicos y los morfismos son funciones continuas.

Ejemplo 4.1.6. Δ, la categoría simplicial. Los objetos son conjuntos $[n] = \{0, \ldots, n\}$ para toda $n \in \mathbb{Z}^+ \cup \{0\}$ y los morfismos son funciones monótonas.

Ejemplo 4.1.7. AbsSimp, la categoría de complejos simpliciales abstractos. Los objetos son complejos simpliciales abstractos y los morfismos son mapeos simpliciales.

Ejemplo 4.1.8. Dadas categorías $\mathbf{C}$ y $\mathbf{D}$, tenemos la categoría de funtores de $\mathbf{C}$ a $\mathbf{D}$: $\mathbf{D}^{\mathbf{C}}$. Los objetos son funtores y los morfismos son transformaciones naturales entre ellos.

Además de ejemplos de este tipo, tenemos que un objeto particular con ciertas propiedades puede ser modelado a través de una categoría. Ponemos un par de ejemplos a continuación.

Ejemplo 4.1.9. Sea G un grupo. Podemos considerar la categoría $\mathbf{G}$ dada por un solo objeto $*$ y $\mathbf{G}(*, *) = G$, interpretando la composición de morfismos en la categoría como la multiplicación de elementos del grupo.

Ejemplo 4.1.10. Sea $(S, \leq)$ un conjunto preordenado (un preorden en S es una relación transitiva y reflexiva). Podemos considerar la categoría $\mathbf{S}$ con $\mathrm{Obj}(\mathbf{S}) = S$ y poniendo un único morfismo $x \to y$ si y sólo si $x \leq y$.

Definición 4.1.11. Sean $\mathbf{A}$, $\mathbf{B}$ categorías. Un funtor $F : \mathbf{A} \longrightarrow \mathbf{B}$ es una regla de asignación tal que:

1. Asocia a cada $A \in \mathbf{A}$ un único $F(A) = B \in \mathbf{B}$.
2. Asocia a cada morfismo $A \xrightarrow{f} B$ un único morfismo $F(A) \xrightarrow{F(f)} F(B)$.
3. La asociación de morfismos es compatible con composición: $F(gf) = F(g)F(f)$.
4. $F(1_A) = 1_{F(A)}$ para toda $A \in \mathbf{A}$.

Proposición 4.1.12. La composición de funtores es un funtor.

Definición 4.1.13. Sean $F, G \in \mathbf{D}^{\mathbf{C}}$ dos funtores. Decimos que una colección de morfismos $\{\eta_X : F(X) \longrightarrow G(X)\}_{X \in \mathbf{C}}$ tal que para todo morfismo $A \xrightarrow{f} B$ de $\mathbf{C}$ se tiene un diagrama conmutativo

$$\begin{array}{ccc} F(A) & \xrightarrow{F(f)} & F(B) \\ {\scriptstyle \eta_A}\downarrow & & \downarrow{\scriptstyle \eta_B} \\ G(A) & \xrightarrow[G(f)]{} & G(B) \end{array}$$

es una transformación natural de F a G, denotada por $\eta : F \longrightarrow G$.

Definición 4.1.14. Si tenemos un funtor $F : \mathbf{A} \longrightarrow \mathbf{B}$ con Obj($\mathbf{A}$) conjunto, decimos que F es un diagrama.

Presentamos algunos ejemplos de funtores a continuación.

Ejemplo 4.1.15. Un complejo filtrado $\{K_i\}_{i\in I}$ es un ejemplo de un funtor $K : \mathbf{I} \longrightarrow \mathbf{AbsSimp}$, considerando I como una categoría como en el Ejemplo 4.1.10.

Ejemplo 4.1.16. Para cada $l \geq 0$, la l-ésima homología de K con coeficientes en $\mathbb{F}$ es un funtor $H_l : \mathbf{AbsSimp} \longrightarrow \mathbf{Vect}_{\mathbb{F}}$.

De los dos ejemplos anteriores, vemos que podemos reinterpretar la homología de un complejo simplicial filtrado simplemente como composición de funtores.

Ejemplo 4.1.17. Sea $X \in \mathbf{Top}$ y sea $f : X \longrightarrow \mathbb{R}$ una función continua. Interpretamos a $\mathbb{R}$ como categoría como en 4.1.10. Definimos un funtor

$$F : \mathbb{R} \longrightarrow \mathbf{Top}$$

dado por

$$F(t) = f^{-1}(-\infty, t]$$

y

$$F(s \leq t) = F(s) \hookrightarrow F(t).$$

Ejemplo 4.1.18. Sea $S \subseteq \mathbb{R}^n$ un conjunto finito. Definimos la función $f : \mathbb{R}^2 \longrightarrow \mathbb{R}$ dada como

$$f(x) = \text{ distancia de } x \text{ a } S,$$

y le aplicamos la construcción del Ejemplo 4.1.17 a f. Para cada $t \in \mathbb{R}^+$, tenemos que $F(t)$ es una cubierta por bolas cerradas de S. Por el teorema del nervio, la realización geométrica del nervio de esta cubierta es un complejo simplicial geométrico del tipo de homotopía de la unión de los elementos de la cubierta. Esto, de manera directa, nos da un funtor $\bar{F}$ que esta vez cae en **AbsSimp**. Además de esto podemos considerar H_l^{sing}, el funtor de l-ésima homología singular con coeficientes en $\mathbb{F}$. Al considerar las composiciones

$$\mathbb{R} \xrightarrow{F} \mathbf{Top} \xrightarrow{H_l^{sing}} \mathbf{Vect}_{\mathbb{F}}$$

y

$$\mathbb{R} \xrightarrow{\bar{F}} \mathbf{AbsSimp} \xrightarrow{H_l} \mathbf{Vect}_{\mathbb{F}},$$

tenemos que son iguales en este caso, que es el de un conjunto finito $S \subseteq \mathbb{R}^n$.

Ejemplo 4.1.19. Consideremos los funtores $\mathrm{Id}, F : \mathbf{FinVect}_{\mathbb{F}} \longrightarrow \mathbf{FinVect}_{\mathbb{F}}$, donde Id denota al funtor identidad y F está dado por

$$F(V) = V^{**}$$

y

$$F(V \xrightarrow{f} W)(\phi) = \psi,$$

donde $\psi(h) = \phi(h \circ f)$. Los funtores F e Id son naturalmente isomorfos.

Observación 4.1.20. Aunque un espacio vectorial de dimensión finita es naturalmente isomorfo con su doble dual, no lo es con su dual, incluso si se toma en cuenta la contravariancia del funtor.

4.2. Casos con índices reales.

Una ventaja de pensar en índices en $\mathbb{R}$ es que diagramas con dominio en $\{0, 1, \ldots, n\}$, $\mathbb{Z}$, o $\mathbb{Z}^+$ pueden ser interpretados como diagramas con índices en $\mathbb{R}$:

Proposición 4.2.1. Sea $F : \mathbf{C} \longrightarrow \mathbf{Top}$ un funtor, con $\mathbf{C} = \{0, 1, \ldots, n\}$, $\mathbb{Z}$, o $\mathbb{Z}^+$. Entonces existen funtores $S(\mathbf{C}) : \mathbf{C} \longrightarrow \mathbb{R}$, $R(\mathbf{C}) : \mathbb{R} \longrightarrow \mathbf{C}$ tales que $FS(\mathbf{C})R(\mathbf{C}) = F$.

Definición 4.2.2. Definimos los l-ésimos grupos de homología p-persistentes, para un complejo simplicial abstracto filtrado $\bar{F}$ sobre $\mathbb{R}$, como la imagen de $H_l\bar{F}(a \leq a + p)$.

Definición 4.2.3. Definimos los l-ésimos grupos de homología p-persistentes de un diagrama $F : \mathbb{R} \longrightarrow \mathbf{Top}$ sobre $\mathbb{R}$, como la imagen de $H_l F(a \leq a + p)$.

También podemos generalizar el teorema de descomposición para módulos f.g. que usamos anteriormente, como sigue:

Definición 4.2.4. Sea $I \subseteq \mathbb{R}$ un intervalo. Definimos el diagrama $\chi_I : \mathbb{R} \longrightarrow \mathbf{FinVect}_{\mathbb{F}}$ como

$$\chi_I(a) = \begin{cases} \mathbb{F} & \text{si } a \in I, \\ 0 & \text{en otro caso,} \end{cases}$$

y

$$\chi_I(a \leq b) = \begin{cases} \mathrm{Id}_{\mathbb{F}} & \text{si } a, b \in I, \\ 0 & \text{en otro caso.} \end{cases}$$

Decimos que $F : \mathbb{R} \longrightarrow \mathbf{FinVect}_{\mathbb{F}}$ tiene tipo finito si $F \cong \bigoplus_{k=1}^{n} \chi_{I_k}$.

La descomposición de los diagramas de tipo finito en $\mathbf{FinVect}_{\mathbb{F}}^{\mathbb{R}}$ es única salvo orden:

Teorema 4.2.5. Si $F \cong \bigoplus_{k=1}^{n} \chi_{I_k}$ y $F \cong \bigoplus_{k=1}^{m} \chi_{I'_k}$ entonces $n = m$ y se tiene $(I_1, \ldots, I_n) = (I'_{\sigma(1)}, \ldots, I'_{\sigma(m)})$ para alguna permutación σ.

Esto nos permite asignar un código de barras a cada diagrama de tipo finito:

Definición 4.2.6. Sea $F \cong \bigoplus_{k=1}^{n} \chi_{I_k} \in \mathbf{FinVect}_{\mathbb{F}}^{\mathbb{R}}$ un diagrama de tipo finito. Definimos su código de barras como el multiconjunto $\{I_k\}_{k=1}^{n}$.

Y, como en el caso discreto, tenemos que los códigos de barras caracterizan a diagramas de tipo finito:

Teorema 4.2.7. Existe una biyección entre clases de isomorfismo de diagramas de tipo finito en $\mathbf{FinVect}_{\mathbb{F}}^{\mathbb{R}}$ y multiconjuntos finitos de intervalos.

Capítulo 5

Una aplicación: análisis de imágenes naturales de alto contraste

En este capítulo desarrollaremos los pasos seguidos en el artículo [**?**]. En ese artículo, se empieza con una base de datos de imágenes naturales en blanco y negro, y se obtienen a partir ellas parches de 3×3 pixeles. El estudio se hace sobre el espacio de todos estos parches, procesados adecuadamente. Todas las porciones de código en este capítulo, a menos que se especifique lo contrario, serán código para Python. Los módulos usados son numpy [10] y matplotlib [6] (y sus dependencias).

5.1. Preparación de la base de datos

La base de datos usada es una base de datos de imágenes naturales en escala de grises con resolución de 1536×1024 pixeles en las que cada pixel usa 16 bits (sin signo, BigEndian, lineal). La base de datos fue compilada por van Hateren y van der Schaaf [4]. Un posible código para visualizar cada imagen es como sigue:

```
import numpy
import array
import matplotlib.pyplot as pplt

fin = open( "D:/Downloads/IMLs/imk00001.iml", 'rb' )
s = fin.read()
fin.close()
arr = array.array('H', s)
arr.byteswap()
img = numpy.array(arr, dtype='uint16').reshape(1024,1536)
pplt.imshow(img,cmap='Greys_r' , interpolation='nearest')
pplt.show()
```

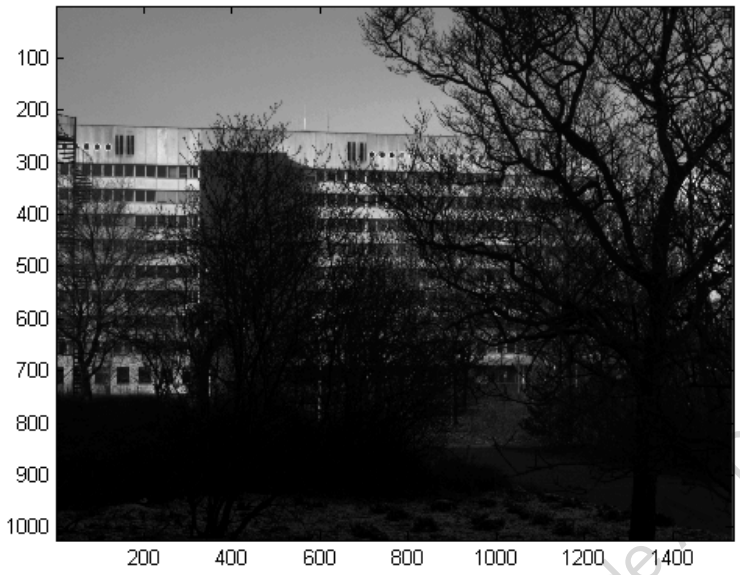


Figura 5.1: Ejemplo de salida para el archivo imk00001.iml.

Por supuesto, no es necesario visualizar en cada paso los resultados obtenidos, pero ayuda a una mejor comprensión del proceso.

Cada parche de 3×3 es considerado como un vector en $\mathbb{R}^9$, y sacamos el logaritmo de cada entrada. Trabajaremos con este nuevo vector de $\mathbb{R}^9$. Además de esto, trasladamos los vectores de tal manera que todos satisfagan que sus entradas tengan media cero. La razón para hacer esto es porque en principio no nos interesan parches que pueden ser obtenidos a partir de otros sumando una constante a todas las entradas (lo que es básicamente modificar el brillo del parche).

Parche 1 Parche 2

Figura 5.2: El parche 2 es obtenido al subir el brillo del parche 1 (suma de una misma constante a todas las entradas).

Para extraer y procesar de esta manera un parche de 3×3 pixeles podemos usar:

```
def fparche(x, y, image):
        """
        Extracts and returns a 9-vector with logarithms of a 3x3 patch
        from an image with parameters:

        x = x coordinate of the upper left corner of the patch to
            compute
        y = y coordinate of the upper left corner of the patch to
            compute
        image = image from which we extract the patch
        """
        patch = image[x:x + 3, y:y + 3]
        try:
                vector = np.vectorize(math.log)(patch)
        except:
                print "Problem with vector \n" + str(patch) + "\n"
                vector = np.array([[0, 0, 0], [0, 0, 0], [0, 0, 0]])
        vector = vector - vector.mean()
        return np.ndarray.flatten(np.reshape(vector, (1, 9)))
```

Extraeremos, para cada imagen, 5000 parches y descartaremos algunos.

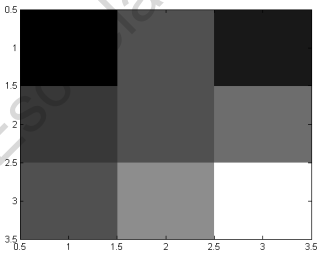


Figura 5.3: Ejemplo de salida para el archivo imk00001.iml con coordenadas $(1471, 809)$
.

Definición 5.1.1 (Norma D)**.** Sea $x \in \mathbb{R}^9$. Definimos la norma D de x como

$$\|x\|_D = \sqrt{\sum_{i \sim j} (x_i - x_j)^2},$$

donde $i \sim j$ si y sólo si son adyacentes horizontalmente o verticalmente (la suma se hace sin repeticiones).

Proposición 5.1.2. $\|x\|_D = \sqrt{xDx^T}$, donde

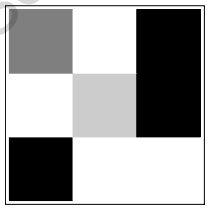
$$D = \begin{pmatrix} 2 & -1 & 0 & -1 & 0 & 0 & 0 & 0 & 0 \\ -1 & 3 & -1 & 0 & -1 & 0 & 0 & 0 & 0 \\ 0 & -1 & 2 & 0 & 0 & -1 & 0 & 0 & 0 \\ -1 & 0 & 0 & 3 & -1 & 0 & -1 & 0 & 0 \\ 0 & -1 & 0 & -1 & 4 & -1 & 0 & -1 & 0 \\ 0 & 0 & -1 & 0 & -1 & 3 & 0 & 0 & -1 \\ 0 & 0 & 0 & -1 & 0 & 0 & 2 & -1 & 0 \\ 0 & 0 & 0 & 0 & -1 & 0 & -1 & 3 & -1 \\ 0 & 0 & 0 & 0 & 0 & -1 & 0 & -1 & 2 \end{pmatrix}.$$

Podemos implementar una función que obtenga la norma D de un vector (fila) de la siguiente manera:

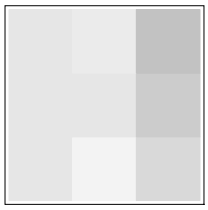
```
def normaD(x):
        D = np.array([[2, -1, 0, -1, 0, 0, 0, 0, 0], \
            [-1, 3, -1, 0, -1, 0, 0, 0, 0], \
            [0, -1, 2, 0, 0, -1, 0, 0, 0], \
            [-1, 0, 0, 3, -1, 0, -1, 0, 0], \
            [0, -1, 0, -1, 4, -1, 0, -1, 0], \
            [0, 0, -1, 0, -1, 3, 0, 0, -1], \
            [0, 0, 0, -1, 0, 0, 2, -1, 0], \
            [0, 0, 0, 0, -1, 0, -1, 3, -1], \
            [0, 0, 0, 0, 0, -1, 0, -1, 2]])
        return math.sqrt(np.dot(x, np.dot(D, (x.transpose()))))
```

Por ejemplo, la norma D del parche de la Figura 5.3 es 87,8977.

La norma D básicamente mide el contraste de un parche. Parches con alto constrate tendrán norma D mayor que parches con menos contraste. Nosotros estamos interesados en parches de alto contraste, puesto que contienen la mayor parte de la información de las imágenes:



Alto contraste

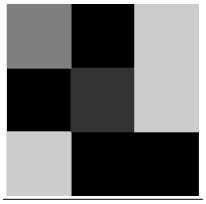


Bajo contraste

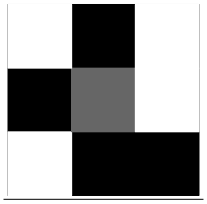
Figura 5.4: Ejemplos de parches.

Después de este paso, quitamos 80 % de parches, correspondientes a los que hayan tenido la menor norma D. Para hacer esto de manera directa, primero ordenamos los parches de

manera ascendente por su valor en norma D, después seleccionamos los 1000 vectores y los normalizamos respecto a la norma D. La razón para normalizar los vectores es porque no nos interesan parches que pueden ser obtenidos a partir de multiplicación por escalares positivos de otros parches (lo que es básicamente modificar el contraste del parche).



Parche 1



Parche 2

Figura 5.5: El parche 2 es obtenido al subir el contraste del parche 1 (multiplicación por escalar).

El código resultante es el siguiente:

```
def parches5000(filename):
        """
        Generates and returns a collection of 1000 high-contrast
           patches interpreted
        as normalized 9-vectors with respect to the D-norm. The
           selection
        of the patches is random. Parameters:

        filename = name of the file from where to generate the
           collection
                                of patches
        """
        print "Processing file: " + filename
        vectores = np.zeros((5000, 9))
        normas = np.zeros((5000, 2))
        x = np.array(range(1534))
        y = np.array(range(1022))
        xy = cartesian_product2((x, y))
        rxy = np.array(random.sample(xy, 5000))
        with open(filename, 'rb') as handle:
                s = handle.read()
        arr = array.array('H', s)
        arr.byteswap()
        img = np.array(arr, dtype='uint16').reshape(1024, 1536)
        for i in xrange(5000):
                vectores[i, :] = fparche(rxy[i][1], rxy[i][0], img)
                normas[i, 0] = normaD(vectores[i, :])
                normas[i, 1] = i
        normas = normas[normas[:, 0].argsort()]
```

```
        parchesaltocontraste = np.zeros((1000, 9))
        for i in xrange(1000):
                x = vectores[normas[4000 + i, 1], :]
                # print str(normaD(x))
                try:
                        x = np.inner(x, 1 / normaD(x))
                except:
                        print "This patch has no contrast.\n"
                parchesaltocontraste[i, :] = x
        return parchesaltocontraste
```

Recapitulando, lo que hemos hecho es tomar vectores en $\mathbb{R}^9$, aplicarles logaritmo, centrarnos en parches que tienen medida cero, remover 80 % de ellos y normalizarlos respecto a la norma D. Este proceso nos ha dejado con vectores que dependen de 7 variables. A continuación lo que haremos será efectuar un cambio de base para que estos vectores estén en una variedad de dimensión 7, a saber: $S^7 \subseteq \mathbb{R}^8$.

Los elementos del arreglo `DCTbasis` son calculados a partir de la transformada discreta en cosenos en dos dimensiones, que definimos a continuación:

Definición 5.1.3. Dado un parche de 3×3 pixeles x, definimos su DCT en 2D X como

$$X_{k_1,k_2} = \sum_{n_1=0}^{N_1-1} \sum_{n_2=0}^{N_2-1} x_{n_1,n_2} \cos\left[\frac{\pi}{N_1}\left(n_1 + \frac{1}{2}\right) k_1\right] \cos\left[\frac{\pi}{N_2}\left(n_2 + \frac{1}{2}\right) k_2\right]. \tag{5.1}$$

En este caso, cada elemento de `DCTbasis` es obtenido al fijar k_1 y k_2, y dejando variar n_1 y n_2 en el rango apropiado. Esto nos arroja 9 vectores de 9 entradas cada uno, con uno de los 9 vectores constante no tomado en cuenta. De manera explícita, al ejecutar una implementación de esta fórmula y desplegando los arreglos como filas obtenemos:

```
X_1 =
1   1   1   1   1   1   1   1   1
X_2 =
0.8660  0.8660  0.8660  0.0000  0.0000  0.0000 -0.8660 -0.8660 -0.8660
X_3 =
0.5000  0.5000  0.5000 -1.0000 -1.0000 -1.0000  0.5000  0.5000  0.5000
X_4 =
0.8660  0.0000 -0.8660  0.8660  0.0000 -0.8660  0.8660  0.0000 -0.8660
X_5 =
0.7500  0.0000 -0.7500  0.0000  0.0000 -0.0000 -0.7500 -0.0000  0.7500
X_6 =
0.4330  0.0000 -0.4330 -0.8660 -0.0000  0.8660  0.4330  0.0000 -0.4330
X_7 =
0.5000 -1.0000  0.5000  0.5000 -1.0000  0.5000  0.5000 -1.0000  0.5000
```

```
X_8 =
0.4330 -0.8660  0.4330  0.0000 -0.0000  0.0000 -0.4330  0.8660 -0.4330
X_9 =
0.2500 -0.5000  0.2500 -0.5000  1.0000 -0.5000  0.2500 -0.5000  0.2500
```

Comparado con `DCTbasis`, vemos que faltan sólo multiplicaciones por escalares y reordenamiento de filas:

```
DCTbasis[0, :] =
np.inner(1 / math.sqrt(6), [1, 0, -1, 1, 0, -1, 1, 0, -1])
DCTbasis[1, :] =
np.inner(1 / math.sqrt(6), [1, 1, 1, 0, 0, 0, -1, -1, -1])
DCTbasis[2, :] =
np.inner(1 / math.sqrt(54), [1, -2, 1, 1, -2, 1, 1, -2, 1])
DCTbasis[3, :] =
np.inner(1 / math.sqrt(54), [1, 1, 1, -2, -2, -2, 1, 1, 1])
DCTbasis[4, :] =
np.inner(1 / math.sqrt(8), [1, 0, -1, 0, 0, 0, -1, 0, 1])
DCTbasis[5, :] =
np.inner(1 / math.sqrt(48), [1, 0, -1, -2, 0, 2, 1, 0, -1])
DCTbasis[6, :] =
np.inner(1 / math.sqrt(48), [1, -2, 1, 0, 0, 0, -1, 2, -1])
DCTbasis[7, :] =
np.inner(1 / math.sqrt(216), [1, -2, 1, -2, 4, -2, 1, -2, 1])
```

Los vectores de `DCTbasis` son eigenvectores de la matriz D, con eigenvalores dados por

$$\frac{1}{\|\texttt{DCTbasis[i,:]}\|^2},$$

para $0 = 1, \ldots, 7$. El código

```
def generaparche(path, filename):
        """
        Converts and returns 1000 9-vectors to 8-vectors linearly using
            the DCT
        basis and a helper matrix G. Parameters:

        path = path to file containing the vectors
        filename = name of the file containing the vectors
        """
        temp = parches5000(path  + filename)
        result = np.zeros((1000, 8))
```

```
        for j in xrange(1000):
                result[j, :] = np.dot(G, np.dot(DCTbasis, (temp[j, :].
                    transpose())))
        return result
```

manda los vectores `DCTbasis` en los vectores de la base estándar. Más aún, tenemos que

$$DCTbasis^t \cdot G^2 \cdot DCTbasis = D,$$

por lo que, para x tal que $\|x\|_D = 1$, tenemos

$$\begin{aligned} \|G \cdot DCTbasis \cdot x^t\|^2 &= (G \cdot DCTbasis \cdot x)^t \cdot G \cdot DCTbasis \cdot x^t \\ &= x \cdot DCTbasis^t \cdot G^2 \cdot DCTbasis \cdot x^t \\ &= x \cdot D \cdot x^t \\ &= 1. \end{aligned}$$

Esto es, el código anterior manda x con $\|x\|_D = 1$ a un vector en $S^7 \subseteq \mathbb{R}^8$, como deseábamos.

La razón por la cual descartamos al primer vector de la transformada discreta en cosenos, y la razón por la que el resultado yace en $\mathbb{R}^8$, es porque si denotamos por $\overline{DCTbasis}$ a la matriz de los nueve vectores, tenemos que para x con media cero (que es el caso para los que estamos trabajando) el vector $\overline{DCTbasis} \cdot x^t$ siempre tiene primera entrada cero.

Denotaremos al espacio de todos los parches procesados de esta manera como $\mathcal{M}$. Aunque estamos interesados en todo $\mathcal{M}$, resulta que $\mathcal{M}$ rellena bastante bien a S^7, por lo que su homología persistente más que nada nos daría la homología de S^7. En lugar de considerar entonces todo $\mathcal{M}$, podemos fijarnos en subconjuntos densamente poblados, variando el parámetro de densidad, y estudiando su homología persistente. Tenemos la siguiente definición:

Definición 5.1.4. Sean $x \in \mathcal{M}$, $k \in \mathbb{Z}^+$. Definimos la codensidad con parámetro k de x en $\mathcal{M}$ como

$$\delta_k(x) = d(x, x') \qquad \text{con } x' \text{ el } k\text{-ésimo vecino cercano a } x.$$

Observación 5.1.5. Esta definición tiene sentido para cualquier espacio métrico finito.

Observación 5.1.6. $\delta_k(x)$ varía de manera inversa respecto a la densidad de x en $\mathcal{M}$.

Figura 5.6: $\delta_5(x_1) < \delta_5(x_2)$: los primeros 5 vecinos cercanos a x_1 están más cerca que los primeros 5 vecinos cercanos a x_2, esto es, x_1 es más denso que x_2.

En [1] se trabaja con una subcolección X de 50000 seleccionados aleatoriamente de $\mathcal{M}$. Notemos que la misma objeción que había a aplicar directamente homología persistente a $\mathcal{M}$ aplica a X, pues la elección es aleatoria, y se estudia la homología persistente de subconjuntos densos en X respecto a la codensidad de la Definición 5.1.4. Ponemos un poco de notación.

Definición 5.1.7. Sea M un espacio métrico finito. Definimos

$$M(k,T) = \{x \in M \mid \delta_k(x) \text{ está en los } T\,\% \text{ menores valores de } \delta_k \text{ en } M\}$$

En estas notas primero haremos un estudio de $\mathcal{M}(15,30)$. Para realizar un análisis en una base de datos de tamaño tan grande (aprox. 4×10^6 elementos), numerosas optimizaciones y consideraciones técnicas debieron ser hechas. Para calcular vecinos k-ésimos, usamos la implementación de KDTrees encontrada en el paquete Scikit-learn [8]. La función principal para obtener los vecinos es:

```
def vecinos(k, start, end, X, kdt):
    """
    This is the main computing function for k-th neighbor distance.
    The parameters start and end are required for granularity.
    Parameters:
    X:     the space of patches
    k:     is the number of closest neighbor
    kdt:   KDTree structure
    """
    temp = kdt.query(X[start:end], k=k)
    return [item[-1] for item in temp[0]]   # some juggling is required
        to get the information we want
```

En principio, esto puede ser llamado de manera secuencial una y otra vez hasta obtener todas las distancias y los elementos que tengan la densidad deseada. Sin embargo, se puede paralelizar fácilmente todo este proceso pues el valor de δ_l es independiente para cada elemento. Las siguientes dos funciones obtienen los parches ordenados por codensidad usando varios (ocho) núcleos.

```
def vecinosgrande(k, pos, start, end, output, X, kdt):
    """
    This function was made for granularity in multiprocessing and
       avoiding running out of memory
    """
    Y = []
    for i in xrange(int((end - start) / 1000) + 1):
            print "Worker " + str(pos) + " processing: [" + str(start +
                i * 1000)\
```

```
                    + " a " + str(min(start + i * 1000 + 1000, end)) +
                        ").\n"
            Y += vecinos(k,
                         start + i * 1000,
                         min(start + i * 1000 + 1000, end),
                         X,
                         kdt)
    output.put((pos, Y))
    print("Worker " + str(pos) + " done.\n")

def init(fullfilename):
    Xprime = np.load(fullfilename)
    starttime = time.time()
    kdt = KDTree(Xprime, leaf_size=10)
    print("Time elapsed for tree generation: %s seconds.\n" % (time.
       time() - starttime))
    return Xprime, kdt

def kvecinosmp(k,pathsource,filenamesource,path,filename):
    print "Computing distances to k-neighbor with k = " + str(k) + ".\n
       "
    X, kdt = init(pathsource + filenamesource)
    N = len(X)
    out = []  # output array for the distances, ordered by index
              # (i.e. out[0] is the distance to the k-th neighbor of
                 patches[0])
    num_processes = 8
    starttime = time.time()
    output = mp.Queue()
    processes = [mp.Process(target=vecinosgrande,
                            args=(k, i, int(i * N / num_processes),
                                  int((i + 1) * N / num_processes),
                                  output,
                                  X,
                                  kdt))
                 for i in xrange(num_processes)]
    for p in processes:
        p.start()
    Y = [output.get() for p in processes]
    for p in processes:
        p.join()
    print("Time elapsed for full computation: %s segundos.\n" % (time.
       time() - starttime))
    Y.sort(key=lambda x:x[0])  # sort by worker to preserve original
       ordering
    for x in Y:
        out.extend(x[1])
    np.save(path + filename, out)  # saves the distances
```

Se consideró, basándose en sugerencias aportadas en un seminario, usar $\|\cdot\|_1$ en lugar de $\|\cdot\|_2$ para acelerar un poco el proceso, pero no se encontraron ganancias significativas.

5.2. Complejos de Testigos

Para estudiar la homología persistente de los datos obtenidos en la sección anterior, usamos JAVAPLEX. Usaremos una variante de Python llamada Jython para tener compatibilidad con JAVAPLEX. Para entender un poco más de cómo es que el código obtiene la homología persistente a partir de los datos, daremos un breve repaso acerca de complejos simpliciales llamados complejos de testigos.

Definición 5.2.1. Sea X un espacio métrico, $L \subseteq X$ un subespacio finito, y $\epsilon > 0$. Definimos el complejo de testigos de L con parámetro ϵ como

$$W(X, L, \epsilon) = \{\{l_0, \ldots, l_k\} \subseteq L \,|\, \exists\, x \in X \text{ tal que } d(x, l_i) \leq d(x, L) + \epsilon \text{ para } i = 0, \ldots, k\}$$

Existe una versión "floja" de este complejo de testigos, que es la siguiente:

Definición 5.2.2. Sea X un espacio métrico, $L \subseteq X$ un subespacio finito, y $\epsilon > 0$. Definimos el complejo de testigos flojo de L con parámetro ϵ como:

$$W_{VR}(X, L, \epsilon) = \{\{l_0, \ldots, l_k\} \subseteq L \,|\, \{l_i, l_j\} \in W(X, L, \epsilon) \quad \forall\, i, j \in \{0, \ldots, k\}\}$$

Una ventaja de trabajar con complejos de testigos es que, usando el conjunto L de puntos de referencia (usualmente mucho menor que X), podemos obtener más rápido la homología persistente de los datos X.

Proposición 5.2.3. Si $\epsilon_1 \leq \epsilon_2$, entonces $W_{VR}(X, L, \epsilon_1) \subseteq W_{VR}(X, L, \epsilon_2)$.

Observación 5.2.4. En vista de la Proposición 5.2.3, tiene sentido hablar de la homología persistente de un complejo filtrado sobre el parámetro ϵ. La elección de trabajar con complejos de testigos es una decisión práctica: en principio nada nos impediría considerar el complejo de Vietoris-Rips o el Čech de nuestra nube de datos, aparte de que los cálculos tomarían mucho más tiempo.

Observación 5.2.5. La construcción de JAVAPLEX usada para complejos de testigos es ligeramente distinta a esta. Detalles adicionales pueden encontrarse en las secciones 5.3 y 5.4 de la documentación de JAVAPLEX o en [1]

5.3. Resultados Experimentales

Para calcular la homología persistente usamos el siguiente código, adaptado del código que viene en el archivo `optical_image_example.m`, incluido en los ejemplos de JAVAPLEX, ajustando parámetros dependiendo del caso de las variables k y T y del número total de parches. Esta vez se hace uso de Jython. Primero obtenemos los valores de los intervalos de persistencia:

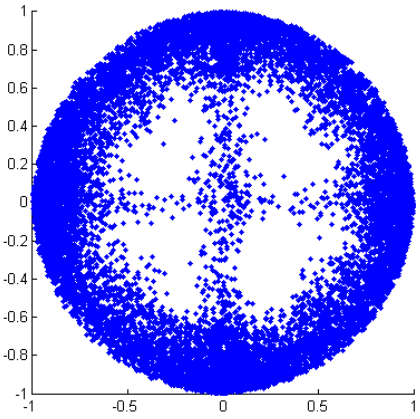
```
from edu.stanford.math.plex4 import *
import cPickle as cp
from java.lang import Runtime

def barcodes(k,t,pathsource,filenamesource,path,filename):
    max_dimension = 3;
    num_landmark_points = 50;
    nu = 1;
    num_divisions = 100;
    Dintervals = []
    parches = cp.load(open(pathsource + filenamesource, 'rb'))
    print "Creating maxmin landmark selector"
    print "Total available memory: " + str(Runtime.getRuntime().
       totalMemory())
    # create a sequential maxmin landmark selector
    landmark_selector = api.Plex4.createMaxMinSelector(parches,
       num_landmark_points);
    #R = landmark_selector.getMaxDistanceFromPointsToLandmarks()
    max_filtration_value = .3  # R*.7;

    print "Constructing lazy witness stream"
    # create a lazy witness stream
    stream = streams.impl.LazyWitnessStream(landmark_selector.
       getUnderlyingMetricSpace(),
                                            landmark_selector,
                                            max_dimension,
                                            max_filtration_value,
                                            nu,
                                            num_divisions);
    stream.finalizeStream()
    print "Lazy witness stream construction complete"
    # print out the size of the stream
    #num_simplices = stream.getSize()

    # get persistence algorithm over Z/2Z
    persistence = api.Plex4.getModularSimplicialAlgorithm(max_dimension
       , 2);
    print "Computing intervals"
    # compute the intervals
    intervals = persistence.computeIntervals(stream);

    # process and pickle the intervals
    for dimension in xrange(max_dimension):
        temp = []                                                #
           collection of intervals at dimension
        temp0 = intervals.getIntervalsAtDimension(dimension)
        for ninterval in xrange(len(temp0)):
            temp.append([temp0[ninterval].getStart(),
```

```
                        temp0[ninterval].getEnd()])            #
                           intervals as pairs [start,end]
            Dintervals.append([dimension, temp])                    # adds
               dimension information for the collection of intervals

        #=========================================================================

        # Dintervals are in the format:
        # [[dim_1,listofintervalsofdim_1],...,[dim_k,listofintervalsofdim_k
           ]]
        #=========================================================================

        cp.dump(Dintervals, open(path + filename, 'wb'))
        print "Intervals saved."
```

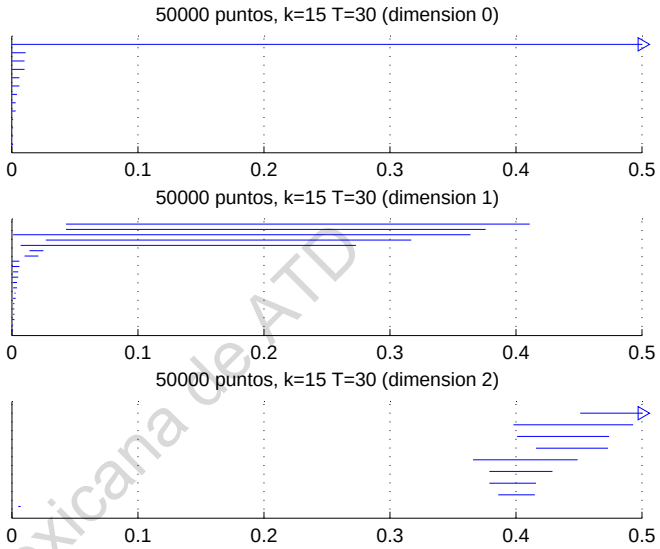
```
import matplotlib.pyplot as plt
import numpy as np
import cPickle as cp


if __name__ == "__main__":
    k = 300
    T = 0.3
    x = cp.load(open('barcodek' + str(k) + 'T' + str(T) + '.p', 'rb'))
    x = np.asanyarray(x)
    MAX_X = .8
    plt.figure(1)
    for i in xrange(len(x)):
        ax = plt.subplot(len(x), 1, i + 1)
        ax.set_title('Dimension '+str(i))
        ax.margins(0, .2)                    # padding in x and y axis,
           respectively
        ax.set_xlim(0, MAX_X)                # limits of graph in axis x
        ax.set_yticks([])                    # removes ticks and labels
           along the y-axis
        for j in xrange(len(x[i][1])):
            print str(i) + ": " + str(x[i][1][j])
            plt.plot(x[i][1][j], [j * .1 + 1, j * .1 + 1])
            if x[i][1][j][1] == None:
                plt.plot([x[i][1][j][0], MAX_X], [j * .1 + 1, j * .1 +
                   1])
    plt.suptitle("Barcodes for k = " + str(k) + " and T = " + str(T))
    plt.savefig('barcodek' + str(k) + 'T' + str(T) + '.png')
    plt.tight_layout()
    plt.subplots_adjust(top=.89)
    plt.show()
```

Esto genera una imagen con el código de barras. A continuación presentamos los códigos de barras obtenidos al aplicar el código a diversos casos.

Caso 1. Aplicamos el procedimiento con los parámetros $k = 15$ y $T = 30$ a 50000 parches elegidos aleatoriamente de los 4167000.

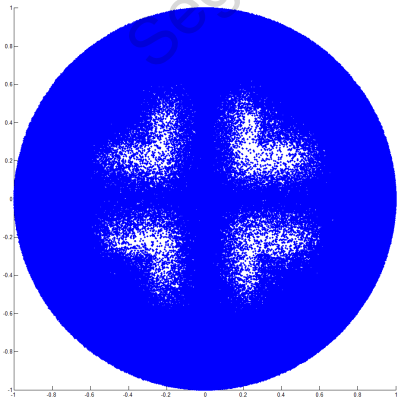


Proyección de la nube

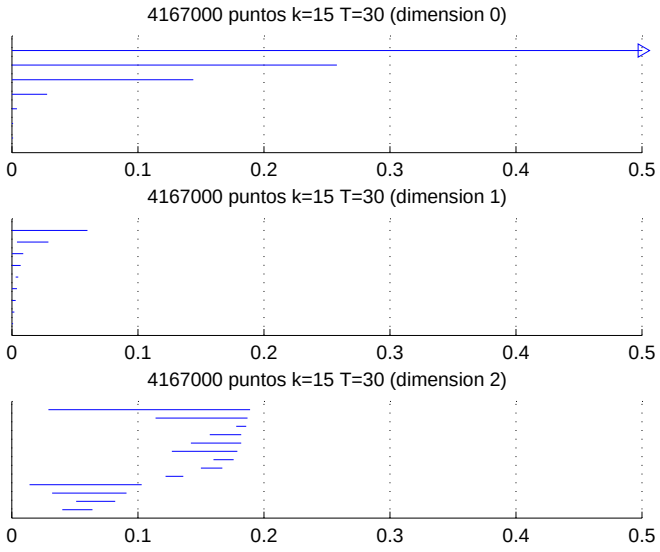


Código de barras

Caso 2. Aplicamos el procedimiento con los parámetros $k = 15$ y $T = 30$ al conjunto de todos los parches, obteniendo:

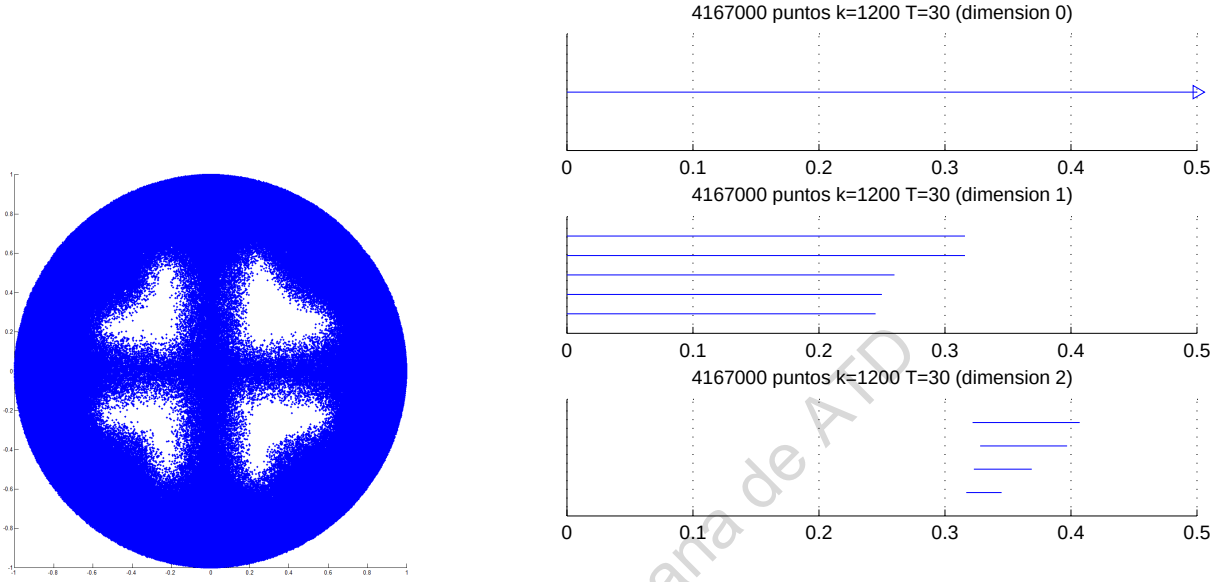


Proyección de la nube



Código de barras

Caso 3. Aplicamos el procedimiento con los parámetros $k = 1200$ y $T = 30$ al conjunto de todos los parches, obteniendo (cf. Figura 7 de [**?**]):



Proyección de la nube

Código de barras

Figura 5.9

Bibliografía

[1] G. Carlsson and V. de Silva. Topological estimation using witness complexes.

[2] Gunnar E. Carlsson, Tigran Ishkhanov, Vin de Silva, and Afra Zomorodian. On the local behavior of spaces of natural images. *International Journal of Computer Vision*, 76(1):1–12, 2008.

[3] H. Edelsbrunner and J. Harer. *Computational Topology: An Introduction*.

[4] J. H. van Hateren and A. van der Schaaf. Independent component filters of natural images compared with simple cells in primary visual cortex. *Proceedings: Biological Sciences*, 265(1394):359–366, Mar 1998.

[5] K. Hoffman and R. Kunze. *Linear Algebra*.

[6] J.D. Hunter. Matplotlib: A 2d graphics environment. *Computing in Science Engineering*, 9(3):90–95, May 2007.

[7] D. Letscher. On persistent homotopy, knotted complexes and the alexander module.

[8] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

[9] Andrew Tausz, Mikael Vejdemo-Johansson, and Henry Adams. JavaPlex: A research software package for persistent (co)homology. In Han Hong and Chee Yap, editors, *Proceedings of ICMS 2014*, Lecture Notes in Computer Science 8592, pages 129–136, 2014.

[10] S. van der Walt, S.C. Colbert, and G. Varoquaux. The numpy array: A structure for efficient numerical computation. *Computing in Science Engineering*, 13(2):22–30, March 2011.