



**UNIVERSIDAD DE BUENOS AIRES**  
**Facultad de Ciencias Exactas y Naturales**  
**Departamento de Matemática**

**Tesis de Licenciatura**

**Evolute EigenGame: Resolviendo PCA con un algoritmo evolutivo**

**Ezequiel Diego Criboli**

**Director: Dr. Juan Pablo Pinasco**

**Jurado: Dr. Gabriel Acosta y Dr. Pablo Groisman**

17 de agosto de 2022



UNIVERSIDAD DE BUENOS AIRES



# Evolutive EigenGame: Resolviendo PCA con un algoritmo evolutivo

TESIS

QUE PARA OBTENER EL TÍTULO DE

LICENCIADO EN CIENCIAS MATEMÁTICAS

PRESENTA

EZEQUIEL DIEGO CRIBIOLI

BAJO LA DIRECCIÓN DE

JUAN PABLO PINASCO

BUENOS AIRES, ARGENTINA

17 DE AGOSTO DE 2022



*Dedicado a mi abuela Marta y a la Tía Mili*



# Agradecimientos

El primer agradecimiento va para mi vieja, que siempre fue mi pilar fundamental cuando me chocaba contra una pared, y siempre estuvo al lado mío para aconsejarme y vivir cada pasito de este viaje conmigo. Sé que estabas esperando esta tesis incluso más que yo, así que esto es para vos, ma.

La segunda persona que quiero mencionar es a mi primer mentor, que ya sabe todo lo que voy a decir y todo lo que significa para mí, pero quiero que quede este registro. Cuando, a mis 13 años, Agustín Barreto entro a mi aula para promocionar las olimpiadas de matemática, nunca hubiera imaginado lo que esa decisión me iba a cambiar la vida. Y que me haya contagiado ese amor por la matemática y entrenado por 4 años, solo por haber visto una chispa en mí, definitivamente me hizo estar donde estoy hoy. Eso sin mencionar que sigo aprendiendo de él hasta hoy en día, tanto dentro como fuera de la matemática. Gracias por ser un gran amigo Agus.

También le debo un gran agradecimiento a Juan Pablo, por ser el mejor director de tesis que podría haber pedido, por ser tan abierto conmigo y por demostrarme ser una gran persona. Me hubiera encantado juntarnos a charlar más seguido, pero estoy seguro de que nos quedan un montón de cafés por charlar sin una tesis de por medio.

A mi viejo, por ser tan incondicional y acompañarme a donde sea siempre que lo necesité. Por interesarse por mis cosas y por comprenderme a mí y a un mundo tan extraño para él.

A mis hermanos, Nacho y Agus, por siempre saber como hacerme reír y por ser dos inmensos hermanos mayores.

A mis abuelas, por cada comida, por cada vez que cuidaron de mí y cada enseñanza que me dejaron.

A todas las personas involucradas en la organización de la Olimpiada Matemática Argentina. Entre ellas, Patricia Fauring, Flora Gutierrez, Veronica Sunkel, Gabriel Estrany, Silvia Mamone, Matias Saucedo y a todos los exolímpicos, jurados y secretarios. Sepan que la olimpiada me cambio la vida y siempre voy a estar agradecido con todos ustedes.

Pero la Olimpiada es lo que es gracias a la comunidad, por eso quiero agradecer a todas las personas que me dieron la oportunidad de ser su amigo. Me encantaría decirle a cada uno todo lo que les agradezco estos años, pero este texto sería eterno. Estoy seguro de que me voy a olvidar de alguien, pero voy a intentar nombrarlos a todos: A Trivial (Buso, Euge, Chenna, Fer, Caro, Ivan, Meli y Nati), Marchi, Masli, Deamo, Tute y Lalo Rinaldo, Nico y Juli Ferres, Bat, Nico Melgar, Vitu, Facu y Marti Alvarez Motta, Franco Bongiovanni, Santi Cubino, Marcos y Pedro Van Keulen, Lichu, Nico Schaievitch, Sandy, Joa, Santi Clemente, Charo, Fiebre, Mateo y Javi Carranza Velez, Ian, Charles, Turko, Mono, Carla, Fleury, Yami, Julia Conde, Ghersi, Luca Martini, Santiago Blasco, Ale Candioti, Agustin Sansone, Cami Garcia, Bruno di Sanzo, Mati Bergerman, Tima y Luigi.

Tengo que hacer un párrafo aparte para dos de mis mejores amigos hoy en día, Bruno Giordano y Juli Garbulsky, que además de ser de las personas que más me hacen reír, son unos grandes y apasionados matemáticos.

De la misma forma tengo que mencionar a Sebastian Cherny y a Gaston Salgado, dos hermanos que desde que me adoptaron hace casi 2 años no paran de demostrar que tienen un corazón gigante. Que me hayan invitado a su equipo también me cambio la vida aunque ellos no se den cuenta.

A mis amigos uruguayos, Lu, Mateo, Franco, Marti, Magy y a toda la familia Fernandez Reyes, por siempre estar ahí y por hacerme sentir que tengo una casa del otro lado del río.

A todos los amigos que hice en la facultad, por hacer de estos 4 años una experiencia todavía más hermosa. Chino, Leo, Gabi, Nico, Marian, Pau, Jan, Mati Avellaneda, Mati Sandacz, Julian Zylber, Gabo Szwarcberg, Guido, Maki, y por supuesto a mi querido CALI (Ale, Dina, Agoff, Joel, Franco, Santi y Juli).

A mi hermosa comunidad Kimlu, donde conocí personas maravillosas, algunas de ellas me acompañan hasta el día de hoy: Estefi, Joules, Fermin, Edu Pavez, Dylan Fridman, Facu Molina, Francois, Eddu Guzman, Tanke,

Vane, Benja, Banje, las terremoto, Javi Silva x2, Cami Farias, Martin Flores, Joaquin Bocaz, Cata Tapia, Cata Tamburrino, Cata Silva, Cota, Coti, Vicky Pereira, Dikran, Gabo Pizarro, Ana Stumpf, Jose Barquin, Juani Martony, Majo Mendoza, Martin Palisson, Rocio Rivas, Chiara, Joaco Zur, Cristi Haug, Anita Donet, Cami Pinat, Cami Reynoso, Enzo, Pipi y Clari Mise.

A todos los amigos de ExpC que hice en la comunidad exogámica, por hacerme sentir parte y ser siempre re buena onda conmigo: Santi, Frodo y Rena Di Tullio, Emma, Axel Fridman, Seba de Lellis, Sere, ari, Braude, Manu Fernandez, Dani Wappner, Fran Cuello, Male Verduga, Karen, Sol Hartz, Agus PR, Lenka, Santi Stalder, Lu Arballo, Juani Iribarren, Lu Gotuzzo y Fede Santana.

A los grandes profesores que tuve la suerte de tener, y que fueron parte indispensable de este camino y de hacerme disfrutarlo tanto: Patu, Ezequiel Rela, Javier Marenco, Julio Rossi, Fer Martin, Dario Aza, Juan Francisco Piombo, Marcelo Valdettaro, Martin Blufstein, Gabriel Minian, Gabriela Jeronimo, Teresa Krick, Pablo Blanc, Ivan Rey, Mariano Chehebar, Sebastian Velazquez, Kevin Piterman, Juan Jose Guccione, Carolina Mosquera, Esteban Andruschow, Ariel Salort.

A los amigos nerds de mi mamá, Fernando Beitia, Sebastian Kuljis y Nancy Jorge, por confirmarme que este era el camino para mí, por alentarme siempre y por vivirlo conmigo.

A Tomas Schitter, que me ayudo en un momento en que lo necesitaba mucho, casi sin conocerme y de forma totalmente desinteresada. No sé que será de su vida, pero si estás leyendo esto: Muchas gracias Tomi.

A Guido de la Dirección de Estudiantes por ser empático, humilde, buen labrador.

A Ricardo Malmoria, por organizar ese taller de olimpiadas en el colegio y por acompañarme en mis primeras aventuras olímpicas.

A los maravillosos alumnos a los que tuve la posibilidad de devolverles un poquito de todo lo que la matemática me dio en el camino. Maia, Crovara, MacKenzie, Arroyo, Ledesma, Hofferle, Russo, Lategana, Ormaechea, Zancai y Roberts. Y a Marcela Rosas por darme la oportunidad de enseñar.

Y quiero cerrar esta hermosa sección con una frase bien corta y contundente pero que me representa mucho: Aguante la Universidad Pública.



# Introducción

El Machine Learning es una rama de la Inteligencia Artificial que desarrolla modos en que las máquinas aprenden a predecir resultados y tomar sus propias decisiones basadas en datos. A través del Machine Learning, los equipos informáticos son capaces de mejorar procesos aprendiendo de su propia experiencia y de los datos introducidos. De este modo, perfeccionan y facilitan cualquier proceso sin haber sido específicamente programados para hacerlo. Estos sistemas, en otras palabras, automatizan procesos y eliminan la necesidad de que intervenga un humano para dar instrucciones concretas a la máquina.

Este campo está ganando cada vez más terreno en cuanto a investigación y aplicaciones respecta. Cada año que pasa se ven nuevas aplicaciones y nuevos métodos para lograr una mayor capacidad de realizar tareas en distintos ámbitos. Estas aplicaciones llegan a áreas de todo tipo, como pueden ser las finanzas, la medicina, las comunicaciones y los transportes, o el marketing y la publicidad, entre muchas otras.

Las principales aplicaciones de Machine Learning tienen que ver con el análisis del Big Data, una tarea que sería inabordable por humanos y que los sistemas informáticos pueden no obstante ejecutar de forma rápida. Pero para esto, se necesita siempre tener presente un buen manejo de los datos para no desaprovechar el poder de cómputo que las máquinas nos ofrecen. Es por eso que un área con mucha relevancia en este contexto es el de las técnicas de reducción de dimensionalidad.

Por ejemplo, si trabajamos con fotos en resolución 1080*p*, que ni siquiera es la calidad más alta disponible hoy en día, quiere decir que cada foto consta de  $1920 \times 1080 = 2,073,600$  píxeles. A su vez, cada pixel es una tira de 3 números (un número entre 0 y 255 para representar el valor del rojo,

uno para el verde y otro para el azul si estamos usando escala RGB), o sea que cada foto sería una tira de  $3 \times 2,073,600 = 6,220,800$  números, un número de dimensión bastante complicado para trabajar si no se procesan los datos de forma inteligente antes.

En este trabajo, vamos a utilizar la técnica llamada Análisis de Componentes Principales, PCA por sus siglas en inglés (Principal Component Analysis), que, como se explica en el capítulo 1 de esta tesis, en rasgos generales busca identificar las direcciones (llamadas *componentes*) que nos brindan mayor información sobre el conjunto de datos a analizar, para así poder trabajar con solo unas cuantas de estas componentes perdiendo la menor cantidad de información posible en el proceso.

La dificultad del método de PCA radica en que, como veremos más adelante, es necesario encontrar los primeros autovectores de una matriz simétrica definida positiva. Esta tarea en general es difícil de resolver de forma exacta, por lo que se recurre a algoritmos de aproximación, es decir, algoritmos que encuentren no una solución exacta, sino una solución aproximada, y tal que al darle más margen de cómputo al algoritmo obtenemos una respuesta aún más aproximada. Con este fin, en 2021 *Ian Gemp, Brian McWilliams, Claire Vernade y Thore Graepel* publican un artículo llamado *EigenGame: PCA as a Nash Equilibrium* (*EigenGame: PCA como un Equilibrio de Nash*) [4], sobre el cual se desarrolla este trabajo.

Allí se introduce el juego *EigenGame* que, como veremos en el capítulo 2, es un juego de  $d$  jugadores tal que tiene un único Equilibrio de Nash (es decir, una única situación en la que todos los jugadores adoptan una estrategia óptima dadas las estrategias de los otros) y se da cuando obtenemos los primeros  $d$  autovectores de una matriz simétrica dada. La novedad de este concepto es que ahora el problema de encontrar los primeros  $d$  autovectores de una matriz simétrica se reduce a encontrar el equilibrio de Nash de un juego, pudiendo así abordarlo desde otra perspectiva. En el artículo original se incluyen entonces algunos algoritmos de aproximación para encontrar dicho equilibrio, e incluso en un segundo artículo [5] exhibieron una mejora para su propio enfoque inicial. Pero no contentos con eso, en esta tesis vamos a explorar algunas variantes alternativas usando la idea de algoritmo evolutivo, dándole el nombre de *Evolutuve EigenGame*.

En el capítulo 3 damos a conocer la idea de nuestro algoritmo evolutivo, junto con pseudocódigos y una pequeña discusión de complejidad para cada

una de sus variantes.

En el capítulo 4 exhibimos un ejemplo que muestra la ejecución del algoritmo evolutivo en  $\mathbb{R}^3$ , junto con un teorema que podría ser útil para una eventual prueba completa de convergencia, además de ilustrar brevemente lo que está pasando de fondo en la ejecución del algoritmo.

En el capítulo 5 hacemos unas experimentaciones con nuestro algoritmo para tener una mejor idea de su funcionamiento y de como evoluciona la cantidad de iteraciones necesarias para obtener resultados.

En el capítulo 6 salimos un poco del contexto de PCA y le hacemos una pequeña modificación al algoritmo para usarlo en la resolución de una ecuación diferencial ordinaria.

X

# Índice general

<b>1. Análisis de Componentes Principales</b>	<b>1</b>
1.1. Método . . . . .	3
<b>2. EigenGame</b>	<b>5</b>
2.1. Motivación . . . . .	5
2.2. El Juego . . . . .	6
2.3. Algoritmos con el Método del Gradiente . . . . .	9
2.4. Ventajas . . . . .	11
2.5. Desventajas . . . . .	12
<b>3. Algoritmos Evolutivos</b>	<b>13</b>
3.1. Versión Cuadrática . . . . .	14
3.2. Versión Cúbica . . . . .	17
3.3. Consideraciones Generales . . . . .	21
<b>4. Intuiciones</b>	<b>23</b>
4.1. Visualización . . . . .	23
4.2. Teorema de PseudoConvergencia . . . . .	32
<b>5. Experimentación</b>	<b>35</b>
5.1. Bases de la Experimentación . . . . .	35
5.2. Experimento 1: Cantidad de Muestras . . . . .	38
5.3. Experimento 2: Dimensión de las Muestras . . . . .	40
<b>6. Calculando los Últimos Autovalores</b>	<b>45</b>
6.1. Un Nuevo Uso . . . . .	45

6.2. Aplicación . . . . .	49
<b>A. Ángulos en Dimensiones Altas</b>	<b>61</b>
<b>Bibliografía</b>	<b>65</b>

# Capítulo 1

## Análisis de Componentes Principales

El análisis de componentes principales, desde ahora “PCA”, es una técnica utilizada para describir un conjunto de datos en términos de nuevas variables no correlacionadas llamadas “componentes”. Las componentes se ordenan por la cantidad de varianza original que describen, por lo que la técnica es útil para reducir la dimensionalidad de un conjunto de datos.

Técnicamente, el PCA busca la proyección según la cual los datos queden mejor representados en términos de mínimos cuadrados, es decir, el subespacio sobre el cual podemos proyectar los puntos muestrales “perdiendo la menor cantidad de información posible”.

Por ejemplo, un conjunto de datos puede describir la altura y el peso de 100 niños entre 2 y 15 años. Ambas variables están, obviamente, correlacionadas (los niños de más edad en general son más altos y más pesados). El análisis de componentes principales describe los datos en términos de dos nuevas variables. La primera componente se puede interpretar como “tamaño” o “edad” y recoge la mayor parte de la varianza de los datos originales. La segunda componente describe variabilidad en los datos que no está correlacionada en absoluto con la primera componente principal “tamaño”, y probablemente sea difícil de interpretar. Si el objetivo es reducir la dimensionalidad de los datos, se puede descartar esta segunda componente principal. Lo mismo aplica si el conjunto de datos contiene un número mayor de variables que se pueden interpretar como medidas aproximadas

de “tamaño”. Por ejemplo, longitud del fémur o longitud de los brazos. Un conjunto de datos de este tipo podría describirse generalmente con una única componente principal que se podría interpretar como “tamaño” o “edad”.

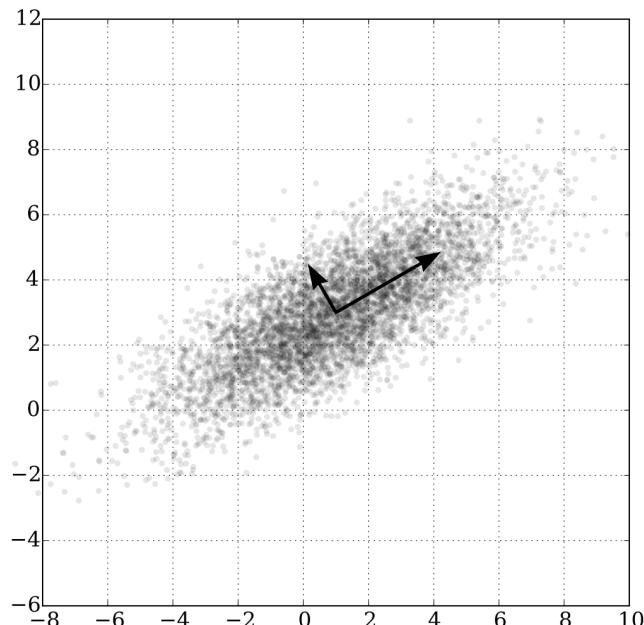


Figura 1.1: Ejemplo de PCA: Los vectores muestran los autovectores de la matriz de correlación escalados mediante la raíz cuadrada del correspondiente autovalor, y desplazados para que su origen coincidan con la media estadística.

El PCA es particularmente útil para reducir la dimensionalidad de un grupo de datos. Las primeras componentes principales describen la mayor parte de la varianza de los datos (mayor descripción cuanto más correlacionadas estuvieran las variables originales). Estas componentes de bajo orden a veces contienen el aspecto “más importante” de la información, y

los demás componentes se pueden ignorar.

Otro ejemplo que describe esto: Un análisis consideró las calificaciones escolares de 15 estudiantes en 8 materias (lengua, matemáticas, física, inglés, filosofía, historia, química, gimnasia). Las dos primeras componentes principales explicaban juntas el 82,1% de la varianza. La primera de ellas parecía fuertemente correlacionada con las materias de humanidades (lengua, inglés, filosofía, historia) mientras que la segunda aparecía relacionada con las materias de ciencias (matemáticas, física, química). Así parece que existe un conjunto de habilidades cognitivas relacionadas con las humanidades y un segundo conjunto relacionado con las ciencias. Estos dos conjuntos de habilidades son estadísticamente independientes, por lo que un alumno puede puntuar alto en solo uno de ellos, en los dos o en ninguno.

## 1.1. Método

El PCA construye una transformación lineal que escoge un nuevo sistema de coordenadas para el conjunto original de datos en el cual la varianza de mayor tamaño del conjunto de datos es capturada en el primer eje (llamada la Primera Componente Principal), la segunda varianza más grande es el segundo eje, y así sucesivamente. Para construir esta transformación lineal debe construirse primero la matriz de covarianza o matriz de coeficientes de correlación. Debido a la simetría de esta matriz existe una base completa de vectores propios de la misma. La transformación que lleva de las antiguas coordenadas a las coordenadas de la nueva base es precisamente la transformación lineal necesaria para reducir la dimensionalidad de datos.

Concretamente, si tenemos un conjunto  $k$  de muestras cada una de las cuales tiene  $n$  entradas que las describen (o sea,  $k$  puntos en  $\mathbb{R}^n$ ), consideramos  $X \in \mathbb{R}^{k \times n}$  la matriz cuyas filas son nuestros puntos muestrales. La matriz de covarianza será entonces

$$M = \frac{X^T X}{k - 1} \in \mathbb{R}^{n \times n}.$$

Notemos que, así construida,  $M$  es una matriz simétrica y semi-definida positiva, por lo que sabemos que posee una base de autovectores todos

ellos con autovalores reales no negativos. Además, durante todo este trabajo vamos a asumir que estos autovalores son todos distintos, ya que el conjunto de matrices en las que hay un autovalor con multiplicidad mayor a 1 es de medida 0. El método de PCA consiste entonces en hallar las componentes  $v_i \in \mathbb{R}^n$  tales que

$$M \cdot v_i = \lambda_i \cdot v_i$$

donde  $\lambda_1$  es el mayor autovalor,  $\lambda_2$  el segundo mayor, y así siguiendo.

Notar que con el problema así planteado no tiene sentido considerar que un autovalor sea nulo. Esto es porque si  $\lambda_i = 0$  para algún  $i$ , como estamos calculando las componentes en orden y dijimos que todos los autovalores son no negativos, todos los autovalores a partir de este son nulos. O sea que  $\lambda_j = 0 \forall i \leq j$ . En este caso podremos describir perfectamente a la matriz  $M$  con unas pocas  $(i - 1)$  componentes, ya que manda todo lo demás a 0 y entonces tenemos un caso particular del problema que vamos a resolver. Por lo tanto, con todo lo ya dicho, durante la totalidad de este trabajo vamos a asumir que los autovalores de las matrices son distintos y estrictamente positivos.

Para profundizar sobre este tema, consultar la bibliografía [6].

# Capítulo 2

## EigenGame

### 2.1. Motivación

Como se vio anteriormente, para aplicar PCA se necesita calcular autovectores de una matriz simétrica semi-definida positiva  $M$  o equivalente-mente, hallar una matriz  $V \in \mathbb{R}^{n \times n}$  tal que  $V^{-1}MV = \Lambda$  con  $\Lambda$  una matriz diagonal. Además, recordemos que dos autovectores de una matriz simétrica con valores distintos siempre son ortogonales entonces, si asumimos que los autovectores están normalizados, podemos concluir que la matriz  $V$  es ortogonal y, por tanto, el problema se reduce a encontrar  $V$  tal que

$$V^T MV = \Lambda.$$

Por otra parte, notemos que si  $v_1, v_2, \dots, v_n$  son las columnas de  $V$ , entonces

$$(V^T MV)_{ij} = \langle v_i, Mv_j \rangle.$$

El problema de PCA comúnmente se plantea en dirección a maximizar las expresiones sobre la diagonal. Es decir, si  $\hat{V}$  es un estimativo de  $V$ , maximizar la traza de la matriz  $\hat{V}^T M \hat{V}$ , que es igual a

$$\sum_{i=1}^n \langle \hat{v}_i, M \hat{v}_i \rangle.$$

Sin embargo, en este contexto nos interesa también minimizar los términos que no están en la diagonal. Es altamente deseable que los términos de la

pinta  $\langle \hat{v}_i, M\hat{v}_j \rangle$  con  $i \neq j$  sean lo mas cercanos a 0 posible, o equivalentemente, que  $\langle \hat{v}_i, M\hat{v}_j \rangle^2$  sea lo menor posible.

Notar que esto es coherente con las propiedades de los autovectores  $v_i$ , ya que

$$\langle v_i, Mv_j \rangle = \langle v_i, \lambda_j v_j \rangle = \lambda_j \langle v_i, v_j \rangle = \begin{cases} \lambda_j & \text{si } i = j \\ 0 & \text{si } i \neq j \end{cases}$$

Además, como una intuición general, el producto interno de dos vectores nos dice “cuán alineados están” (luego de escalar por las normas de los vectores). Entonces querer maximizar  $\langle \hat{v}_i, M\hat{v}_i \rangle$  sería querer que  $\hat{v}_i$  y  $M\hat{v}_i$  estén alineados, lo cual es cierto, ya que queremos que  $M\hat{v}_i$  tenga la misma dirección que  $\hat{v}_i$  (queremos  $M\hat{v}_i = \lambda_i \hat{v}_i$ ) y querer minimizar  $\langle \hat{v}_i, M\hat{v}_j \rangle^2$  cuando  $i \neq j$  es querer que  $\hat{v}_i$  y  $M\hat{v}_j$  estén lo menos alineados posible. Esto es, en otras palabras, querer que sean ortogonales, lo cual también es cierto si  $M\hat{v}_j$  tiene la misma dirección que  $\hat{v}_j$ .

## 2.2. El Juego

Imaginemos que queremos hallar las  $d$  primeras componentes principales de un conjunto de muestras. Entonces planteamos el EigenGame para  $d$  jugadores, donde cada jugador tiene un vector unitario  $\hat{v}_i \in \mathcal{S}^{n-1}$  que pueden modificar, y una función de utilidad  $u_i$  que tienen que maximizar. En cada turno, cada uno de los  $d$  jugadores elige un vector en la esfera, y la utilidad del jugador  $i$  entonces será

$$u_i(\hat{v}_i \mid \hat{v}_{j < i}) = \langle \hat{v}_i, M\hat{v}_i \rangle - \sum_{j < i} \frac{\langle \hat{v}_i, M\hat{v}_j \rangle^2}{\langle \hat{v}_j, M\hat{v}_j \rangle},$$

donde  $\hat{v}_{j < i}$  denota el conjunto  $\{\hat{v}_j : j < i\}$ .

Notar que esta función utilidad es la razonable dada la motivación anterior. Se tiene sumando los términos que se quieren maximizar, restando los que se quieren minimizar, y se tiene unos términos  $\langle \hat{v}_j, M\hat{v}_j \rangle$  dividiendo a los términos  $\langle \hat{v}_i, M\hat{v}_j \rangle^2$  para mantener todos los términos de la función en una misma escala “lineal” y para facilitar el análisis.

**Definición 2.2.1.** Dado un juego de 2 o más jugadores, un conjunto de estrategias para los jugadores es un **Equilibrio de Nash** si cada jugador individual no gana nada modificando su estrategia mientras los otros mantengan las suyas. En otras palabras, es una situación en la cual los jugadores no tienen ningún incentivo para cambiar su estrategia en caso de que las estrategias de sus oponentes no se alteren. [3]

**Teorema 2.2.2.** El equilibrio de Nash del EigenGame para d jugadores es único y se da cuando  $\hat{v}_i = v_i \forall 1 \leq i \leq d$ , donde  $v_1$  el autovector de mayor valor de la matriz de covarianza,  $v_2$  el autovector de segundo mayor valor, y así siguiendo.

*Demostración.* Supongamos que  $\hat{v}_1, \hat{v}_2, \dots, \hat{v}_d$  es un equilibrio de Nash, y vamos a probar por inducción global que  $\hat{v}_i = v_i$  con probabilidad 1.

Primero, si  $i = 1$ , la función  $u_1 = \langle \hat{v}_1, M\hat{v}_1 \rangle$  depende únicamente de  $\hat{v}_1$ . Entonces  $\hat{v}_1$  solo puede tomar vectores que optimicen la expresión  $\langle \hat{v}_1, M\hat{v}_1 \rangle$ , y sabemos que los únicos vectores unitarios que hacen esto son los autovectores de valor máximo. Por tanto, podemos concluir que  $\hat{v}_1 = v_1$ .

Ahora, supongamos que  $\hat{v}_j = v_j \forall 1 \leq j \leq i - 1$ . Entonces,

$$\begin{aligned} u_i(\hat{v}_i \mid v_{j < i}) &= \langle \hat{v}_i, M\hat{v}_i \rangle - \sum_{j < i} \frac{\langle \hat{v}_i, Mv_j \rangle^2}{\langle v_j, Mv_j \rangle} \\ &= \langle \hat{v}_i, M\hat{v}_i \rangle - \sum_{j < i} \frac{\langle \hat{v}_i, \lambda_j v_j \rangle^2}{\langle v_j, \lambda_j v_j \rangle} \\ &= \langle \hat{v}_i, M\hat{v}_i \rangle - \sum_{j < i} \lambda_j \langle \hat{v}_i, v_j \rangle^2. \end{aligned}$$

Además, como sabemos que tenemos una base ortonormal de autovectores, podemos escribir  $\hat{v}_i = \sum_{j=1}^n \alpha_j v_j$ . Si reemplazamos esto y usamos

ortonormalidad, nos queda que

$$\begin{aligned}
 u_i(\hat{v}_i \mid v_{j < i}) &= \left\langle \sum_{j=1}^n \alpha_j v_j, \sum_{j=1}^n \lambda_j \alpha_j v_j \right\rangle - \sum_{j < i} \lambda_j \alpha_j^2 \\
 &= \sum_{j=1}^n \lambda_j \alpha_j^2 - \sum_{j < i} \lambda_j \alpha_j^2 \\
 &= \sum_{j \geq i} \lambda_j \alpha_j^2.
 \end{aligned}$$

Ahora, como  $\sum_{j=1}^n \alpha_j^2 = 1$  ya que son las coordenadas de un vector unitario en una base ortonormal, claramente esta expresión se maximiza cuando  $\alpha_j = \delta_{ij}$ , ya que  $\lambda_j < \lambda_k$  si  $j > k$ . Esto nos muestra que el valor óptimo de  $\hat{v}_i$  es  $v_i$ , como queríamos probar.  $\square$

En general, vamos a usar fuertemente que dado un vector  $v = \sum_{j=1}^n \alpha_j v_j$ , se tiene que  $u_i(v \mid v_{j < i}) = \sum_{j \geq i} \lambda_j \alpha_j^2$ .

Antes de ver los distintos algoritmos para hallar el equilibrio de Nash de un EigenGame, hay que mencionar que en el blog *Towards Data Science* [8] se debate que EigenGame no es en realidad un juego de  $d$  jugadores, sino de 1 solo jugador. Esta afirmación se basa en que los vectores no compiten entre ellos, sino que cada vector intenta buscar su mejor posición basado en los vectores anteriores, que no se ven afectados por el vector en cuestión. Esto tiene sentido, ya que al observar las funciones de utilidad de cada vector podemos notar que la posición óptima de cada uno depende únicamente de como estén ubicados los vectores anteriores, y no así los siguientes. Es decir, el primer vector  $\hat{v}_1$  no compite contra nadie, puesto que su utilidad depende solo de sí mismo. El segundo vector  $\hat{v}_2$  no compite contra nadie, ya que su utilidad depende solo de sí mismo y de  $\hat{v}_1$ , que es autónomo, entonces va a buscar la mejor posición posible dada la posición de  $\hat{v}_1$ , y así siguiendo con los demás. Por lo tanto, no habría competencia real entre  $d$  jugadores, sino que sería un solo jugador intentando maximizar todas las utilidades a la vez. De todas formas, en este trabajo no vamos a abordar este debate ya que no afecta el enfoque que vamos a desarrollar.

## 2.3. Algoritmos con el Método del Gradiente

En el artículo original de DeepMind [4], sobre el cual se basa este trabajo, se propone resolver este juego con una estrategia del estilo *Gradient Ascent* (ascenso del gradiente) aprovechando el siguiente resultado:

**Teorema 2.3.1.** *Si  $\{v_1, v_2, \dots, v_n\}$  son los autovalores de  $M$  ordenados de mayor valor a menor valor, dado  $1 \leq i \leq n$ , se tiene que*

$$u_i(v|v_{j < i}) = \lambda_i - \sin(\theta)^2 \left( \lambda_i - \sum_{l > i} z_l \lambda_l \right)$$

donde  $\theta$  es la distancia angular entre  $v$  y  $v_i$ , y  $z \in \Delta^{n-1}$ , es decir un vector que cumple  $0 \leq z_j \leq 1 \forall j$  y  $\sum_{j=1}^n z_j = 1$ .

*Demuestra*ción. Si  $\theta$  es la desviación angular entre  $v$  y  $v_i$ , podemos entonces escribir

$$v = \cos(\theta)v_i + \sin(\theta)w$$

con  $w \perp v_i$ . Además sean  $\{\alpha_1, \alpha_2, \dots, \alpha_n\}$  las coordenadas de  $w$  en la base  $\{v_1, v_2, \dots, v_n\}$ . Como esta es base ortonormal y  $w \perp v_i$ , se tiene que  $\alpha_i = 0$ . Ahora, usando la expresión de  $u_i$  probada en el teorema anterior,

$$\begin{aligned} u_i(v|v_{j < i}) &= \lambda_i \cos(\theta)^2 + \sum_{l > i} \lambda_l \sin(\theta)^2 \alpha_l^2 \\ &= \lambda_i (1 - \sin(\theta)^2) + \sum_{l > i} \lambda_l \sin(\theta)^2 \alpha_l^2 \\ &= \lambda_i - \sin(\theta)^2 \left( \lambda_i - \sum_{l > i} \lambda_l \alpha_l^2 \right). \end{aligned}$$

Tomando  $z_l = \alpha_l^2$  sale lo pedido.  $\square$

Usando que la función utilidad  $u_i$  se parece a una sinusoidal en función de la desviación angular hacia el autovector verdadero, y que, por tanto, usar el método del gradiente en esa función es buena idea, en [4] se propone el siguiente algoritmo para resolver el EigenGame:

---

**Algorithm 1**  $\alpha$ -EigenGame con Gradient Ascent

---

**Input** La matriz de muestras  $X$ , un número de iteraciones  $T$ , un conjunto de vectores iniciales  $\{\hat{v}_1^0, \hat{v}_2^0, \dots, \hat{v}_d^0\}$ , y un tamaño de step  $\alpha$ .

**Output** Un conjunto de vectores  $\{\hat{v}_i : 1 \leq i \leq d\}$  estimativos de las primeras  $d$  componentes principales.

**procedure**

$\alpha$ -EigenGame( $T, \alpha$ ) :

**for**  $i \leftarrow 1, 2, \dots, d$  **do:**

$\hat{v}_i \leftarrow \hat{v}_i^0$

**for**  $t \leftarrow 1, 2, \dots, T$  **do:**

$recompensa \leftarrow X \cdot \hat{v}_i$

$penalidad \leftarrow \sum_{j < i} \frac{\langle X\hat{v}_i, X\hat{v}_j \rangle}{\langle X\hat{v}_j, X\hat{v}_j \rangle} X \cdot \hat{v}_j$

$\nabla_{\hat{v}_i} \leftarrow 2X^T \cdot (recompensa - penalidad)$

$\nabla_{\hat{v}_i}^R \leftarrow \nabla_{\hat{v}_i} - \langle \nabla_{\hat{v}_i}, \hat{v}_i \rangle \hat{v}_i$

$\hat{v}_i' \leftarrow \hat{v}_i + \alpha \nabla_{\hat{v}_i}^R$

$\hat{v}_i \leftarrow \frac{\hat{v}_i'}{\|\hat{v}_i'\|}$

**return**  $\{\hat{v}_i : 1 \leq i \leq d\}$

**end procedure**

---

Luego, el mismo grupo de investigación de DeepMind sacaría un segundo artículo [5] en el que propone una versión mejorada de este mismo algoritmo, con ventajas que explicamos más adelante.

---

**Algorithm 2**  $\mu$ -EigenGame con Gradient Ascent

---

**Input** La matriz de muestras  $X$ , un número de iteraciones  $T$ , un conjunto de vectores iniciales  $\{\hat{v}_1^0, \hat{v}_2^0, \dots, \hat{v}_d^0\}$ , y una sucesión de tamaños de step  $(\eta_t)_{t=1}^T$ .

**Output** Los vectores  $\{\hat{v}_i : 1 \leq i \leq d\}$  estimativos de componentes principales.

**procedure**

$\mu$ -EigenGame( $T, (\eta_t)_t$ ) :

**for**  $i \leftarrow 1, 2, \dots, d$  **do:**

$\hat{v}_i \leftarrow \hat{v}_i^0$

**for**  $t \leftarrow 1, 2, \dots, T$  **do:**

**for**  $i \leftarrow 1, 2, \dots, d$  **do:**

$recompensa \leftarrow \frac{1}{k} X^T X \cdot \hat{v}_i$

$penalidad \leftarrow \frac{1}{k} \sum_{j < i} \langle X\hat{v}_i, X\hat{v}_j \rangle \hat{v}_j$

$\nabla_i^\mu \leftarrow recompensa - penalidad$

$\nabla_i^{\mu,R} \leftarrow \nabla_i^\mu - \langle \nabla_i^\mu, \hat{v}_i \rangle \hat{v}_i$

$\hat{v}_i' \leftarrow \hat{v}_i + \eta_t \nabla_i^{\mu,R}$

$\hat{v}_i \leftarrow \frac{\hat{v}_i'}{\|\hat{v}_i'\|}$

**return**  $\{\hat{v}_i : 1 \leq i \leq d\}$

**end procedure**

---

## 2.4. Ventajas

- Si bien este no es un aspecto que vamos a considerar o analizar en profundidad durante este trabajo, este algoritmo es paralelizable. Es decir, podríamos correr el algoritmo en  $d$  máquinas al mismo tiempo para acelerar su ejecución. La idea es que podemos asignarle una máquina a cada una de las componentes que buscamos y para calcular *penalidad* en cada iteración de la máquina  $i$  le pedimos el vector  $\hat{v}_j$  a la máquina  $j$  para cada  $j < i$ .

Esto es algo que diferencia este enfoque de, por ejemplo, la solución con el método de la potencia, en el que necesitamos terminar de encontrar un autovector para pasar al siguiente.

- La ventaja que tiene el algoritmo  $\mu$  por sobre el  $\alpha$  es que el  $\mu$  no tiene sesgo. Esto quiere decir básicamente que el desplazamiento  $\nabla_i^\mu$  es lineal en  $M$ . O sea, si queremos hacer una partición en la matriz de covarianza y correr el algoritmo de forma paralela en cada uno de los subconjuntos para luego juntar los datos obtenidos, podemos hacerlo sin cambiar el resultado respecto de si no hiciéramos la partición.

Por ejemplo, si yo pudiera descomponer mi matriz de covarianza  $M$  como la suma de un conjunto de matrices  $M = \sum_{t=1}^m M_t$ , podemos separar la suma cuando calculamos el desplazamiento en el algoritmo. Es decir, nosotros calculamos

$$\begin{aligned}\nabla_i^\mu &= \frac{1}{k} \left( M\hat{v}_i - \sum_{j < i} \langle \hat{v}_i, M\hat{v}_j \rangle \hat{v}_j \right) = \frac{1}{k} \left( \sum_{t=1}^m M_t \hat{v}_i - \sum_{j < i} \langle \hat{v}_i, \sum_{t=1}^m M_t \hat{v}_j \rangle \hat{v}_j \right) \\ &= \sum_{t=1}^m \frac{1}{k} \left( M_t \hat{v}_i - \sum_{j < i} \langle \hat{v}_i, M_t \hat{v}_j \rangle \hat{v}_j \right)\end{aligned}$$

Entonces, podemos paralelizar el cálculo de  $\nabla_i^\mu$  calculando cada

$$\nabla_{i,t}^\mu = \frac{1}{k} \left( M_t \hat{v}_i - \sum_{j < i} \langle \hat{v}_i, M_t \hat{v}_j \rangle \hat{v}_j \right)$$

por separado y después sumar todo para obtener  $\nabla_i^\mu$ .

## 2.5. Desventajas

- La desventaja más significativa de este algoritmo radica en lo siguiente: Su performance depende en buena medida de los vectores de inicialización  $\hat{v}_i^0$ . No existe una cota asintótica de cantidad de operaciones en función de la distancia angular del vector inicial  $\hat{v}_i^0$  y el autovalor  $v_i$ . Es decir que, dada la matriz  $X$ , la cantidad de iteraciones necesarias para que  $\hat{v}_i$  converja a  $v_i$  puede ser arbitrariamente grande cuando el ángulo  $\theta_0$  entre  $\hat{v}_i^0$  y  $v_i$  se acerca a  $\pi$ .

Esto es grave por lo siguiente: Dado que PCA es una técnica de reducción de dimensión, es esperable hacer esta técnica en espacios de dimensión muy grande, i.e.  $n$  muy grande. Además, se sabe que, dado un  $w \in \mathbf{S}^n$  arbitrario y un ángulo  $0 \leq \theta < \frac{\pi}{2}$ , la probabilidad de que un vector sorteado con distribución uniforme sobre la esfera caiga dentro del conjunto

$$\{v \in \mathbf{S}^n : \angle(v, w) < \theta\}$$

tiende a 0 cuando  $n \rightarrow \infty$ , donde  $\angle(v, w)$  es el ángulo entre  $w$  y  $v$ . Adjuntamos una demostración de esto en el apéndice A. Esto nos dice que, en dimensiones altas, encontrar un vector inicial  $\hat{v}_i^0$  con distancia angular a  $v_i$  menor a cierto umbral no es algo asumible ni trivial.

- Otra desventaja viene de que los algoritmos para resolver EigenGame presentados en estos artículos usan fuertemente la forma de la función y el hecho de que sea, en cierto sentido, una sinusoidal. Es decir, si la función de utilidad que queremos maximizar fuera otra no tan amigable, el algoritmo no serviría.

Por ejemplo, si quisiéramos resolver otro problema de maximización donde tuviéramos una función con máximos locales que no coinciden con el máximo global deseado, utilizar alguno de estos algoritmos con un método de gradiente no nos asegura que vayamos a converger a la solución que estamos buscando.

A continuación, proponemos un algoritmo sin estas desventajas.

## Capítulo 3

# Algoritmos Evolutivos

Como vimos en los capítulos anteriores, el problema de PCA se puede resolver mediante el EigenGame. Concretamente, encontrando su equilibrio de Nash, que es único. Si bien en los artículos de DeepMind se discuten algunos algoritmos, a continuación vamos a desarrollar un camino alternativo para resolver el EigenGame mediante algoritmos evolutivos.

Los algoritmos evolutivos son métodos de optimización y búsqueda de soluciones basados en los postulados de la evolución biológica. En ellos se mantiene un conjunto de entidades que representan posibles soluciones, las cuales se mezclan, y compiten entre sí, de tal manera que las más aptas son capaces de prevalecer a lo largo del tiempo, evolucionando hacia mejores soluciones cada vez.

En este caso, como queremos buscar el óptimo de una función en la esfera unitaria  $\mathbf{S}^{n-1}$ , la idea será, dado un vector inicial, modificarlo en una coordenada, evaluarlo, y quedarnos con el mejor vector de entre el inicial y todos los modificados. Esto lo hacemos sumando o restando un real positivo “step” a alguna de las  $n$  coordenadas y después normalizando, ya que solo buscamos un óptimo sobre la esfera. Esto nos asegura que la función utilidad solo pueda crecer con el paso de las iteraciones.

Otro truco para el correcto funcionamiento del algoritmo es que, si ninguno de los  $2n$  vectores modificados tiene mayor utilidad que el actual, reducimos el valor “step”. Esto significaría que estamos “lo suficientemente cerca” del óptimo como para que grandes modificaciones impliquen una peor utilidad, entonces debemos buscar en vectores más cercanos al actual.

Finalmente, debemos pasarle al algoritmo como parámetros un valor real positivo  $\varepsilon$  y un entero positivo  $T$ , que van a ser usados para establecer el criterio de parada del algoritmo (cuando ya hayamos hecho  $T$  iteraciones o cuando el valor de *step* sea menor a  $\varepsilon$ ) y un valor *step\_inicial* para inicializar la variable *step*.

### 3.1. Versión Cuadrática

La primera versión de *Evolutionary EigenGame* que vamos a ver se centra en ser eficiente y tener una mejor complejidad. La clave de esta versión es agilizar el cálculo de la función *utilidad* mediante variables auxiliares que iremos actualizando a medida que evolucione la ejecución.

Las variables clave en este caso van a ser la lista de vectores  $Mv$  que va a guardar el vector resultante de  $M \cdot \hat{v}_i$  para cada  $1 \leq i \leq d$ , y la matriz  $vMv$  que va a guardar el resultado del producto interno  $\langle \hat{v}_i, M \cdot \hat{v}_j \rangle$  para cada  $1 \leq i, j \leq d$ . Además, los notaremos de la siguiente forma:

$$\begin{aligned} Mv_i &= M \cdot \hat{v}_i \quad \forall 1 \leq i \leq d \\ v_j Mv_i &= \langle \hat{v}_i, M \cdot \hat{v}_j \rangle \quad \forall 1 \leq i, j \leq d. \end{aligned}$$

En este algoritmo, solo vamos a calcular la utilidad de vectores de la pista  $\hat{v}_i \pm step \cdot e_j$ . Por lo tanto, podemos calcular las utilidades como

$$\begin{aligned}
u_i(\hat{v}_i \pm step \cdot e_j \mid v_{l < i}) &= \langle \hat{v}_i \pm step \cdot e_j, M \cdot (\hat{v}_i \pm step \cdot e_j) \rangle \\
&\quad - \sum_{l < i} \frac{\langle \hat{v}_i \pm step \cdot e_j, M \cdot \hat{v}_l \rangle^2}{\langle \hat{v}_l, M \cdot \hat{v}_l \rangle} \\
&= \langle \hat{v}_i, M \cdot \hat{v}_i \rangle \pm step(\langle e_j, M \cdot \hat{v}_i \rangle + \langle \hat{v}_i, M \cdot e_j \rangle) \\
&\quad + step^2 \langle e_j, M \cdot e_j \rangle - \sum_{l < i} \frac{(\langle \hat{v}_i, M \cdot \hat{v}_l \rangle \pm step \langle e_j, M \cdot \hat{v}_l \rangle)^2}{v_l M v_l} \\
&= v_i M v_i \pm step(\langle e_j, M v_i \rangle + \langle \hat{v}_i, M_j \rangle) + step^2 \cdot M_{jj} \\
&\quad - \sum_{l < i} \frac{(v_i M v_l \pm step \langle e_j, M v_l \rangle)^2}{v_l M v_l} \\
&= v_i M v_i \pm step(M v_i[j] + \langle \hat{v}_i, M_j \rangle) + step^2 \cdot M_{jj} \\
&\quad - \sum_{l < i} \frac{(v_i M v_l \pm step \cdot M v_l[j])^2}{v_l M v_l}
\end{aligned}$$

donde  $M_j$  denota la  $j$ -ésima columna de la matriz  $M$  y  $w[j]$  denota la  $j$ -ésima coordenada del vector  $w$ . Entonces, podemos definir la función

$$\begin{aligned}
utilidad(i, j, signo, step) &= v_i M v_i \pm step(M v_i[j] + \langle \hat{v}_i, M_j \rangle) + step^2 \cdot M_{jj} \\
&\quad - \sum_{l < i} \frac{(v_i M v_l \pm step \cdot M v_l[j])^2}{v_l M v_l}
\end{aligned}$$

donde la variable *signo* determina si los  $\pm$  son  $+$  o  $-$ . Notar que si quisieramos calcular la utilidad de uno de los vectores  $\hat{v}_i$ , simplemente tenemos que setear la variable  $step = 0$ , y entonces las variables  $j$  y *signo* pasan a ser irrelevantes.

De esta forma, vemos que podemos calcular la utilidad de estos vectores en  $O(n)$  operaciones: Todas las operaciones, salvo el producto interno, toman  $O(1)$ , y tenemos  $i + 2$  términos. Como  $i \leq d \leq n$ , tenemos  $O(n)$  términos que se calculan en una cantidad acotada de operaciones, y un producto interno que se calcula en  $O(n)$  operaciones.

---

**Algorithm 3** Evolutive-EigenGame Cuadrático para  $d$  jugadores

---

**Input** La matriz de covarianza  $M$ , un real  $\varepsilon > 0$  que actúe como error mínimo, un máximo de iteraciones  $T$ , y un valor *step\_inicial*.

**Output** Un conjunto de vectores  $\{\hat{v}_i : 1 \leq i \leq d\}$  estimativos de las primeras  $d$  componentes principales.

```

procedure
  for  $i \leftarrow 1, 2, \dots, d$  do:
     $\hat{v}_i \leftarrow e_i$ 
     $M\hat{v}_i \leftarrow M \cdot \hat{v}_i$ 
    for  $j \leftarrow 1, 2, \dots, d$  do:
       $v_i M v_j \leftarrow \langle \hat{v}_i, M \hat{v}_j \rangle$ 

Evolutive_EigenGame( $T, step\_inicial, \varepsilon$ ) :
  for  $i \leftarrow 1, 2, \dots, d$  do:
     $step \leftarrow step\_inicial, t \leftarrow 0$ 
    while  $t < T, step \geq \varepsilon$  do:
       $candidato \leftarrow \hat{v}_i, utilidad\_actual \leftarrow utilidad(i, 0, +, 0)$ 
      for  $j \leftarrow 1, 2, \dots, n$  do:
         $utilidad\_nueva \leftarrow utilidad(i, j, +, step)$ 
        if  $utilidad\_nueva > utilidad\_actual$  do:
           $candidato \leftarrow \frac{\hat{v}_i + step \cdot e_j}{\|\hat{v}_i + step \cdot e_j\|}, utilidad\_actual \leftarrow utilidad\_nueva$ 
         $utilidad\_nueva \leftarrow utilidad(i, j, -, step)$ 
        if  $utilidad\_nueva > utilidad\_actual$  do:
           $candidato \leftarrow \frac{\hat{v}_i - step \cdot e_j}{\|\hat{v}_i - step \cdot e_j\|}, utilidad\_actual \leftarrow utilidad\_nueva$ 
        if  $candidato = \hat{v}_i$  do:
           $step \leftarrow \frac{step}{2}$ 
        else:
           $\hat{v}_i \leftarrow candidato, M\hat{v}_i \leftarrow M \cdot \hat{v}_i$ 
          for  $j \leftarrow 1, 2, \dots, d$  do:
             $v_i M v_j \leftarrow \langle \hat{v}_i, M \hat{v}_j \rangle, v_j M v_i \leftarrow \langle \hat{v}_j, M \hat{v}_i \rangle$ 
  return  $\{\hat{v}_i : 1 \leq i \leq d\}$ 
end procedure

```

---

Notar que, como cada vector depende únicamente de los anteriores, conviene primero obtener un buen estimativo de  $\hat{v}_1$ , después usar ese para obtener un buen estimativo de  $\hat{v}_2$ , y así siguiendo. A esta forma de proceder se la denomina “secuencial”. Además, es claro que con este proceso el valor de  $u_i(\hat{v}_i)$  solo puede crecer con el pasar de las iteraciones.

Si queremos analizar la complejidad de este algoritmo, podemos empezar con el cuerpo dentro del **while**. Como vimos, la función *utilidad* se calcula en  $O(n)$ , entonces esa es la complejidad de la primera línea. Dentro

del loop siguiente calculamos más utilidades y sumas y normas de vectores, todas operaciones en  $O(n)$ . Como estamos iterando de 1 a  $n$ , el loop completo está en  $O(n^2)$ . Luego tenemos un **if** en  $O(1)$  y un **else** en  $O(d \cdot n)$  pues iteramos de 1 a  $d$  haciendo productos de  $O(n)$ . Como  $d \leq n$ , tenemos que todo el cuerpo del while está en  $O(n^2)$ .

Ahora, si llamamos  $\mathcal{I}(X, T, step\_inicial, \varepsilon, i)$  a la cantidad de iteraciones en  $t$  necesarias para calcular la  $i$ -ésima componente de la matriz  $M = \frac{X^T M}{k-1}$  con el algoritmo inicializado con  $T, step\_inicial, \varepsilon$ , entonces el proceso para calcular la  $i$ -ésima coordenada está en  $O(n^2 \cdot \mathcal{I}(X, T, step\_inicial, \varepsilon, i))$ , lo cual vamos a notar  $O(n^2 \cdot \mathcal{I})$ .

## 3.2. Versión Cúbica

Si bien en la práctica la versión cuadrática funciona correctamente, a la hora de probar la convergencia del algoritmo surgen dos modificaciones del mismo:

Notar que, a la hora de modificar el vector actual, se usan los vectores canónicos como direcciones arbitrarias. Sin embargo, luego tenemos que normalizar el vector, pues solo buscamos vectores en la esfera. Por esto, sería más lógico movernos en direcciones tangentes a la esfera en el punto en el que estamos para obtener un vector más alejado, o sea, movernos en direcciones ortogonales al vector actual. Si nos movemos en direcciones arbitrarias, podríamos incluso terminar en el mismo vector actual luego de normalizar, o mismo en el origen, si justo el vector actual es uno de los canónicos. Se puede, entonces, hacer Gram-Schmidt en cada iteración del while con el vector *actual* para así obtener una base ortonormal del plano tangente a la esfera en *actual*. Como veremos más adelante, esto no afectara la complejidad del algoritmo siempre y cuando la dimensión del espacio sea razonable respecto a la cantidad de muestras (equivalentemente, si  $n \in O(k)$ ).

La segunda modificación tiene que ver con el valor de *step* y los posibles incrementos de la función utilidad. Podría pasar, en principio, que  $u_i(\hat{v}_i)$  converja a una asintota estrictamente menor al óptimo buscado manteniendo el valor de *step* sin cambiar. Dicho en otras palabras, en cada iteración pegamos saltos de la misma longitud pero sin tender al óptimo. Para evi-

tar esto, se puede introducir una *tasa de crecimiento* para controlar que la mejora de  $u_i(\hat{v}_i)$  en cada paso sea por lo menos proporcional al valor *step*. Esto nos ayudará mucho en el teorema de la sección siguiente.

Por último, notemos que como ahora vamos a calcular la utilidad de vectores más generales comparados con los de la versión anterior, entonces debemos usar una versión más directa y menos eficiente de la función *utilidad*. Como resultado, nos quedaría la siguiente variante del *Evolutionary EigenGame*:

---

**Algorithm 4** Evolutive-EigenGame Cúbico para  $d$  jugadores v2

---

**Input** La matriz de covarianza  $M$ , un real  $\varepsilon > 0$  que actúe como error mínimo, un máximo de iteraciones  $T$ , un valor  $step\_inicial$ , y una tasa de crecimiento  $\tau_c > 0$ .

**Output** Un conjunto de vectores  $\{\hat{v}_i : 1 \leq i \leq d\}$  estimativos de las primeras  $d$  componentes principales.

```

procedure
  for  $i \leftarrow 1, 2, \dots, d$  do:
     $\hat{v}_i \leftarrow e_i$ 
     $Mv_i \leftarrow M \cdot \hat{v}_i$ 
     $v_i Mv_i \leftarrow \langle \hat{v}_i, Mv_i \rangle$ 

  utilidad(vector,  $i$ ) :
    respuesta  $\leftarrow \langle vector, M \cdot vector \rangle$ 
    for  $j \leftarrow 1, 2, \dots, i - 1$  do:
      respuesta  $\leftarrow respuesta - \frac{\langle vector, Mv_j \rangle^2}{v_j Mv_j}$ 
    return respuesta

Evolutive_EigenGame_v2( $T, step\_inicial, \varepsilon, \tau_c$ ) :
  for  $i \leftarrow 1, 2, \dots, d$  do:
    step  $\leftarrow step\_inicial$ ,  $t \leftarrow 0$ 
    while  $t < T$ ,  $step \geq \varepsilon$  do:
       $\{dir_1, dir_2, \dots, dir_{n-1}\} \leftarrow Gram\_Schmidt(actual)$ 
      actual  $\leftarrow \hat{v}_i$ , utilidad_actual  $\leftarrow utilidad(actual, i)$ 
      for  $j \leftarrow 1, 2, \dots, n - 1$  do:
        candidato  $\leftarrow \frac{actual \pm step \cdot dir_j}{\|actual \pm step \cdot dir_j\|}$ 
        utilidad_nueva  $\leftarrow utilidad(candidato, i)$ 
        if  $utilidad\_nueva - utilidad\_actual \geq \tau_c \cdot step$  do:
          actual  $\leftarrow candidato$ , utilidad_actual  $\leftarrow utilidad\_nueva$ 
        if  $actual = \hat{v}_i$  do:
          step  $\leftarrow \frac{step}{2}$ 
        else:
           $\hat{v}_i \leftarrow actual$ ,  $Mv_i \leftarrow M \cdot \hat{v}_i$ ,  $v_i Mv_i \leftarrow \langle \hat{v}_i, Mv_i \rangle$ 
    return  $\{\hat{v}_i : 1 \leq i \leq d\}$ 
end procedure

```

---

Podemos observar que en esta versión, solo necesitamos los productos  $\langle \hat{v}_i, M \cdot \hat{v}_i \rangle$ , y no todos los  $\langle \hat{v}_i, M \cdot \hat{v}_j \rangle$  para  $j \neq i$ .

Si queremos analizar la complejidad del cuerpo del **while**, esta será por lo menos  $O(n^3)$ , que es la complejidad de hacer *Gram-Schmidt*. Por otro lado, la función *utilidad* usada en este caso tiene complejidad  $O(n^2)$ , ya que tenemos que hacer la multiplicación  $M \cdot vector$  e  $i$  productos internos.

Entonces, como tenemos que hacer esto para cada índice  $j = 1, \dots, n - 1$ , se tiene que el cuerpo del **while** está en  $O(n^3)$ , y de allí el nombre de esta versión de *EigenGame*. Por lo tanto, este algoritmo tardaría  $O(n^3 \cdot \mathcal{I})$  para calcular cada componente.

Si bien esta versión es menos eficiente que la discutida anteriormente, las modificaciones hechas nos permiten establecer el teorema del siguiente capítulo que, si bien no es una prueba de complejidad, puede ser de ayuda para una prueba futura, además de ilustrar y dar una intuición de que está pasando de fondo en la ejecución del algoritmo.

Algunas pequeñas observaciones respecto a esta versión cúbica:

- Al hacer Gram-Schmidt, lo que nos importa es que las direcciones en las que nos movemos sean tangentes a la esfera. Pero además les podemos pedir que sean ortogonales a los autovectores ya encontrados. Así, en vez de  $n - 1$ , bastaría con mirar  $n - i$  direcciones para el vector  $\hat{v}_i$ . Como esto no cambia la complejidad final del algoritmo y restringe la cantidad de nuevos candidatos a evaluar en cada paso, esta modificación no fue analizada en este trabajo, pero puede ser una futura fuente de investigación.
- Como vimos, calcular la *utilidad* de un vector en esta versión cuesta  $O(n^2)$  operaciones. Sin embargo, en el hipotético caso en el que  $k < n$  esto se puede reducir a  $O(n \cdot k)$  operaciones de la siguiente manera: En vez de computar  $M \cdot \text{vector}$ , que cuesta  $O(n^2)$  operaciones, podemos computar  $X \cdot \text{vector}$  que cuesta  $O(nk)$  al ser  $X$  una matriz de  $k \times n$ . Con esto, podemos asignarle a *respuesta* el valor  $\|X \cdot \text{vector}\|^2$ , ya que

$$\begin{aligned} \langle \text{vector}, M \cdot \text{vector} \rangle &= \langle \text{vector}, X^T \cdot X \cdot \text{vector} \rangle = \langle X \cdot \text{vector}, X \cdot \text{vector} \rangle \\ &= \|X \cdot \text{vector}\|^2. \end{aligned}$$

Y en este caso calcular  $\|X \cdot \text{vector}\|^2$  cuesta  $O(k)$  operaciones por ser un vector de  $k$  coordenadas.

Y después, en vez de restar el término  $\frac{\langle \text{vector}, Mv_j \rangle^2}{v_j M v_j}$  que cuesta  $O(n)$  para cada  $j$ , podemos restar

$$\frac{\langle X \cdot \text{vector}, X \cdot v_j \rangle^2}{v_j M v_j}$$

ya que  $X \cdot \text{vector}$  ya lo tenemos y podemos tener precalculado  $X \cdot v_j$  para cada  $j$ , evitando operaciones innecesarias. En este caso, nos tomaría  $O(k)$  iteraciones por cada  $j$ , y, por tanto,  $O(n \cdot k)$  operaciones en total.

### 3.3. Consideraciones Generales

Vamos a mencionar dos aspectos a tener en cuenta que abarcan a ambos algoritmos y a su estudio:

- Teóricamente podría pasar que en algún estado intermedio de la ejecución de cualquiera de los dos algoritmos, cuando está buscando la  $i$ -ésima componente el vector  $\hat{v}_i$  valga lo mismo que alguna de las direcciones en las que nos movemos para analizar candidatos ( $e_j$  o  $-e_j$  en la versión cuadrática y  $dir_j$  o  $-dir_j$  en la versión cúbica, para algún  $j$ ) y que la variable  $step$  sea 1. En cuyo caso, al calcular  $\hat{v}_i - step \cdot e_j$ ,  $\hat{v}_i + step \cdot e_j$ ,  $\hat{v}_i - step \cdot dir_j$ , o  $\hat{v}_i + step \cdot dir_j$  según corresponda, vamos a obtener el vector 0 y vamos a dividir por 0 tratando de normalizar el vector. Sin embargo, en la práctica esto no sucede, y si así ocurriera, simplemente se puede descartar el 0 como vector candidato. Como creímos que no valía la pena agregar esta condición a un algoritmo ya de por sí largo, obviamos este aspecto en los pseudocódigos.
- En el artículo [7] se analiza una técnica que consiste en reducir la dimensión  $n$  proyectando todas las muestras a un subespacio generado por  $d + l$  vectores en el que se estima que van a estar las primeras  $d$  componentes, y luego se recurre a algoritmos para resolver PCA en un espacio de dimensión mucho menor. Experimentos detallados en el artículo muestran que esto puede mejorar la performance del cálculo de PCA significativamente. Sin embargo, decidimos no experimentar esta técnica con nuestro algoritmo, y dejarlo como un trabajo a futuro.



# Capítulo 4

## Intuiciones

### 4.1. Visualización

En esta sección vamos a ver gráficamente que pasa de fondo en el algoritmo en el caso de 3 dimensiones, que sirve para imaginarnos después los casos de dimensiones más altas.

El hecho clave para esto es recordar el resultado probado en el capítulo 2:

**Propiedad 4.1.1.** *Si evaluamos la  $i$ -ésima función de utilidad en un vector  $\hat{v}_i$  con coordenadas  $(\alpha_1, \dots, \alpha_n)$  en la base ortonormal de autovectores  $\{v_1, \dots, v_n\}$ , tenemos que*

$$u_i(\hat{v}_i | v_{j < i}) = \sum_{j \geq i} \lambda_j \alpha_j^2.$$

Como los autovectores forman una base ortonormal, después de un cambio de base ortogonal (una serie de rotaciones y reflexiones) podemos asumir sin perdida de la generalidad que la base de autovectores es la canónica  $\{e_1, \dots, e_n\}$ , y que por tanto, como las coordenadas de algún vector  $x$  en la base canónica es el mismo vector  $x$ , se tiene

$$u_i(x | v_{j < i}) = \sum_{j \geq i} \lambda_j x_j^2.$$

Notar que esto nos manda la base canónica  $\{e_1, \dots, e_n\}$  a una base ortonormal que vamos a notar con  $\{d_1, \dots, d_n\}$ .

Esto nos permite expresar la función utilidad  $u_i$  como una composición de dos funciones más visuales:

- Sea  $L_i : \mathbb{R}^n \rightarrow \mathbb{R}^n$  la función que multiplica las primeras  $i - 1$  coordenadas por 0 y las otras por  $\sqrt{\lambda_j}$ . Concretamente,

$$L_i(x) = (0, \dots, 0, \sqrt{\lambda_i}x_i, \sqrt{\lambda_{i+1}}x_{i+1}, \dots, \sqrt{\lambda_n}x_n).$$

Si restringimos esta función a la esfera, obtenemos una función  $L_i : \mathcal{S}^{n-1} \rightarrow \mathbb{R}^n$  cuya imagen  $L_i(\mathcal{S}^{n-1})$  es una variedad diferencial elípticoide.

Por ejemplo, si  $i = 1$ ,

$$L_1(\mathcal{S}^{n-1}) = \{x \in \mathbb{R}^n : \sum_{j=1}^n \frac{x_j^2}{\lambda_j} = 1\}$$

que es un elípticoide de dimensión  $n - 1$ . Además en este caso  $L_1$  es biyección, ya que la asignación  $(x_1, \dots, x_n) \rightarrow (\sqrt{\lambda_1}x_1, \dots, \sqrt{\lambda_n}x_n)$  es una correspondencia entre  $\mathcal{S}^{n-1}$  y el elípticoide ya definido.

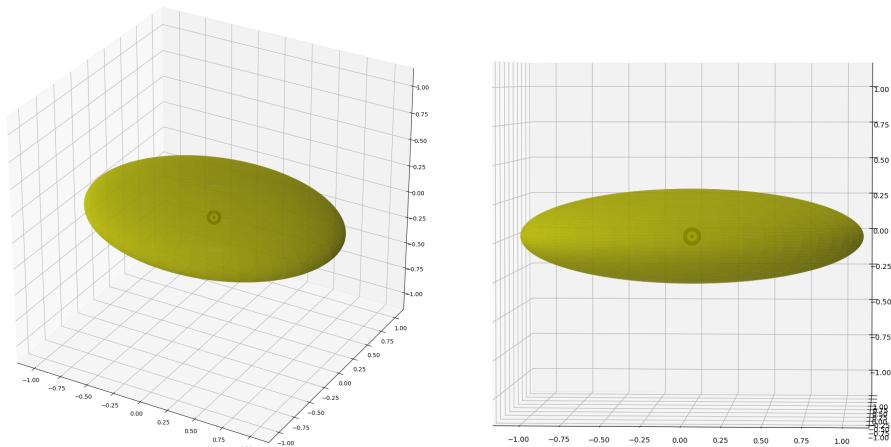


Figura 4.1: Ejemplo de elípticoide de dimensión 2.

Y si  $i \geq 2$ ,

$$L_i(\mathcal{S}^{n-1}) = \{x \in \mathbb{R}^n : x_j = 0 \forall j < i \text{ y } \sum_{j=i}^n \frac{x_j^2}{\lambda_j} \leq 1\}$$

que es una variedad diferencial de dimensión  $n + 1 - i$  cuyo borde es un elipsoide de dimensión  $n - i$ . Más precisamente, es un elipsoide de dimensión  $n - i$  junto con su cápsula convexa.

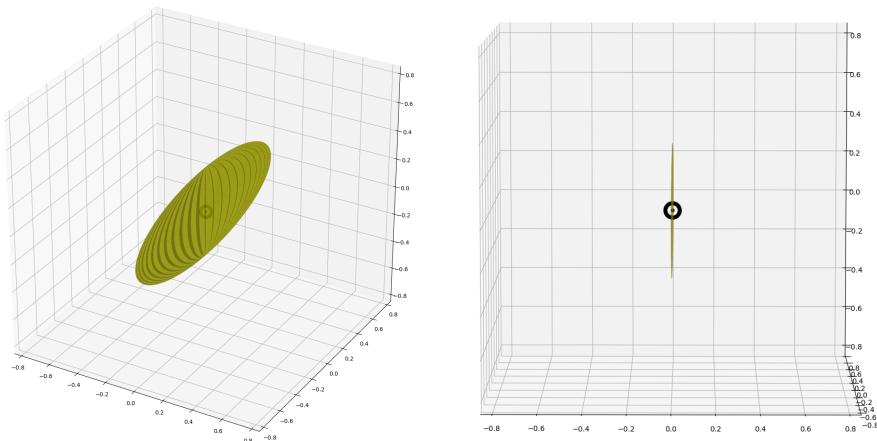


Figura 4.2: Ejemplo de  $L_i(\mathcal{S}^{n-1})$  con  $i = 2$  y  $n = 3$ .

La principal diferencia cuando  $i > 1$  es que  $L_i$  manda a 0 algunas coordenadas de los vectores, entonces si por ejemplo  $L_i(x) = y$  se tiene que  $\sum_{j=i}^n \frac{y_j^2}{\lambda_j} = \sum_{j=i}^n x_j^2 = \|x\|^2 - \sum_{j=1}^{i-1} x_j^2$  y este valor puede ser cualquier real entre 0 y 1. Entonces podríamos “perder norma” en las coordenadas que se van a 0, y es por eso que, además del elipsoide, la cápsula convexa también forma parte de la imagen de  $L_i$ .

- Sea  $N : \mathbb{R}^n \rightarrow \mathbb{R}_{\geq 0}$  la función *norma al cuadrado* dada por  $N(x) = \|x\|^2 = \sum_{j=1}^n x_j^2$ .

Ya habiendo introducido estas dos funciones, es fácil notar que, después

de un cambio de coordenadas,  $u_i \equiv N \circ L_i$ , ya que tenemos

$$N \circ L_i(x) = N(0, \dots, 0, \sqrt{\lambda_i}x_i, \sqrt{\lambda_{i+1}}x_{i+1}, \dots, \sqrt{\lambda_n}x_n) = \sum_{j=i}^n \lambda_j x_j^2.$$

Esto nos dice que, a fin de cuentas, lo que queremos maximizar es la norma de un vector sobre un elipsoide. Notar que entonces, para maximizar la  $u_i$  cuando  $i > 1$ , vamos a tener que minimizar las componentes que se van a 0 al aplicar la  $L_i$ , ya que estas determinan cuánta norma perdemos en el proceso y, por tanto, cuán dentro estamos de la cápsula convexa del elipsoide. Esto deriva en que el  $i$ -ésimo autovector tiene que ser ortogonal a los anteriores, lo que explica el hecho de que en el algoritmo *Evolutive EigenGame* nunca nos fijamos que los autovectores aproximados sean ortogonales entre ellos, y, sin embargo, siempre terminan siéndolo, como veremos en la experimentación.

Ahora que ya tenemos una idea gráfica de qué es lo que hace la función  $u_i$  sobre la esfera, podemos ver un ejemplo de como procedería el *Evolutive EigenGame Cuadrático* en el caso de  $\mathbb{R}^3$ :

Primero elegimos un punto aleatorio  $v_0$  sobre la esfera y marcamos todos los puntos que vamos a probar como candidatos. En este caso, supongamos que  $step\_inicial = 1$ , entonces son 6 y son de la forma  $\frac{v_0 \pm d_j}{\|v_0 \pm d_j\|}$  donde los  $\{d_j\}_{j=1}^n$  es la base ortonormal a la que van a parar los canónicos después del cambio de coordenadas hecho al principio de este capítulo.

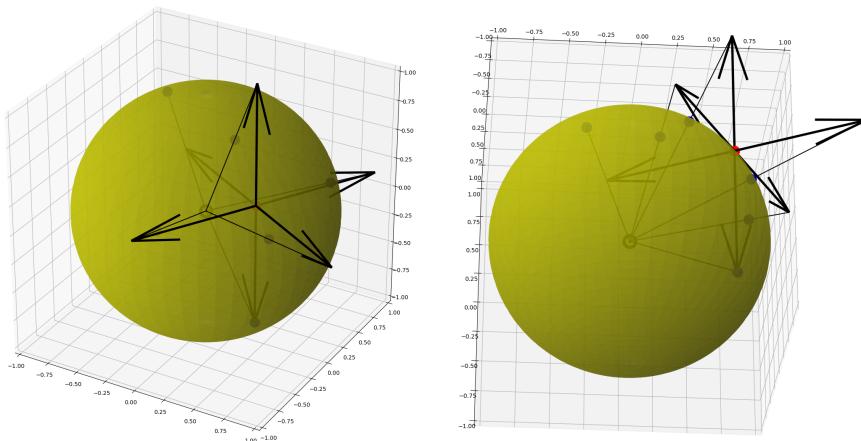


Figura 4.3: El punto rojo es  $v_0$ , las flechas negras son los desplazos por los  $d_j$ , y los puntos azules son las proyecciones de estos 6 puntos de la pista  $v_0 \pm d_j$  a la esfera (proyección remarcada con segmentos hacia el origen).

Ahora, con los puntos candidatos, aplicamos el mapa  $L_1$ , mandando la esfera a un elipsode.

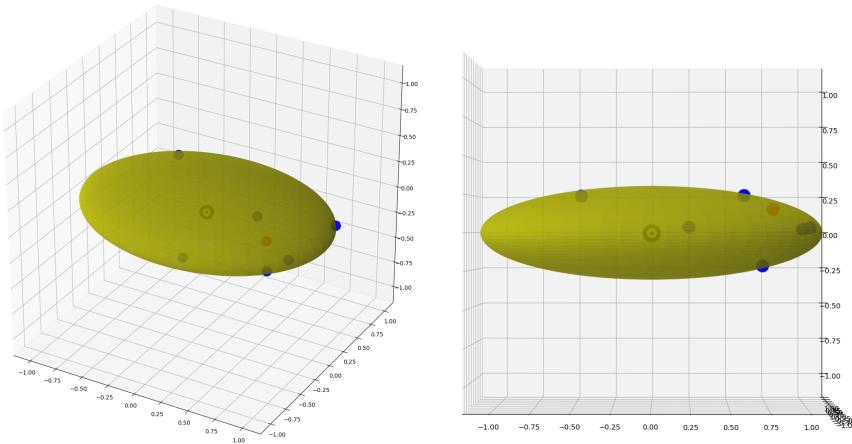


Figura 4.4: Así es como quedaría aplicar  $L_1$  a la esfera con los puntos mostrados anteriormente.

En este paso se aprecia por qué nuestro algoritmo casi no depende del vector inicial que sorteamos: Cuando los desplazamientos todavía son grandes, hay puntos candidatos (que posteriormente vamos a evaluar) lejos del punto inicial en el elipsoide.

Ahora, nos quedamos con el punto con mayor norma, que sería el que está más alejado del origen. Este punto va a ser nuestro nuevo punto rojo. Con este nuevo punto rojo, evaluamos nuestros nuevos 6 puntos candidatos sobre el elipsoide y nos quedamos con el de mayor norma

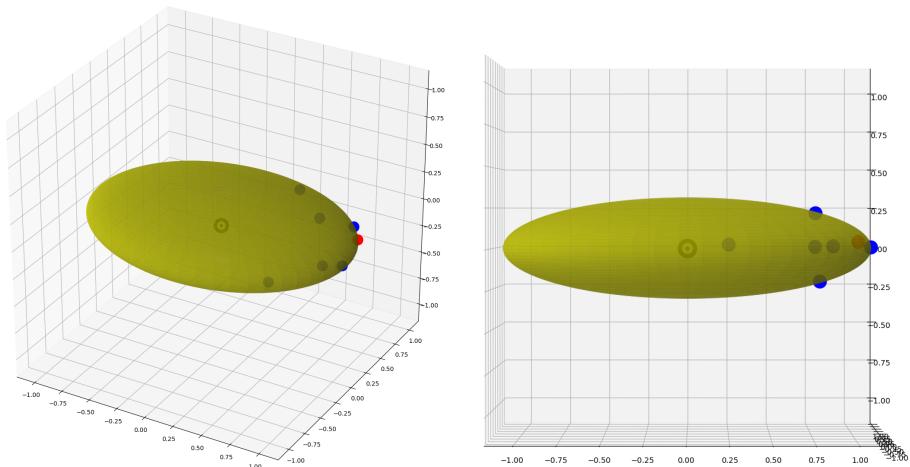


Figura 4.5: Marcamos como punto rojo el de mayor norma y como puntos azules a los nuevos candidatos a partir del rojo.

El siguiente paso va a tener esta pinta:

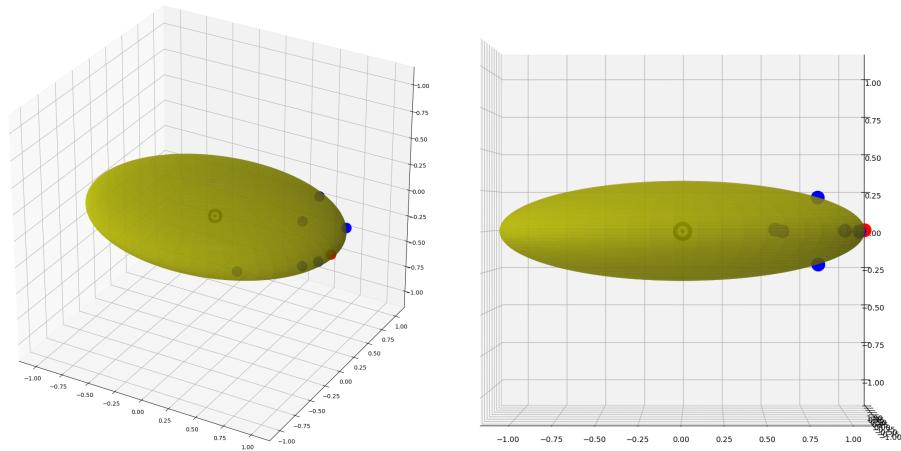


Figura 4.6: Volvemos a elegir punto rojo basado en su norma y marcamos los 6 nuevos candidatos.

Notar que ninguno de los 6 puntos azules tiene mayor norma que el nuevo punto rojo, por lo tanto, se procede a dividir por 2 al parámetro *step*. Dividir el *step* por 2 provoca que la magnitud de los desplazos por los  $d_j$  al momento de marcar a los candidatos en la esfera se reduzca a la mitad. Entonces, en el próximo paso los candidatos se van a calcular así:

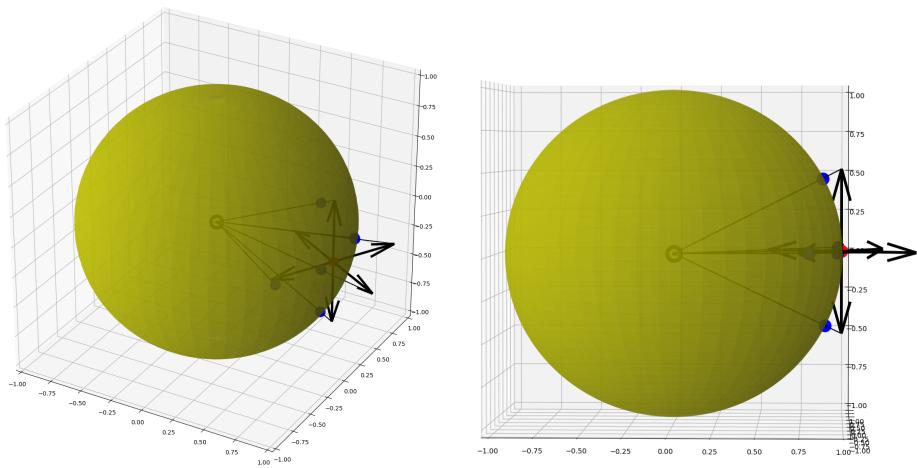


Figura 4.7: Notar como las flechas negras tienen longitud 0,5 mientras que en los pasos anteriores tenían longitud 1.

Esto hace que los candidatos a partir de ahora van a estar más cerca del punto rojo, como podemos ver en las siguientes imágenes:

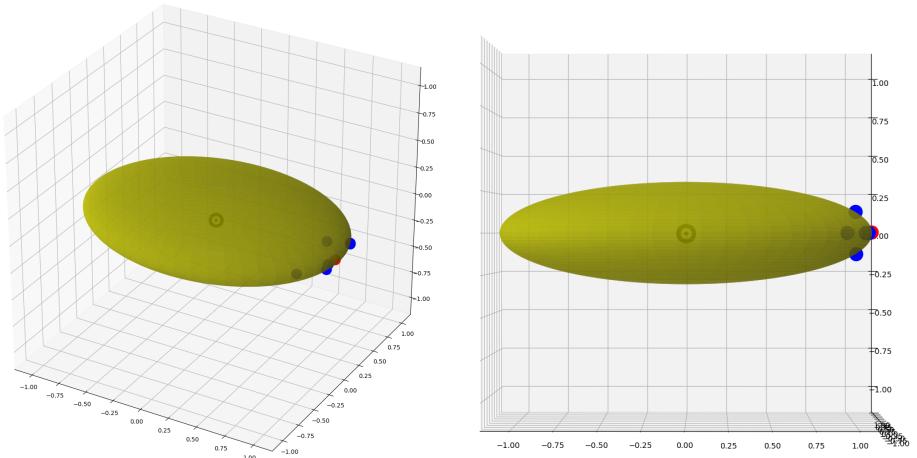


Figura 4.8: Así quedaría un punto rojo con sus 6 candidatos al reducir el tamaño del *step*.

Y así procede la ejecución hasta que  $step$  sea inferior a cierto umbral dado o se supere un número total de iteraciones.

Cuando terminamos de buscar el primer autovector, pasamos al segundo. En este caso, mandamos a 0 la componente del primer autovector hallado, resultando en imágenes como estas:

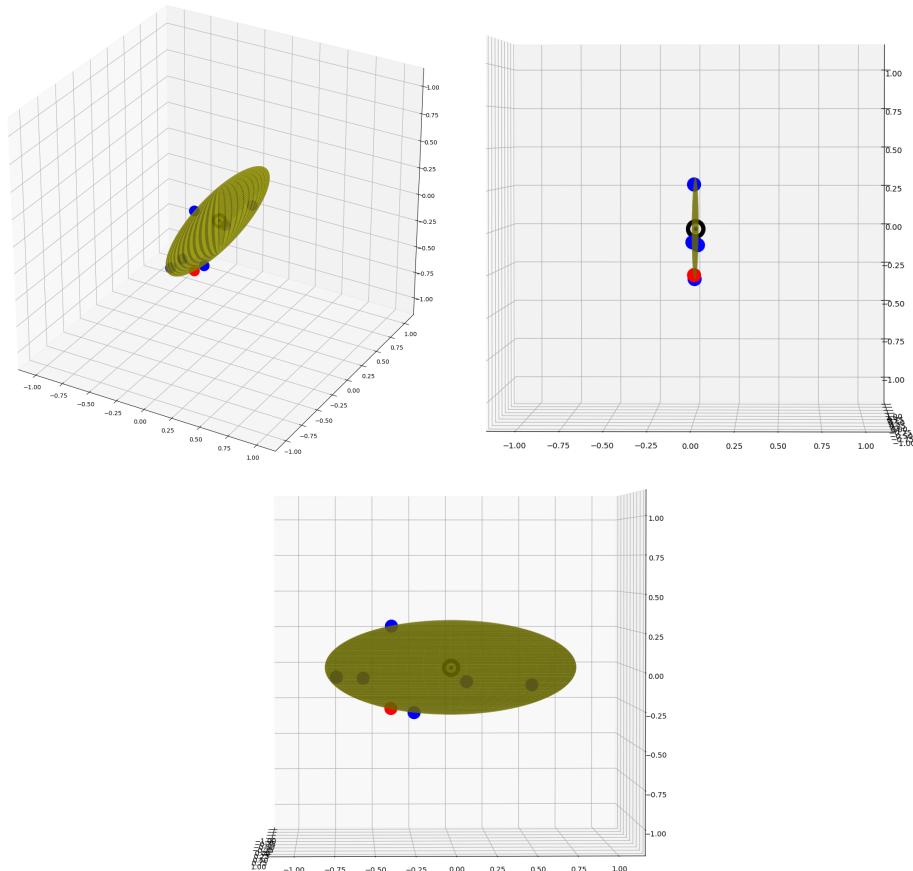


Figura 4.9: Notar como en este caso, sí aparecen puntos azules en la cápsula convexa del elipsoide.

Finalmente, el mismo proceso que vimos antes con  $L_1$  ocurre con  $L_2$  y las demás componentes que sigan.

## 4.2. Teorema de PseudoConvergencia

Dada la intuición ya vista, en esta sección vamos a probar el siguiente teorema:

**Teorema 4.2.1.** *Si ya encontramos  $\{v_j : j < i\}$  y sabemos que  $u_i(\hat{v}_i | v_{j < i})$  no converge a ningún  $\lambda_l$  con  $l > i$ , el algoritmo Evolutive EigenGame Cúbico hace que  $\hat{v}_i \rightarrow v_i$  cuando  $\varepsilon \rightarrow 0, T \rightarrow \infty$ .*

Primero, dado que los autovectores forman una base ortonormal, luego de una serie de rotaciones podemos transformar  $v_1, v_2, \dots, v_n$  en  $e_1, e_2, \dots, e_n$ , por lo que podemos asumir que  $v_j = e_j \forall j$ . Además, como la función  $u_i(\hat{v}_i)$  es no decreciente a lo largo de las iteraciones, basta con probar que la asíntota a la que converge esta sucesión de reales (que es acotada, ya que  $u_i \leq \lambda_i$ ) no puede ser otra cosa que  $\lambda_i$ .

Supongamos entonces que  $u_i(\hat{v}_i) \leq M < \lambda_i$  a lo largo de toda la ejecución del algoritmo. Ahora, dado un  $\delta > 0$ , definimos la “banda de ancho  $\delta$ ” como el conjunto

$$B_\delta = \overline{u_i^{-1}([M - \delta, M])}.$$

Notemos  $B_\delta$  es compacto (pues es cerrado en  $\mathcal{S}^{n-1}$  que es compacto) y que  $e_i \notin B_\delta \forall \delta > 0$ . Además, si  $M \neq \lambda_j \forall j$ , existe un  $\delta_0 > 0$  tal que  $e_j \notin B_\delta$  para todo  $\delta \leq \delta_0$ . Entonces, sabemos que eventualmente la cola de la sucesión de puntos de  $\hat{v}_i$  se mete a la banda  $B_\delta$  para cualquier  $\delta$ , en particular para los  $\delta \leq \delta_0$ .

Como probamos antes, tenemos que  $u_i(x) = \sum_{j \geq i} \lambda_j \alpha_j^2$  donde los  $\alpha_j$  eran las coordenadas de  $x$  en la base de autovectores. Como en este caso la base es la canónica, se tiene que  $u_i(x) = \sum_{j \geq i} \lambda_j x_j^2$ . De esto podemos deducir que entonces

$$\nabla u_i(x) = 2 \cdot (0, 0, \dots, 0, \lambda_i x_i, \lambda_{i+1} x_{i+1}, \dots, \lambda_n x_n).$$

Ahora, recordemos que si estamos parados en  $x$  en el algoritmo, nos movemos en direcciones tangentes a la esfera en  $x$ , o sea en direcciones ortogonales a  $x$ . Para esto, la componente del gradiente que nos interesa es la tangente a la esfera, que también se puede escribir como

$$\nabla u_i(x) - \langle x, u_i(x) \rangle \cdot x.$$

**Propiedad 4.2.2.** *La componente  $\nabla u_i(x) - \langle x, u_i(x) \rangle \cdot x$  es nunca nula en  $B_{\delta_0}$ .*

*Demuestra.* Si  $\nabla u_i(x) - \langle x, u_i(x) \rangle \cdot x = 0$  es porque  $\nabla u_i(x) = \lambda x$  para algún  $\lambda \in \mathbb{R}$ . Escrito de otra forma,

$$(0, 0, \dots, 0, \lambda_i x_i, \lambda_{i+1} x_{i+1}, \dots, \lambda_n x_n) = (\lambda x_1, \lambda x_2, \dots, \lambda x_n).$$

Notemos que esto no es posible si existen dos  $j \geq i$  tales que  $x_j \neq 0$ , ya que los  $\lambda_j$  son todos distintos.

Además, debe haber por lo menos uno, pues  $x_j = 0 \forall j < i$  dada la igualdad. Entonces, esto solo puede suceder si  $x = e_j$  para algún  $j \geq i$ , que es un absurdo, ya que dijimos que ningún  $e_j$  está en  $B_{\delta_0}$ .  $\square$

Gracias a esta propiedad, sabemos que  $\|\nabla u_i(x) - \langle x, u_i(x) \rangle \cdot x\|$  es una función continua y nunca nula en  $B_{\delta_0}$  que es compacto. Por tanto,  $\|\nabla u_i(x) - \langle x, u_i(x) \rangle \cdot x\| \geq \gamma > 0$  en todo  $B_{\delta_0}$ .

Ahora, como  $\{dir_1, dir_2, \dots, dir_{n-1}\}$  es una base ortonormal del plano tangente a la esfera en  $x$ , se tiene que

$$\|y\|^2 = \sum_{j=1}^{n-1} \langle y, dir_j \rangle^2$$

para todo  $y$  en el plano tangente a la esfera en  $x$ . Esto nos dice que existe  $j$  que  $\langle y, dir_j \rangle^2 \geq \frac{\|y\|^2}{n-1}$ , y, por tanto, existe  $j$  tal que  $\langle y, \sigma \cdot dir_j \rangle \geq \frac{\|y\|}{\sqrt{n-1}}$  con  $\sigma \in \{1, -1\}$ . Entonces, existe  $j$  y  $\sigma \in \{1, -1\}$  tal que

$$\begin{aligned} \langle \nabla u_i(x), \sigma \cdot dir_j \rangle &= \langle \nabla u_i(x) - \langle \nabla u_i(x), x \rangle \cdot x, \sigma \cdot dir_j \rangle \\ &\geq \frac{\|\nabla u_i(x) - \langle \nabla u_i(x), x \rangle \cdot x\|}{\sqrt{n-1}} \\ &\geq \frac{\gamma}{\sqrt{n-1}} > 0, \end{aligned}$$

donde la primera igualdad se da porque  $dir_j \perp x$ .

Además, sabemos que la derivada de  $u_i$  en dirección a  $w$  es  $\langle \nabla u_i, w \rangle$ , entonces dado cualquier punto  $x \in B_{\delta_0}$  se tiene al menos una dirección

resultante del Gram-Schmidt del algoritmo que tiene derivada direccional mayor o igual a  $\frac{\gamma}{\sqrt{n-1}}$ . Ahora, por la definición de derivada direccional, existe un  $l_0 > 0$  tal que  $u_i(x + l \cdot (\sigma \cdot \text{dir}_j)) - u_i(x) \geq \frac{\gamma \cdot l}{2 \cdot \sqrt{n-1}}$  para todo  $l \leq l_0$ .

Mejor aún, se puede elegir el  $l_0$  de forma uniforme, es decir, que no dependa del  $x$ . Podemos tomar la función  $U : B_{\delta_0} \times [0, 1] \rightarrow \mathbb{R}$  dada por  $U(x, l) = \max_j \left( \frac{u_i(x + l \cdot (\sigma \cdot \text{dir}_j)) - u_i(x)}{l} \right)$ , que es continua porque el procedimiento *Gram-Schmidt*( $x$ ) es continuo y tomar máximo de  $n - 1$  funciones continuas es continuo. Ahora podemos tomar el abierto  $V = U^{-1} \left( \frac{\gamma}{2 \cdot \sqrt{n-1}}, \infty \right)$  que contiene al compacto  $B_{\delta_0} \times \{0\}$ . Para cada  $x \in B_{\delta_0} \times \{0\}$ , podemos tomar  $\varepsilon_x > 0$  tal que  $B_{2\varepsilon_x}(x) \subset V$ . Como  $\{B_{\varepsilon_x}(x)\}_{x \in B_{\delta_0} \times \{0\}}$  es un cubrimiento por abiertos de  $B_{\delta_0} \times \{0\}$ , podemos tomar finitos  $\{x_m\}_{m=1}^d$  tal que las  $d$  bolas de estos  $x_m$  cubren al compacto  $B_{\delta_0} \times \{0\}$ . Ahora, podemos tomar  $l_0 = \min_{1 \leq m \leq d} \varepsilon_{x_m}$  por lo siguiente: Dado  $x \in B_{\delta_0}$ ,  $(x, l_0)$  está a distancia  $l_0$  de  $(x, 0)$ , que a su vez está a distancia menor a  $\varepsilon_{x_m}$  de  $x_m$  para algún  $m$ . Entonces,  $(x, l_0)$  está a distancia menor a  $l_0 + \varepsilon_{x_m} \leq 2\varepsilon_{x_m}$  de  $x_m$ , lo que dice que  $(x, l_0) \in B_{2\varepsilon_{x_m}}(x_m) \subset V$ .

Recapitulando: gracias a lo que acabamos de probar, sabemos que existe un  $l_0$  tal que si  $\text{step} \leq l_0$  y  $x \in B_{\delta_0}$ , cada iteración del algoritmo va a aumentar  $u_i(x)$  en por lo menos  $\frac{\text{step} \cdot \gamma}{2 \cdot \sqrt{n-1}}$ . Además sabemos que eventualmente siempre se llega a un *step* tan chico como queramos gracias a que solo cambiamos el punto actual si el incremento es por lo menos  $\text{growth\_rate} \cdot \text{step}$ , y como la sucesión de valores  $u_i(\hat{v}_i)$  es creciente y acotada, los incrementos tienden a 0.

Como asumimos que  $u_i(\hat{v}_i) \nearrow M$ , eventualmente  $\hat{v}_i$  entra a  $B_{\delta_0}$  y *step* se hace menor o igual a  $l_0$ . Entonces,  $t$  pasos después de que esto pase,  $u_i(\hat{v}_i)$  va a haber aumentado  $t \cdot \frac{\text{step} \cdot \gamma}{2 \cdot \sqrt{n-1}}$  lo cual es un absurdo, pues entonces después de un número finito de pasos tendremos que  $u_i(\hat{v}_i) > M$ .

# Capítulo 5

## Experimentación

Dada la poca información que tenemos en la teoría acerca de la función  $\mathcal{I}(X, T, step\_inicial, \varepsilon, i)$  antes introducida, vamos a dedicar este capítulo a realizar experimentos que nos den una idea de cómo se comporta esta función, y, por lo tanto, qué cantidad de iteraciones podemos esperar que tarde el algoritmo para calcular cada componente.

### 5.1. Bases de la Experimentación

Los experimentos de esta sección van a consistir en simular un espacio muestral y un cálculo de PCA con EigenGame. Los parámetros del experimento son los siguientes:

1.  $n$ : La dimensión de las muestras. Es decir, las muestras van a ser vectores en  $\mathbb{R}^n$ .
2.  $k$ : La cantidad de muestras en nuestro espacio. Es decir, las muestras van a ser  $k$  vectores.
3.  $C$ : Cantidad de componentes reales que va a tener de fondo nuestro espacio muestral.
4.  $C_E$ : La cantidad de componentes esperadas, o sea la cantidad de componentes principales que vamos a intentar hallar con nuestro experimento. En la mayoría de casos este valor va a ser el mismo que el

parámetro anterior, aunque siempre vamos a pedir  $C_E \leq C$ , ya que no tendría sentido buscar más componentes de las que de antemano sabemos que tiene nuestro espacio.

5.  $r$ : Esta variable de ruido va a ser un valor que vamos a usar para sortear nuestras muestras  $y$ , como su nombre lo indica, darles un factor de ruido o varianza para que las muestras no estén completamente contenidas en el espacio generado por las componentes principales. Cuando veamos como se sortean las muestras en los experimentos va a quedar más clara la función de este parámetro.
6.  $T$ : Esta es la cantidad máxima de iteraciones que vamos a permitir en nuestro algoritmo. Es exactamente la misma  $T$  que vamos a utilizar en *Evolutive-Eigengame*.
7.  $\varepsilon$ : Este es el valor real positivo que vamos a usar para frenar el algoritmo, comparándolo con el valor de *step*. Es exactamente el mismo  $\varepsilon$  que vamos a utilizar en *Evolutive-Eigengame*.
8. *step\_inicial*: Este es el valor real positivo que vamos a usar para inicializar la variable *step* en la ejecución del algoritmo. Es exactamente el mismo *step\_inicial* que vamos a emplear en *Evolutive-Eigengame*.

Al principio de cada experimento, llevamos a cabo la siguiente inicialización:

- Sorteamos  $C$  vectores unitarios en  $\mathcal{S}^{n-1}$  llamados direcciones:  $\{\text{dirección}_i\}_{i=1}^C$
- Sorteamos  $C \cdot k$  coeficientes  $\{w_i^j\}_{i=1, \dots, C}^{j=1, \dots, k}$  con una distribución normal estándar.
- Tomamos las  $k$  muestras dadas por las coordenadas  $w_i^j$  sumándole alguna dirección aleatoria por el coeficiente de ruido. O sea,

$$\text{muestra}_j = \sum_{i=1}^C w_i^j \cdot \text{dirección}_i + r \cdot \text{sesgo}$$

con el vector *sesgo* sorteado en la esfera  $\mathcal{S}^{n-1}$  para todo  $j = 1, \dots, k$ .

- Tomar como  $X \in \mathbb{R}^{k \times n}$  a la matriz cuyas filas son las muestras elegidas.
- Tomar como  $M \in \mathbb{R}^{n \times n}$  a  $X^T \cdot X$ .

**Algorithm 5** Inicialización de los Experimentos

**Input** Las variables  $n$ ,  $k$ ,  $C$ ,  $C_E$ ,  $r$ ,  $T$ ,  $\varepsilon$ ,  $step\_inicial$  antes mencionadas.

```

procedure
  for  $i \leftarrow 1, 2, \dots, C$  do:
     $dirección_i \leftarrow Sortear(\mathcal{S}^{n-1})$ 
  for  $j \leftarrow 1, 2, \dots, k$  do:
     $muestra_j \leftarrow 0$ 
    for  $i \leftarrow 1, 2, \dots, C$  do:
       $w_i^j \leftarrow Sortear(N(0, 1))$ 
       $muestra_j \leftarrow muestra_j + w_i^j \cdot dirección_i$ 
     $sesgo \leftarrow Sortear(\mathcal{S}^{n-1})$ 
     $X_j \leftarrow muestra_j + r \cdot sesgo$ 
   $M \leftarrow X^T \cdot X$ 
end procedure
```

---

Ahora bien, luego de hacer las ejecuciones necesarias de cada experimento, necesitamos una forma de validar si la respuesta final del algoritmo es una buena estimación de los autovectores o no. Para verificar esto, vamos a tomar la matriz  $V \in \mathbb{R}^{C \times n}$  cuyas filas son, en orden, cada uno de los autovectores aproximados que nos devuelve el algoritmo. Con esto vamos a calcular

$$VV^T \quad \text{y} \quad VMV^T.$$

Lo que vamos a pretender de estas matrices es que  $VV^T \approx I_C$  la matriz identidad de dimensión  $C$  y que  $VMV^T \approx \Lambda \in \mathbb{R}^{C \times C}$  una matriz diagonal con  $\Lambda_{ii} > \Lambda_{(i+1)(i+1)} \forall 1 \leq i < C$ .

Como sabemos, un conjunto de vectores es ortonormal si  $\hat{V}\hat{V}^T = I$  donde  $\hat{V}$  es la matriz cuyas filas son los vectores de dicho conjunto. Por lo tanto, la primera condición es una forma de verificar cuán cerca está nuestro conjunto aproximado de autovectores de ser efectivamente una base ortonormal, como es el caso del conjunto verdadero de autovectores.

Si la primera condición está razonablemente cumplida, entonces  $V^T \approx V^{-1}$ , lo que implica que  $VMV^T \approx VMV^{-1}$ . Además, sabemos que el

conjunto de vectores fila de  $V$  es un conjunto de autovectores de  $M$  si  $VMV^{-1}$  es una matriz diagonal. Por consiguiente, la segunda condición nos permite, en caso de que la primera se cumpla, chequear si los vectores resultantes son parecidos a los autovectores verdaderos.

## 5.2. Experimento 1: Cantidad de Muestras

En este primer experimento, nos va a interesar determinar si algunos de los siguientes aspectos afectan al valor de  $\mathcal{I}$ :

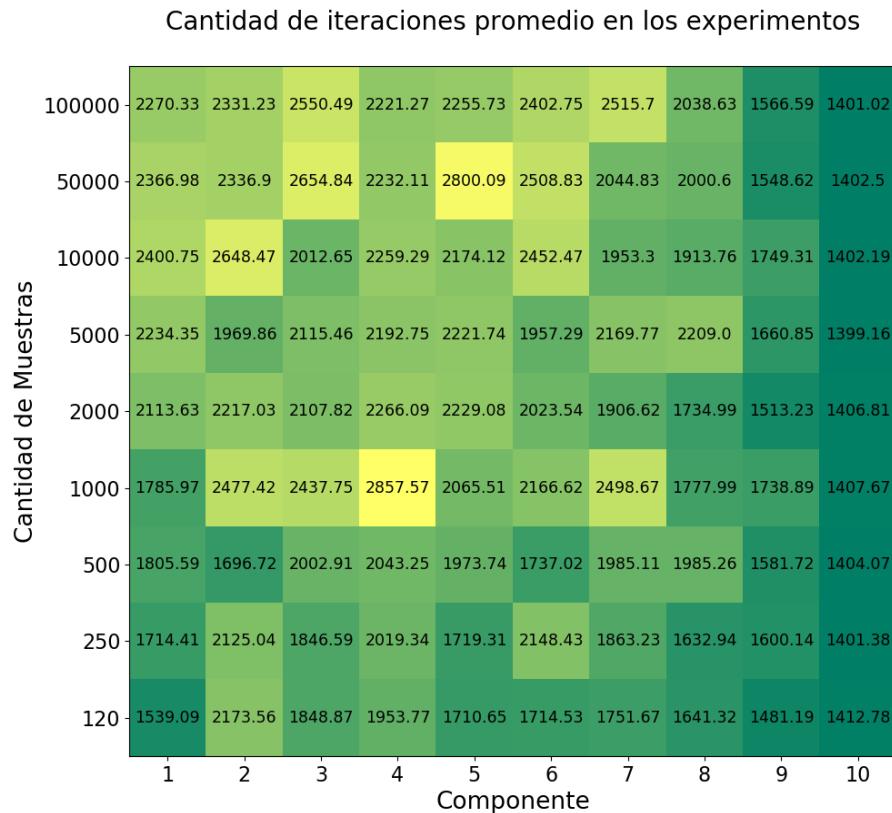
- La cantidad de muestras, valor representado por la variable  $k$ .
- El número de componente que estamos calculando, valor representado por la variable  $i$  al momento de la ejecución del algoritmo. Este aspecto también va a reflejar los cambios de performance del algoritmo respecto a los cambios de la variable  $C_E$ .

Para esto, vamos a hacer simulaciones para correr el *Evolutuve Eigen-Game* variando el parámetro  $k$ . Precisamente, vamos a inicializar las simulaciones con los valores  $n = 120$ ,  $C = C_E = 10$ ,  $r = 1$ ,  $T = 100000$ ,  $step\_inicial = 32$  y  $\varepsilon = 10^{-8}$ . Además, vamos a probar dándole a  $k$  cada uno de los siguientes valores:

$$\{120, 250, 500, 1000, 2000, 5000, 10000, 50000, 100000\}$$

Y, como no podía ser de otra manera, para cada una de las ejecuciones vamos a guardar el valor de  $\mathcal{I}$  resultante para cada una de las posibles configuraciones con  $k \in \{120, 250, 500, 1000, 2000, 5000, 10000, 50000, 100000\}$  y  $1 \leq i \leq 10$ .

Luego de hacer esta simulación 100 veces para cada combinación de parámetros, promediamos todos los valores de  $\mathcal{I}$  obtenidos para cada caso y obtenemos la siguiente tabla:



Lo primero que llama la atención al ver esta tabla es lo poco uniformes que son los valores de  $\mathcal{I}$  encontrados. Esto nos dice que, o bien la función es bastante caótica, o bien la cantidad de experimentos realizada no es suficiente para obtener valores fieles al comportamiento de la función (siendo esta segunda opción la más realista).

Por contraparte, a su vez esta tabla nos estaría indicando que las últimas componentes vendrían a ser las más fáciles de obtener y para las cuales la función  $\mathcal{I}$  fluctúa menos, dado que las columnas son notablemente más uniformes a medida que avanzamos de componente.

Otra conclusión que podríamos afirmar, es que, si bien hay varianza

en los valores obtenidos, todos son del mismo orden. En el experimento exploramos 4 órdenes de magnitud distintos para el valor  $k$ , y, sin embargo, los valores para la cantidad de iteraciones obtenidos rara vez cambiaban de magnitud. En efecto, todos los valores de la tabla se encuentran en el intervalo [1300, 2900].

Ahora bien, estos son los tiempos que le lleva al algoritmo llevar la variable  $step$  a un valor por debajo de  $\varepsilon$ , pero en principio esto no nos dice nada acerca de cuán cerca están las estimaciones resultantes respecto de los autovalores verdaderos. Para esto, vamos a analizar las matrices  $VV^T$  y  $VMV^T$  que obtenemos al usar los vectores arrojados por nuestro algoritmo.

En el caso de  $VV^T$ , en todas las instancias de nuestro experimento obtuvimos una matriz con todas sus entradas en la diagonal iguales a 1, y todas sus entradas fuera de la diagonal con un valor absoluto menor a  $10^{-7}$ . Este margen nos permite asegurar que, salvo errores numéricos, el algoritmo siempre devuelve un conjunto ortonormal de vectores.

En el caso de  $VMV^T$ , la condición de que la secuencia de entradas en la diagonal sea decreciente también se cumplió en cada instancia de nuestro experimento. Y respecto a los términos que no están en la diagonal, su valor absoluto nunca superó el umbral de  $5 \cdot 10^{-5}$ , y además en una misma instancia siempre se cumplió que las entradas de la diagonal tienen 6 órdenes de magnitud más que los de fuera de la diagonal. Es decir, en caso de que haya algunos términos no diagonales de magnitud de  $10^{-5}$ , todos los valores de la diagonal eran de la magnitud por lo menos  $10^1$ .

### 5.3. Experimento 2: Dimensión de las Muestras

Para el segundo experimento, queremos saber de qué manera los siguientes parámetros afectan al valor de  $\mathcal{I}$ :

- La dimensión de las muestras, valor representado por la variable  $n$ .
- El valor de stop que le ponemos a la búsqueda del algoritmo, representado por la variable  $\varepsilon$ .

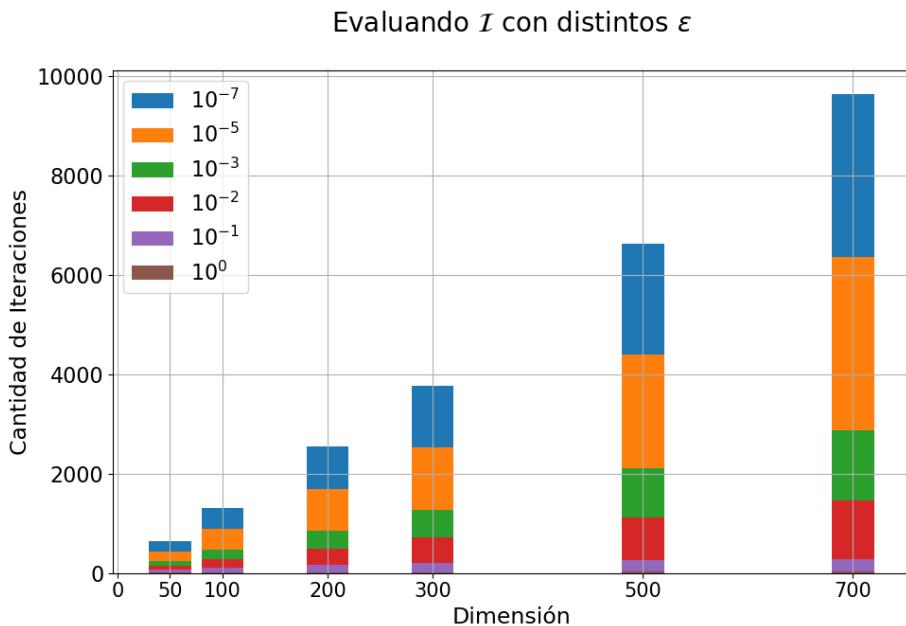
Para esto, vamos a hacer simulaciones para correr el *Evolutuve Eigen-Game* variando el parámetro  $n$ . Precisamente, vamos a inicializar las simulaciones con los valores  $C = C_E = 5$ ,  $r = 1$ ,  $T = 100000$ ,  $step\_inicial = 32$

y  $\varepsilon = 10^{-7}$ . Además, vamos a probar dándole a  $n$  cada uno de los siguientes valores:

$$\{50, 100, 200, 300, 500, 700\}$$

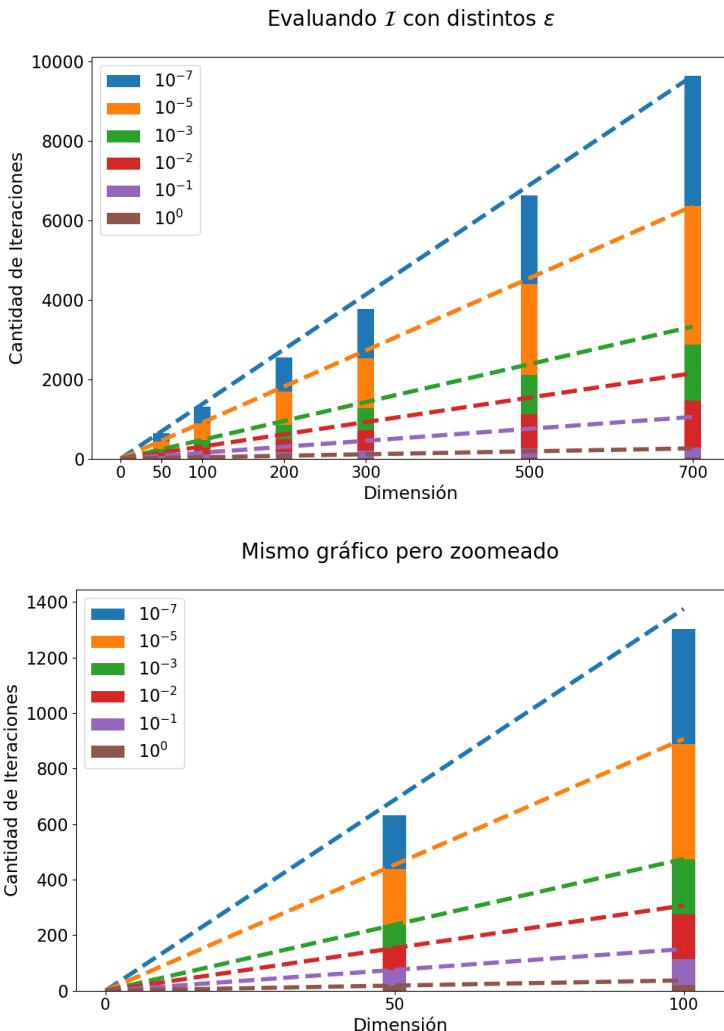
y en cada caso vamos a tomar  $k = 3*n$  para darle una cantidad de muestras razonable al espacio buscado. Los resultados guardados en cada una de las ejecuciones van a ser los valores de  $\mathcal{I}$  cuando *step* pase a ser menor a 1, a 0,1, a 0,01, a 0,001, a  $10^{-5}$ , y a  $10^{-7}$ . Es decir, la cantidad de iteraciones necesarias para el valor del paso con el que cambiamos el candidato termine siendo menor a cada uno de los valores  $\{10^0, 10^{-1}, 10^{-2}, 10^{-3}, 10^{-5}, 10^{-7}\}$ .

Luego de hacer esta simulación más de 60 veces para cada combinación de parámetros, promediamos todos los valores de  $\mathcal{I}$  obtenidos para cada caso y obtenemos el siguiente gráfico:



Algo que podemos observar del gráfico es que, aunque está claro que los valores no son tan precisos dada la poca experimentación realizada, si

podríamos afirmar que la complejidad de  $\mathcal{I}$  dado un  $\varepsilon$  fijo es lineal. Para sustentar este argumento, adjuntamos un gráfico que contiene, para cada valor de  $\varepsilon$ , la recta desde 0 con menor pendiente que pasa a través o por encima de todos los puntos.



Como vemos, en el caso de  $\varepsilon = 10^{-7}$  y  $\varepsilon = 10^{-5}$  el punto que alcanza

la pendiente máxima de la recta es el de dimensión 700. Pero en los demás casos, el valor correspondiente a  $n = 700$  queda por debajo de la respectiva proyección de cada recta. Podríamos atribuirle este hecho a la varianza que tienen los valores de  $\mathcal{I}$  obtenidos, ya que la cantidad de experimentos realizados no es muy alta, pero de todas formas esto nos estaría diciendo que la función  $\mathcal{I}$  no estaría muy lejos de ser sublineal, y, por lo tanto, si  $\varepsilon$  es fijo,  $\mathcal{I}$  estaría en promedio en  $O(n)$ .

Para sostener esta afirmación, quisimos profundizar la experimentación con dimensiones más altas, como por ejemplo con  $n = 1000$ . Dado que con este parámetro se necesita mucho tiempo para completar una ejecución, no pudimos llevar a cabo suficientes experimentos para analizar resultados y por eso no incluimos este caso en los análisis anteriores. Sin embargo, las pocas experimentaciones que se hicieron parecen apoyar lo afirmado anteriormente.

Hicimos 3 experimentos con los siguientes parámetros:

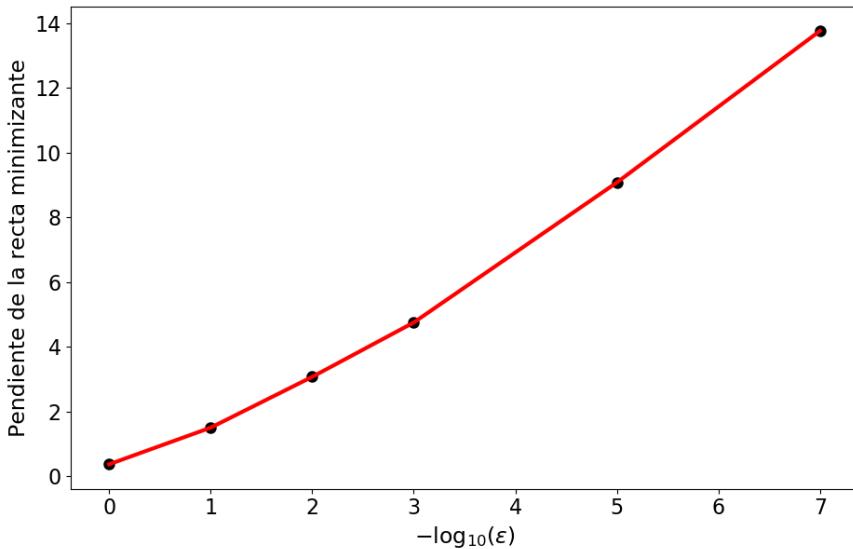
$$n = 1000, k = 3000, C = 10, C_E = 5, r = 1,$$

$$T = 100000, \varepsilon = 10^{-7}, \text{ y } step\_inicial = 32.$$

En la primera ejecución, obtuvimos la primera componente en 10999 iteraciones y la segunda en 19703. En la segunda ejecución, obtuvimos la primera coordenada en 10777, la segunda en 14524 y la tercera en 12009. Y en la tercera ejecución, obtuvimos la primera componente en 12558, la segunda en 11755 y la tercera en 17222. En total, obtuvimos 8 componentes en 109547, lo que da un promedio de 13693,375. Si hiciéramos una proyección para  $n = 1000$  con la información de las mas de 60 ejecuciones exhibidas anteriormente, nos daría un valor muy cercano a 14000, por lo que los valores obtenidos resultan coherentes con la primera parte del experimento.

Ahora, si nos ponemos a hacer un análisis respecto a  $\varepsilon$ , obtenemos que las pendientes de las rectas minimizantes son: 0,37 para  $\varepsilon = 10^0$ , 1,5 para  $\varepsilon = 10^{-1}$ , 3,07 para  $\varepsilon = 10^{-2}$ , 4,75 para  $\varepsilon = 10^{-3}$ , 9,07 para  $\varepsilon = 10^{-5}$  y 13,75 para  $\varepsilon = 10^{-7}$ . Si graficamos estos valores, obtenemos lo siguiente:

Evaluando las pendientes de la recta según la magnitud de  $\varepsilon$



De nuevo, notamos que la pendiente está muy cerca de ser una lineal en ese gráfico. Como el eje horizontal está en función del logaritmo de  $\varepsilon$ , se puede apreciar que la pendiente de la recta minimizante está cerca de estar en  $O(-\log(\varepsilon))$ . Con este resultado, podemos afirmar que los experimentos apuntan a que, en promedio en estos casos,  $\mathcal{I} \in O(-\log(\varepsilon)n)$ .

Pero de nuevo, esto no nos dice nada acerca de los resultados del algoritmo. Para saber si los vectores que nos devuelve el algoritmo son acertados o no, volveremos a analizar las matrices  $VV^T$  y  $VMV^T$ .

En el caso de  $VV^T$  obtuvimos los mismos resultados que en el experimento anterior. En todas las instancias los términos en la diagonal son iguales a 1 y los de fuera de la diagonal tienen valor absoluto menor a  $10^{-7}$ .

Respecto a  $VMV^T$ , la condición de que la sucesión de valores de la diagonal sea decreciente siempre fue cumplida. Además, las entradas no diagonales rara vez superaban  $2 \cdot 10^{-7}$  en valor absoluto y nunca superaron  $4 \cdot 10^{-7}$ .

# Capítulo 6

## Calculando los Últimos Autovalores

### 6.1. Un Nuevo Uso

A fin de cuentas lo que hace nuestro algoritmo es, dada una matriz simétrica, calcular la cantidad de autovectores que le pidas, partiendo desde los de mayor valor hacia los de menor valor. Pero, ¿qué pasa si nosotros queremos calcular los de menor valor? Calcular todos los autovalores llevaría una innecesaria cantidad de tiempo y aumentaría el riesgo de obtener vectores menos fieles gracias a errores numéricos. ¿Hay algún atajo para evitar este problema?

Algo que se puede hacer es invertir la matriz en cuestión, obteniendo así una matriz cuyos autovalores son los inversos a los de la matriz inicial, pero no tendría sentido porque te sale más caro el collar que el perro. Sin embargo, en este capítulo vamos a exponer una solución alternativa sin la necesidad de invertir ninguna matriz.

Si volvemos al capítulo de *Intuiciones*, recordamos que la función utilidad  $u_i$  es en el fondo tomar norma sobre un elipsoide, eventualmente también teniendo en cuenta su cápsula convexa. Después de un cambio de coordenadas ortogonal, el elipsoide para  $u_1$  tiene esta pinta:

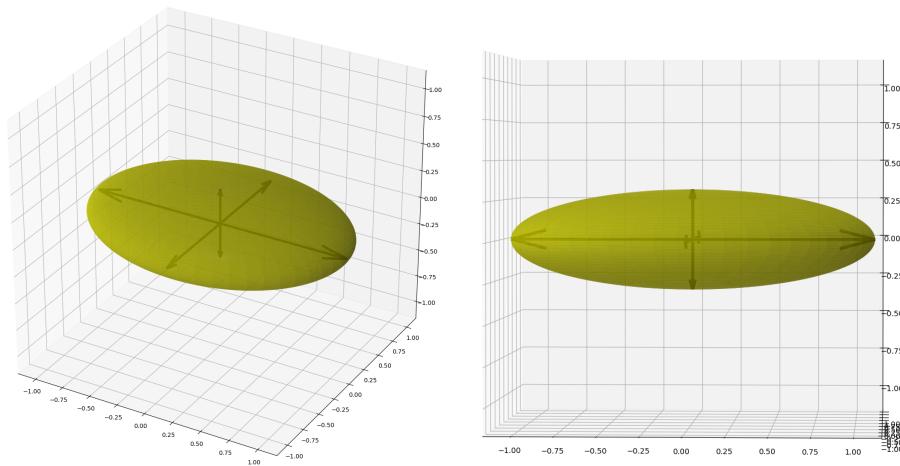


Figura 6.1: Acá se ve el elipsoide con sus 3 ejes de deformación marcados.

Los ejes de deformación son justamente los autovectores que queremos buscar. En los capítulos anteriores queríamos encontrar el autovector de mayor valor y, por tanto, queríamos maximizar la norma. Ahora, si queremos encontrar el autovector de menor valor, basta con minimizar la norma, ya que se sigue cumpliendo que el vector de menor norma en el elipsoide es el eje de deformación con menor magnitud. Es decir, para encontrar el menor autovector  $v_n$  basta con minimizar la función utilidad  $u_1$  en vez de maximizarla. Esta es una modificación muy sencilla a hacer en el *Evolutive EigenGame* original.

Sin embargo, cuando queremos encontrar el **segundo** autovector con menor valor, nos topamos con un problema. Ahora que ya encontramos  $v_n$ , vamos a operar con  $u_2$  que ahora es equivalente a calcular la norma en un elipsoide **o en su cápsula convexa**. Es decir, ahora el  $(0, 0, \dots, 0)$  es un candidato posible a analizar, entonces buscar el vector con menor norma no va a servirnos, ya que sabemos que este vector es el vector 0.

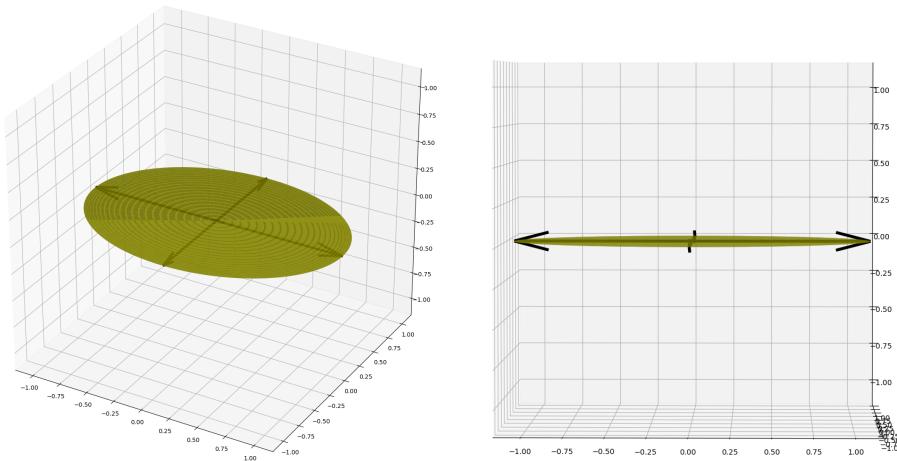


Figura 6.2: Elipsoide de dimensión 1 junto con su cápsula convexa, producto de anular la menor componente al elipsoide de dimensión 2.

Para sortear esto tenemos que encontrar una forma de evitar los vectores de la cápsula convexa y solo analizar vectores en el borde de la variedad, ya que es ahí donde se encuentra nuestro segundo autovector.

Si vamos a la sección de *Visualizacion*, vemos que ahí se explica que los vectores que están en el interior de la cápsula convexa son los que “ pierden norma” cuando las primeras componentes se van a 0 después de aplicar  $L_2$ . Recordemos que

$$L_i(x) = (0, \dots, 0, \sqrt{\lambda_i}x_i, \dots, \sqrt{\lambda_n}x_n).$$

Como las componentes que se van a 0 son las componentes de los autovectores ya hallados, podemos notar que un vector conserva su norma después de aplicarle  $L_i$  si y solo si es ortogonal a todos los autovectores anteriores. Como asumimos que los autovectores son, en orden, los canónicos  $\{e_1, e_2, \dots, e_n\}$ , podemos convencernos de esto observando que un vector es ortogonal a  $e_1, \dots, e_{i-1}$  si y solo si sus primeras  $i-1$  coordenadas son 0, y estos son exactamente los vectores que conservan su norma después de aplicarles  $L_i$ .

Entonces, para evitar el problema mencionado al calcular el  $i$ -ésimo menor autovector cuando  $i > 1$ , podemos pedir que los candidatos a evaluar

siempre sean ortogonales a los  $i - 1$  autovectores anteriores, o directamente agarrar cada candidato y proyectarlo al complemento ortogonal de los vectores anteriores, que es una tarea sencilla y más aun si los vectores tienen norma 1.

Entonces, basados en la misma idea del *Evolutive EigenGame* podemos desarrollar un algoritmo para calcular los menores autovectores de una matriz simétrica. Vamos a llamarlo *Evolutive EigenGame Inverso* y quedaría así:

---

**Algorithm 6** Evolutive-EigenGame Inverso para  $d$  jugadores

---

**Input** El mismo que el Evolutive EigenGame clásico.

**Output** Un conjunto de vectores  $\{\hat{v}_i : 1 \leq i \leq d\}$  estimativos de los  $d$  menores autovectores.

```

procedure
    Proyección_Ortogonal(candidato, eje):
        proyección  $\leftarrow$  candidato  $- eje \cdot \langle candidato, eje \rangle$ 
        if  $\|proyección\| = 0$  do:
            return proyección
        else:
            return  $\frac{proyección}{\|proyección\|}$ 

    Evolutive_EigenGame( $T, step\_inicial, \varepsilon$ ) :
        for  $i \leftarrow 1, 2, \dots, d$  do:
            step  $\leftarrow step\_inicial$ ,  $t \leftarrow 0$ 
            while  $t < T$ ,  $step \geq \varepsilon$  do:
                actual  $\leftarrow \hat{v}_i$ , utilidad_actual  $\leftarrow utilidad(actual)$ 
                for  $j \leftarrow 1, 2, \dots, n$  do:
                    candidato  $\leftarrow \frac{\hat{v}_i \pm step \cdot e_j}{\|\hat{v}_i \pm step \cdot e_j\|}$ 
                    for  $k \leftarrow 1, 2, \dots, i - 1$  do:
                        candidato  $\leftarrow$  Proyección_Ortogonal(candidato,  $\hat{v}_k$ )
                    if  $\|candidato\| = 0$  do:
                        continue
                    utilidad_nueva  $\leftarrow utilidad(candidato)$ 
                    if  $utilidad\_nueva < utilidad\_actual$  do:
                        actual  $\leftarrow candidato$ , utilidad_actual  $\leftarrow utilidad\_nueva$ 
                    if  $actual = \hat{v}_i$  do:
                        step  $\leftarrow \frac{step}{2}$ 
                    else:
                         $\hat{v}_i \leftarrow actual$ 
                return  $\{\hat{v}_i : 1 \leq i \leq d\}$ 
end procedure

```

---

Notar que tenemos que salvar los casos en los que la proyección nos da el vector 0.

La idea de la función *Proyección\_Ortogonal* es que si tenemos un vector *candidato*, lo podemos escribir como

$$\text{candidato} = \alpha \cdot \text{eje} + \beta \cdot z,$$

donde  $z$  es un vector en el complemento ortogonal del eje, o sea  $z \in \langle \text{eje} \rangle^\perp$ . Ahora,  $\langle \text{candidato}, \text{eje} \rangle = \alpha \cdot \|\text{eje}\|^2 + \beta \cdot \langle \text{eje}, z \rangle = \alpha$  si  $\|\text{eje}\| = 1$ . Entonces

$$\text{candidato} - \text{eje} \cdot \langle \text{candidato}, \text{eje} \rangle = \text{candidato} - \text{eje} \cdot \alpha = \beta \cdot z$$

que es justamente la proyección al complemento ortogonal de *eje*. Por tanto, solo queda normalizar y obtenemos la proyección buscada.

## 6.2. Aplicación

Una posible utilidad del algoritmo mostrado en este capítulo es resolver la siguiente ecuación:

$$-u''(x) = (\lambda + V(x))u(x),$$

donde  $u : [0, 1] \rightarrow \mathbb{R}$  función suave tal que  $u(0) = u(1) = 0$  y  $\lambda \in \mathbb{R}$  para una función de distorsión  $V : [0, 1] \rightarrow \mathbb{R}$  dada.

La forma en la que transformamos la ecuación en algo computable es usando la siguiente propiedad:

**Propiedad 6.2.1.** *Si  $u : [0, 1] \rightarrow \mathbb{R}$  es una función  $C^2$ , entonces*

$$\lim_{h \rightarrow 0} \frac{u(x-h) + u(x+h) - 2u(x)}{h^2} = u''(x)$$

para todo  $x \in (0, 1)$ .

Esta propiedad sale de que

$$u''(x) = \lim_{h \rightarrow 0} \frac{u'(x + \frac{h}{2}) - u'(x - \frac{h}{2})}{h}.$$

Ahora  $u'(x + \frac{h}{2}) \approx \frac{u(x+h) - u(x)}{h}$  y  $u'(x - \frac{h}{2}) \approx \frac{u(x) - u(x-h)}{h}$ , entonces

$$\lim_{h \rightarrow 0} \frac{u'(x + \frac{h}{2}) - u'(x - \frac{h}{2})}{h} = \lim_{h \rightarrow 0} \frac{\frac{u(x+h) - u(x)}{h} - \frac{u(x) - u(x-h)}{h}}{h}$$

y esto da lo que queríamos.

Ahora, podemos representar a  $u$  como un vector en  $\mathbb{R}^n$  tomando  $n$  muestras equidistantes en  $[0, 1]$ . Es decir,  $u \in \mathbb{R}^n$  tal que  $u_i = u(\frac{i}{n} - \frac{1}{2n})$   $\forall 1 \leq i \leq n$ , donde el lado izquierdo representa el vector y el derecho la función que estamos buscando. Y con esta perspectiva, podemos definir la segunda derivada discreta como  $u'' \in \mathbb{R}^n$  tal que

$$u''_i = \begin{cases} \frac{u_{i-1} + u_{i+1} - 2u_i}{\frac{1}{n^2}} = n^2(u_{i-1} + u_{i+1} - 2u_i) & \text{si } 1 < i < n \\ \frac{u_{i+1} - 2u_i}{\frac{1}{n^2}} = n^2(u_{i+1} - 2u_i) & \text{si } i = 1 \\ \frac{u_{i-1} - 2u_i}{\frac{1}{n^2}} = n^2(u_{i-1} - 2u_i) & \text{si } i = n \end{cases}$$

Notar que el valor en los bordes sale de que  $u(0) = u(1) = 0$ , entonces el término anterior a  $u_1$  es 0 y el siguiente a  $u_n$  también es 0 y por eso quedan así las coordenadas de los bordes de  $u''$ . Además, observemos como acá nuestro  $h$  vendría a ser  $\frac{1}{n}$  que es la distancia entre las muestras.

Pero ahora, podemos ver al operador “derivada segunda” como una matriz, ya que  $u'' = \Delta u$  donde  $\Delta \in \mathbb{R}^{n \times n}$  tal que

$$\Delta_{ij} = \begin{cases} -2n^2 & \text{si } i = j \\ n^2 & \text{si } i = j \pm 1 \\ 0 & \text{sino} \end{cases}$$

Por último, si dada la función de distorsión  $V$  definimos la matriz  $\Psi \in \mathbb{R}^{n \times n}$  dada por

$$\Psi_{ij} = \begin{cases} V(\frac{i}{n} - \frac{1}{2n}) & \text{si } i = j \\ 0 & \text{sino} \end{cases}$$

entonces traducir el producto de funciones  $V(x)u(x)$  a vectores es lo mismo que hacer el producto de la matriz  $\Psi$  con el vector  $u$ , es decir  $\Psi \cdot u$ .

Con todo esto en juego, podemos traducir el problema continuo a una discretización, que sería resolver el problema

$$-\Delta u = (\lambda Id + \Psi)u.$$

Notar que podemos escribirlo como un problema de autovectores

$$(-\Delta - \Psi)u = \lambda u.$$

Como ambas matrices  $\Delta$  y  $\Psi$  son simétricas, podemos entonces resolver este problema con el algoritmo descrito al principio del capítulo, tomando  $M = -\Delta - \Psi$ , ya que es simétrica.

Pero ¿por qué estamos tan interesados en calcular los menores autovalores? Primero, hay que mencionar un resultado que se conoce sobre este tipo de ecuaciones y que se puede encontrar como el teorema 2.1 en el capítulo 8 del libro *Theory of ordinary differential equations*[2].

**Teorema 6.2.2.** *Las soluciones de la ecuación  $-u'' = (\lambda + V)u$  son una sucesión  $(\lambda_i)_{i \in \mathbb{N}}$  tal que  $\lambda_i \nearrow \infty$ . Además, todos estos autovalores son simples.*

Ahora, el problema discretizado solo puede tener como mucho  $n$  soluciones, ya que  $M$  puede tener como mucho  $n$  autovalores, y estas  $n$  soluciones van a ser las correspondientes a las primeras  $n$  soluciones del problema continuo.

Para ilustrar mejor esto, tomemos un ejemplo. Supongamos que  $V \equiv 0$ , por lo que nos queda la ecuación  $-u'' = \lambda u$  con  $u(0) = u(1) = 0$ . Sabemos que los autovalores para este problema son  $((i \cdot \pi)^2)_{i \in \mathbb{N}}$ , y las autofunciones para  $u$  son  $\sin(i\pi x)$ . Ahora, las soluciones para los primeros valores son de la siguiente forma:

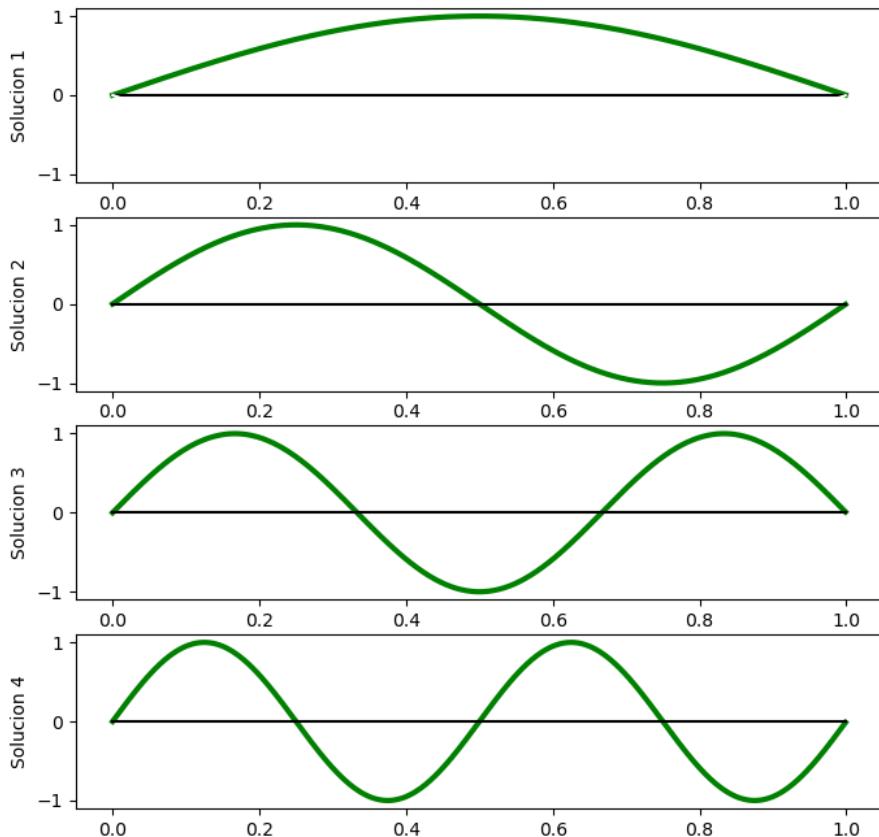
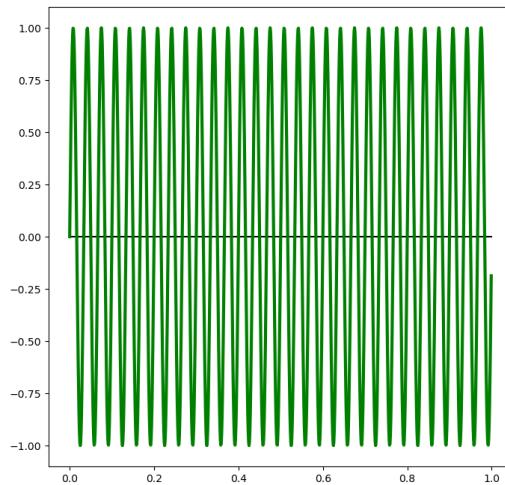


Figura 6.3:  $\sin(i\pi x)$  para cada  $i$  entre 1 y 4. A su vez son las primeras 4 soluciones a la ecuación  $-u'' = \lambda u$ .

Que las primeras soluciones tengan esa forma es beneficioso para la discretización del problema, ya que tomar muestras nos da un vector muy representativo de la función verdadera, dado que no tiene pendientes muy empinadas. Sin embargo, al analizar soluciones de autovalores mayores nos encontramos con funciones no tan amigables. Por ejemplo, esta es la solución para el autovalor número 60:

Figura 6.4:  $\sin(60 \cdot \pi x)$ 

Estas funciones no nos ayudan mucho puesto que, si nosotros queremos resolver el problema discreto en una dimensión no muy alta, por ejemplo en  $\mathbb{R}^{50}$ , tomar 50 muestras de esta función no nos da un vector muy fiel para con la función continua original.

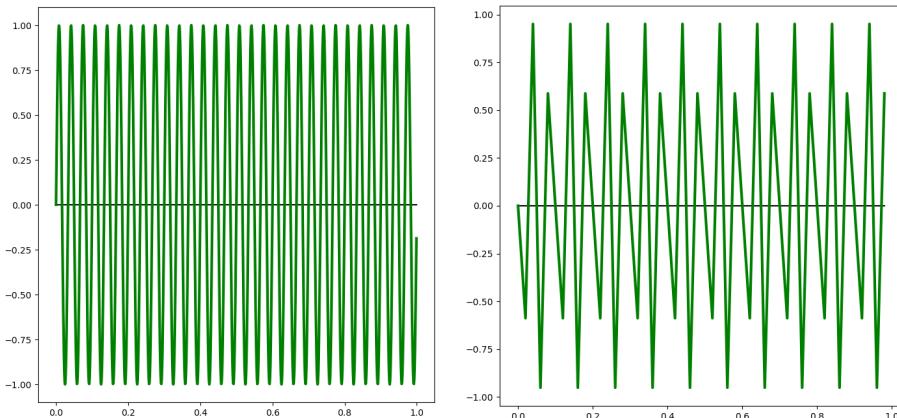


Figura 6.5: Comparación entre la autofunción número 60 y su vector discreto en dimensión 50.

Es por esto que las soluciones más relevantes para resolver el problema discreto son las primeras, y por eso en este caso aplicamos la versión del algoritmo que busca los menores autovalores.

Para ver como funciona todo esto en la práctica, resolvimos este mismo problema con  $V \equiv 0$  en dimensiones 50 y 100. La matriz  $M$  que debíamos darle al algoritmo entonces es simplemente la que tiene  $2n^2$  en cada entrada de la diagonal y  $-n^2$  en las entradas de arriba o de abajo de la diagonal, con  $n = 50$  o  $n = 100$ .

En ambos cálculos seteamos los parámetros  $C_E = 8$ ,  $T = 200000$ ,  $\varepsilon = 10^{-7}$  y  $step\_inicial = 32$ . En el caso de  $n = 50$ , obtuvimos lo siguiente:

Componente ( $i$ )	Iteraciones Necesarias	$\lambda_i$	$(i \cdot \pi)^2$	$\frac{\lambda_i}{(i \cdot \pi)^2}$
1	14202	9.483	9.869	0.96
2	6700	37.897	39.478	0.956
3	10858	85.134	88.826	0.958
4	9236	151.015	157.913	0.956
5	7619	235.289	246.740	0.953
6	6592	337.638	355.305	0.95
7	5660	457.673	483.610	0.946
8	4869	594.939	631.654	0.941

En este caso, cada uno de los autovalores obtenidos por nuestro algoritmo distan de los respectivos autovalores verdaderos de la matriz  $M \in \mathbb{R}^{50 \times 50}$  por menos de  $5 \cdot 10^{-7}$ . Además, si graficamos los vectores que nos devolvió el algoritmo, obtenemos lo siguiente:

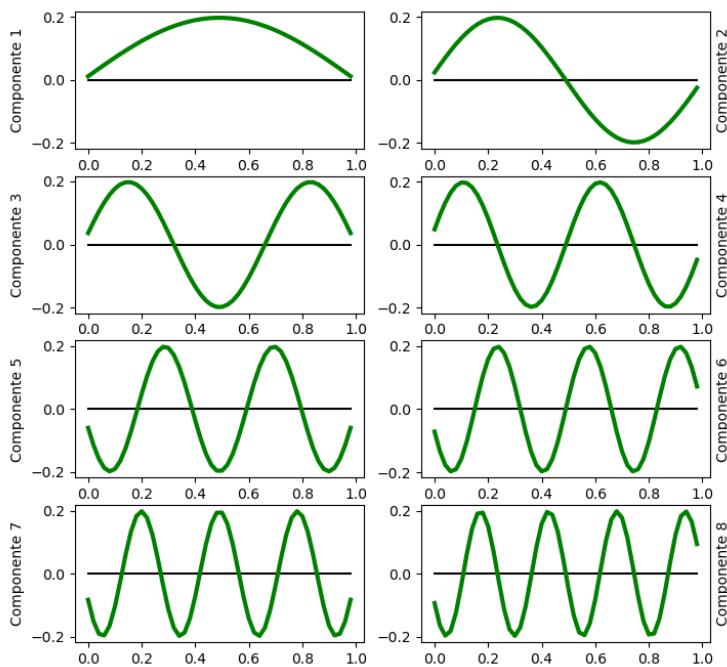


Figura 6.6: Vectores resultantes para el algoritmo en dimensión 50.

El hecho de que el máximo del valor absoluto de las funciones sea 0.2 se puede deber a que el algoritmo solo trabaja con vectores unitarios, entonces puede haber sido dividido por alguna constante. De todas formas, si  $u$  es solución a la ecuación,  $\gamma \cdot u$  también es solución para todo  $\gamma \in \mathbb{R}$ .

En el caso de  $n = 100$ , obtuvimos lo siguiente:

Componente ( $i$ )	Iteraciones Necesarias	$\lambda_i$	$(i \cdot \pi)^2$	$\frac{\lambda_i}{(i \cdot \pi)^2}$
1	120847	9.674	9.869	0.98
2	46334	38.688	39.478	0.98
3	69550	87.013	88.826	0.979
4	60539	154.602	157.913	0.979
5	51735	241.391	246.740	0.978
6	46038	347.295	355.305	0.977
7	40164	472.211	483.610	0.976
8	35320	616.02	631.654	0.975

En este segundo caso, los valores obtenidos distan de los respectivos autovalores verdaderos de la matriz  $M \in \mathbb{R}^{100 \times 100}$  por menos de  $8 \cdot 10^{-7}$ . Además, si graficamos los vectores que nos devolvió el algoritmo, obtenemos lo siguiente:

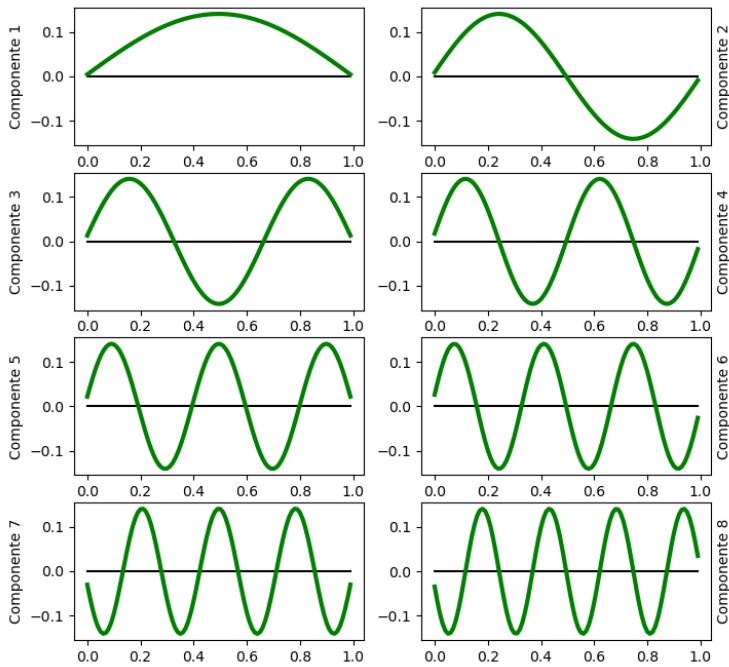


Figura 6.7: Vectores resultantes para el algoritmo en dimensión 100.

Mostrar que, como era de esperar, en este segundo caso obtenemos resultados más cercanos a los del problema continuo tanto para los autovalores como para las autofunciones. Esto es natural, ya que estamos tomando el doble de muestras al discretizar el problema que el primer caso.

Respecto a las autofunciones podemos notar esto particularmente en los bordes, donde en el caso  $n = 100$  estamos mucho más cerca del 0. Además, notar que su valor absoluto máximo es más cercano a 0.1, que es incluso menor que en el caso  $n = 50$ . Esto se debe a que la norma del vector discreto de una misma función aumenta cuando tomamos más muestras, ya que tenemos más términos no nulos en la suma.

Para hacer una última aplicación, resolvimos la misma ecuación pero esa vez con una función de distorsión no nula: Tomamos a  $V$  como una función constante a trozos, para ver como afectaba a las soluciones del problema sin

distorsión. Concretamente, consideramos, de forma totalmente arbitraria, la función

$$V(x) = \begin{cases} 0.06 & \text{si } x \leq 0.35 \\ 240 & \text{si no.} \end{cases}$$

En este caso discretizamos el problema con  $n = 50$ , y entonces nos deja con  $\Psi_{ii} = 0.06$  si  $i \leq 18$  y  $\Psi_{ii} = 240$  si  $i \geq 19$ .

Luego de correr el código en este caso, obtenemos la siguiente tabla de autovalores:

Componente ( $i$ )	Iteraciones Necesarias	Autovalor
1	9852	-220.769
2	6678	-163.853
3	7130	-72.641
4	12207	32.867
5	7098	90.941
6	7031	200.821
7	5603	308.863
8	4977	447.879

Nuevamente, los autovalores devueltos por el algoritmo distan de los verdaderos autovectores de la matriz  $M = -\Delta - \Psi \in \mathbb{R}^{50 \times 50}$  en menos de  $5 \cdot 10^{-7}$ . Además, si graficamos las autofunciones obtenidas, obtenemos lo siguiente:

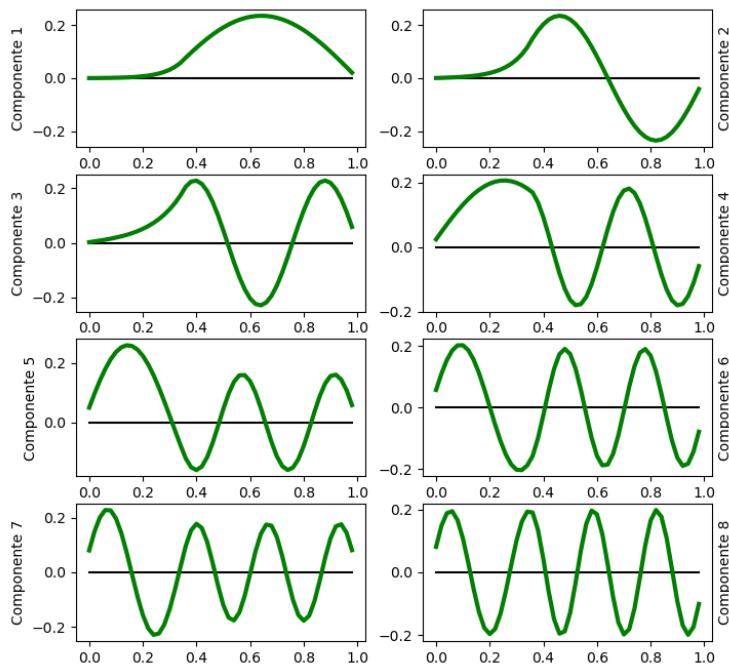


Figura 6.8: Vectores resultantes para el algoritmo en dimensión 50 con una función de distorsión constante a trozos.



## Apéndice A

# Ángulos en Dimensiones Altas

Vamos a demostrar el siguiente teorema:

**Teorema A.0.1.** *Si  $V_n$  es un vector aleatorio sorteado con distribución uniforme sobre  $\mathbf{S}^n$ , entonces*

$$\lim_{n \rightarrow \infty} P(V_n \in \{v \in \mathbf{S}^n : \angle(v, w) < \theta\}) \rightarrow 0$$

para todo  $0 \leq \theta < \frac{\pi}{2}$ .

*Demostración.* Vamos a probar esto simplemente viendo que el cociente de medidas

$$\frac{|\{v \in \mathbf{S}^n : \angle(v, w) < \theta\}|}{|S^n|}$$

tiende a 0 cuando  $n \rightarrow \infty$ .

Primero, sabemos que si  $v$  y  $w$  son unitarios, entonces  $\langle v, w \rangle = \cos(\angle(v, w))$ . Además, pedir que  $\angle(v, w) < \theta$  para algún  $\theta < \frac{\pi}{2}$  es equivalente a pedir que  $\cos(\angle(v, w)) > \epsilon$  para algún  $0 < \epsilon$  en función de  $\theta$ . Por lo tanto, vamos a probar que

$$\lim_{n \rightarrow \infty} \frac{|\{v \in \mathbf{S}^n : \langle v, w \rangle > \epsilon\}|}{|\mathbf{S}^n|} \rightarrow 0$$

para todo  $0 < \epsilon$ .

Para esto, podemos asumir que  $w = e_1 = (1, 0, 0, \dots, 0)$ , ya que las rotaciones preservan ángulos y mantienen medidas de subconjuntos. Ahora,  $\langle v, w \rangle = v_1$ , por lo que nos interesa medir

$$|\{v \in \mathbf{S}^n : v_1 > \epsilon\}|.$$

Ahora, si notamos  $A(n) := |\mathbf{S}^n|$ , el cociente se puede calcular como

$$\frac{1}{A(n)} \cdot \int_{\epsilon}^1 |\{v \in \mathbf{S}^n : v_1 = t\}| dt.$$

Si  $v$  pertenece al conjunto  $\{v \in \mathbf{S}^n : v_1 = t\}$  para algún  $t$ , entonces tenemos que

$$1 = \|v\|^2 = \sum_{i=1}^{n+1} v_i^2 = t^2 + \sum_{i=2}^{n+1} v_i^2,$$

por lo que podemos afirmar que  $\sqrt{\sum_{i=2}^{n+1} v_i^2} = \sqrt{1 - t^2}$ .

Vamos a notar con  $\mathbf{S}_r^d$  a la esfera de radio  $r$  centrada en 0 en  $\mathbb{R}^{d+1}$ . Con lo dicho anteriormente, podemos observar que

$$\{v \in \mathbf{S}^n : v_1 = t\} = \{t\} \times \mathbf{S}_{\sqrt{1-t^2}}^{n-1},$$

y entonces la medida de este conjunto es igual a

$$\left| \mathbf{S}_{\sqrt{1-t^2}}^{n-1} \right| = \left( \sqrt{1-t^2} \right)^{n-1} \cdot |\mathbf{S}_1^{n-1}| = (1-t^2)^{\frac{n-1}{2}} \cdot A(n-1).$$

Volviendo a la integral anterior, la medida de  $\{v \in \mathbf{S}^n : v_1 > \epsilon\}$  es

$$\begin{aligned} \frac{1}{A(n)} \int_{\epsilon}^1 (1-t^2)^{\frac{n-1}{2}} \cdot A(n-1) dt &= \frac{A(n-1)}{A(n)} \int_{\epsilon}^1 (1-t^2)^{\frac{n-1}{2}} dt \\ &\leq \frac{A(n-1)}{A(n)} \int_{\epsilon}^1 (1-\epsilon^2)^{\frac{n-1}{2}} dt \\ &= \frac{A(n-1)}{A(n)} (1-\epsilon^2)^{\frac{n-1}{2}} (1-\epsilon) \\ &\leq \frac{A(n-1)}{A(n)} (1-\epsilon^2)^{\frac{n-1}{2}}. \end{aligned}$$

Ahora, sabemos que  $A(n-1) = (2\pi^{\frac{n}{2}})/\Gamma(\frac{n}{2})$ , entonces

$$\frac{A(n-1)}{A(n)} = \frac{(2\pi^{\frac{n}{2}})/\Gamma(\frac{n}{2})}{(2\pi^{\frac{n+1}{2}})/\Gamma(\frac{n+1}{2})} = \frac{\Gamma(\frac{n+1}{2})}{\sqrt{\pi} \cdot \Gamma(\frac{n}{2})}.$$

Además, sabemos que la función Gamma cumple  $\Gamma(z+1) = z\Gamma(z)^{[1]}$ , entonces

$$\frac{\Gamma(\frac{n+1}{2})}{\sqrt{\pi} \cdot \Gamma(\frac{n}{2})} = \frac{\frac{n-1}{2}}{\sqrt{\pi}} \cdot \frac{\Gamma(\frac{n-1}{2})}{\Gamma(\frac{n}{2})}.$$

Esta misma propiedad nos dice que la función Gamma es creciente en la recta real a partir de un punto, ya que  $\Gamma(z) > 0$  siempre que  $z > 0$ .

Entonces, en el límite cuando  $n \rightarrow \infty$ , se tiene que  $\frac{\Gamma(\frac{n-1}{2})}{\Gamma(\frac{n}{2})} \leq 1$ .

Entonces, recapitulando tenemos

$$\begin{aligned} \lim_{n \rightarrow \infty} \frac{|\{v \in \mathbf{S}^n : \angle(v, w) < \theta\}|}{|\mathbf{S}^n|} &\leq \lim_{n \rightarrow \infty} \frac{A(n-1)}{A(n)} (1 - \epsilon^2)^{\frac{n-1}{2}} \\ &= \lim_{n \rightarrow \infty} \frac{\frac{n-1}{2}}{\sqrt{\pi}} \cdot \frac{\Gamma(\frac{n-1}{2})}{\Gamma(\frac{n}{2})} (1 - \epsilon^2)^{\frac{n-1}{2}} \\ &\leq \lim_{n \rightarrow \infty} \frac{n-1}{2\sqrt{\pi}} (1 - \epsilon^2)^{\frac{n-1}{2}} \\ &= 0 \end{aligned}$$

pues tenemos una exponencial con base  $1 - \epsilon^2 \in (0, 1)$  que converge a 0 más rápido de lo que crece un término lineal  $n-1$ .

□



# Bibliografía

- [1] Emil Artin. *The gamma function*. Courier Dover Publications, 2015.
- [2] Earl A. Coddington and Norman Levinson. *Theory of ordinary differential equations*. Tata McGraw-Hill Education, 1955.
- [3] Drew Fudenberg and Jean Tirole. *Game theory*. MIT press, 1991.
- [4] Ian Gemp, Brian McWilliams, Claire Vernade, and Thore Graepel. EigenGame: PCA as a Nash Equilibrium. *International Conference on Learning Representations (ICLR)*, arXiv preprint arXiv:2010.00554, 2021.
- [5] Ian Gemp, Brian McWilliams, Claire Vernade, and Thore Graepel. EigenGame Unloaded: When playing games is better than optimizing. *International Conference on Learning Representations (ICLR)*, arXiv preprint arXiv:2102.04152, 2022.
- [6] I.T. Jolliffe and Springer-Verlag. *Principal Component Analysis*. Springer Series in Statistics. Springer, 2002.
- [7] Bálint Máté and François Fleuret. Speeding up PCA with priming. arXiv preprint arXiv:2109.03709, 2021.
- [8] Conor Muldoon. DeepMind’s EigenGame Is Not a Game! 2021.