UNIVERSITAT JAUME I

Grado en Matemática Computacional

Estancia en Prácticas y Proyecto Final de Grado

Algunas técnicas matemáticas en la compresión de datos. Aplicación en imágenes digitales.

Autor:
Aleix Alcacer Sales

Supervisor:
Francesc Alted Abad

Tutoras académicas:
Amelia Simó Vidal
Irene Epifanio López

Fecha de lectura: julio de 2018
Curso académico 2017/2018

Resumen

Este documento contiene un resumen de las tareas realizadas en mi estancia en prácticas junto Francesc Alted, así como un desarrollo teórico de técnicas matemáticas aplicadas a la compresión de imágenes.

Durante mi Estancia en Prácticas he desarrollado, para el meta-compresor de datos Blosc, un algoritmo que transforma los datos antes de comprimirlos y los particiona. Esto permite acceder a una parte de los datos comprimidos de una forma mucho más rápida. Se ha hecho todo el desarrollo del código en el lenguaje de programación *Python* y se ha implementado el resultado final en *C*. Aparte de esto, también se ha implementado una interfaz de *Blosc* para poder usarlo desde *Python* y se ha analizado un filtro de *Blosc* cuando se usa de forma reiterada.

Respecto al desarrollo teórico del TFG, este se basa en describir algunas de las técnicas matemáticas usadas en la compresión de imágenes. Por un lado, se han descrito las transformaciones ortogonales y unitarias en dos dimensiones. Además, nos hemos centrado en estudiar dos de ellas: la transformada discreta de Fourier y la transformada discreta del coseno. Por otro lado, se han descrito dos formas de cuantizar los datos: la cuantización escalar y la cuantización vectorial.

Finalmente, se han realizado diversos experimentos relacionados con lo expuesto en el desarrollo teórico del TFG. En concreto, se representaron gráficamente las funciones base de las dos transformaciones, también se implementaron y aplicaron como ilustración en dos ejemplos y, para concluir, se implementó la cuantización vectorial e ilustró con un ejemplo.

Palabras clave

Codificación por transformación, Análisis de componentes principales, DFT, DCT, Análisis de conglomerados

Keywords

Transform coding, Principal component analysis, DFT, DCT, Cluster analysis

Índice general

Capítulo 1

Introducción

1.1. Contexto y motivación del proyecto

El Proyecto Final de Grado junto con la Estancia en Prácticas se enmarcan en el último curso del plan de estudios del Grado de Matemática Computacional.

Al llegar el momento de escoger el destino para realizar la estancia en prácticas, me decanté por hacerlas con Francesc Alted en el Grao de Castellón. Esta elección vino motivada por el hecho de que el mundo de la compresión y la gestión de grandes cantidades de datos era lo que más me apasionaba de todas las alternativas que había.

La compresión de datos se ha usado, desde hace varios siglos, para mejorar y hacer más eficiente la transmisión y el almacenamiento de información. El código Braille, por ejemplo, a la hora de codificar la información, asigna tanto a los caracteres (i. e. 'a') como a algunas de las palabras más usadas (i. e. 'and'), una celda braille (de 6 puntos). De esta forma, se consigue que las palabras más usadas se representen con códigos más cortos que las palabras menos usadas (Salomon and Motta, 2009).

Actualmente, gracias al crecimiento de los dispositivos electrónicos existentes, se generan diariamente enormes cantidades de datos que es necesario transmitir y almacenar de una forma eficiente. Debido a esto, es necesario el uso de compresores de alto nivel a la hora de tratar la información, que permiten la transmisión de datos de una manera mucho más rápida que los compresores tradicionales.

En este Proyecto de Final de Grado, en el Capítulo 2 se detallará todo lo relacionado con la estancia en prácticas con Francesc Alted. En primer lugar, se va a realizar una pequeña

introducción explicando el motivo por el cuál escogí realizar las prácticas con él. Después se detallarán las tareas que realicé durante mi estancia en prácticas así como todo el software que se ha usado para su implementación. Finalmente, se realizará un análisis de los resultados obtenidos y una valoración general sobre mi primera toma de contacto en el mundo laboral.

Por un lado, la primera tarea que tuve que realizar fue implementar una interfaz de *Blosc* para poder usarlo desde el lenguaje de programación *Python*. Esto me permitiría poder usarlo en las siguientes tareas sin tener que programar en *C* y, además, también me serviría para familiarizarme con el funcionamiento de *Blosc* y todas sus características.

La siguiente tarea a realizar fue analizar un filtro de *Blosc*, el filtro 'shuffle', cuando se aplicaba de manera reiterada a los datos. Este filtro aplica una permutación a los bytes (o bits) de los datos para reordenarlos y hacer que los nuevos datos se compriman mejor. Primero se implementó esta versión del filtro y luego se hicieron diversos tests con conjuntos de datos muy diferentes para ver si la tasa de compresión mejoraba. El resultado obtenido fue que no mejoraba los resultados del filtro 'shuffle' original y, por tanto, se descartó su uso.

La implementación de la última tarea fue la que ocupó el grueso de mi estancia en prácticas. En esta tarea se tenía que aplicar un particionamiento (transformación) a conjuntos de datos multidimensionales para mejorar la extracción de datos comprimidos al usar el meta-compresor *Blosc*. La idea de esta transformación consiste en, como *Blosc* comprime los datos por bloques y no todos a la vez, crear particiones de los datos del tamaño de estos bloques y, así, a la hora de descomprimir los datos, solo habrá que descomprimir los bloques cuyas particiones contengan parte de los datos deseados. En la sección 2.4.3, se explicará con mucho más de detalle esta transformación y como son los algoritmos de compresión y descompresión al utilizar esta transformación en *Blosc*.

En el Capítulo 3 se hará un desarrollo teórico, sobre algunas técnicas matemáticas usadas en la compresión, en concreto, en la compresión de imágenes. En primer lugar, en la sección 3.2, se va a realizar una introducción a la compresión, explicando el por qué es tan importante hoy en día, así como se van a definir los conceptos más elementales de la compresión. También se va a introducir una de las técnicas de compresión más usadas en la compresión de imágenes: la codificación por transformación. Posteriormente, se van a describir más profundamente los siguientes dos bloques (de los tres que existen en la compresión): el bloque transformador y el bloque cuantizador.

Por lo que respecta al bloque transformador, éste se encuentra en la sección 3.3. En concreto, en la sección 3.3.1 se van a definir cómo son las transformaciones ortogonales y unitarias tanto para una dimensión como su generalización para dos dimensiones. En esta sección también se van a ver las propiedades que debe cumplir una transformación para que sea eficiente su uso en la compresión. Un vez descrito como son las transformaciones, en las secciones 3.3.2 y 3.3.3 se van a deducir dos de las transformaciones más usadas a partir de sus funciones base: la

transformada discreta de Fourier y la transformada discreta del coseno.

En la sección 3.4 se describe el segundo bloque de la codificación por transformación: el cuantizador. Por un lado, en la sección 3.4.1 se va a explicar la cuantización escalar. Dentro de esta, se va a explicar tanto la cuantización no uniforme como la uniforme así como las relaciones de simetría que existen. Por otro lado, en la sección 3.4.2 se va a introducir la cuantización vectorial, explicando en qué consiste y describiendo un algoritmo para su implementación.

Finalmente, en el Capítulo 4 se hará un análisis de los resultados obtenidos y en el Capítulo 5 se describirán las conclusiones a las que se han llegado. En la sección 4.1 se van a representar gráficamente tanto las funciones base de la transformada de Fourier (la parte real y la parte imaginaria) como las funciones base de la transformada del coseno. También se va a realizar una comparativa del rendimiento de las dos transformaciones. Para ello, se va a aplicar las dos transformaciones a una imagen y se va a recuperar esta imagen a partir de un porcentaje de coeficientes de la transformación obtenida. Por último, se va a realizar un ejemplo de la cuantización vectorial usando distintos tamaños del 'codebook', es decir, estableciendo el número máximo de vectores que se pueden usar para la representación de la imagen.

Capítulo 2

Estancia en prácticas

2.1. Introducción

Francesc Alted es un consultor y desarrollador de software, especializado en los lenguajes de programación *C* y *Python*, que trabaja como autónomo en el Grao de Castellón. Su trabajo consiste, esencialmente, en asesorar a empresas externas para ayudarles a tratar las grandes cantidades de datos que manejan.

Como desarrollador de software, en lo último en lo que ha estado trabajando ha sido *Blosc*, un compresor de datos de alto nivel optimizado para datos binarios implementado en *C*. En realidad, *Blosc* no es un compresor al uso, sino que se le pueden añadir filtros a la hora de comprimir los datos o escoger el compresor (códec) que se va a usar. Es por todo esto por lo que se puede definir como un meta-compresor de datos.

A parte de *Blosc*, Francesc Alted también es el creador de la librería *PyTables*, paquete para manejar, de manera eficiente y fácil, grandes cantidades de datos, o de *bcolz*, paquete que proporciona contenedores de datos que se pueden comprimir en memoria y en disco.

2.2. Software utilizado

Todo este proyecto se ha realizado alrededor del compresor *Blosc* usando el lenguaje de programación *Python*. También se ha usado *Jupyter Notebook* y el software de control de versiones *Git*. El algoritmo final se ha implementado en el lenguaje de programación *C*. A continuación, se darán los detalles de cada uno.

2.2.1. Blosc

Blosc, tal y como se ha descrito anteriormente, es un meta-compresor de datos sin pérdida de información. En este proyecto se va a usar la versión *Blosc2*.

Un compresor sin pérdida de información se caracteriza por la utilización de algoritmos que mantienen toda la información de los datos una vez comprimidos. Por tanto, al descomprimir los datos, se pueden obtener los datos originales sin perder información. Por esta razón es por la que la mayoría de estos algoritmos están basados en la búsqueda de redundancias en los datos, obteniendo una tasa de compresión alta aquellos datos que contengan muchas redundancias.

Blosc tiene dos características que lo hacen diferente del resto de compresores. Por un lado, permite aplicar diversos filtros a los datos antes de comprimir, para obtener unos datos que tengan una mejor tasa de compresión y/o de velocidad. Por otro lado, a la hora de comprimir divide los datos en bloques de datos más pequeños y los comprime por separado. Esto permite realizar la compresión en memoria caché y, así, obtener unas velocidades de compresión más altas.

Para poder comprimir en *Blosc*, como se ha descrito antes, se han de configurar diversos parámetros. Estos parámetros son los siguientes:

- **Códec del compresor**. El códec seleccionado será el algoritmo que vaya a realizar la compresión. Se puede elegir entre *blosclz*, *lz4*, *lz4hc*, *lizard*, *zlib* y *zstd*.
- **Clevel**. Es un número de 0 a 9 que indica el nivel de compresión de los datos. El valor por defecto es 5.
- **Tamaño del tipo de los datos**. Representa el tamaño (en bytes) que ocupa un elemento de los datos a comprimir. El valor por defecto es 8.
- **Número de hilos**[1]. Al ser un compresor multihilos se le puede indicar el número de hilos que use en la compresión. El valor por defecto es 1.
- **Tamaño de bloque**. Como *Blosc* hace la compresión de los datos por bloques, es necesario indicar el tamaño (en bytes) de estos. El 0 indica que se selecciona el tamaño automáticamente.
- **Filtros**. Al permitir la utilización de filtros para transformar los datos antes de la compresión, *Blosc* admite una lista con todos los filtros que se quieren usar. El máximo de filtros simultáneos es 5.

[1]En informática, un hilo es un subproceso que comparte una parte de memoria (el código y los datos) con otros hilos.

2.2.2. Python

Debido a que*Python* ofrece múltiples librerías para poder trabajar con grandes cantidades de datos, se ha escogido como el lenguaje de programación para desarrollar la mayor parte de este proyecto (VanderPlas, 2016). En concreto se ha usado la versión *Python 3.6*.

- **numpy**. Es la librería fundamental para la computación científica. Proporciona estructuras de datos n-dimensionales para almacenar datos numéricos así como múltiples operaciones de álgebra lineal. En este proyecto se ha usado tanto para almacenar como para transformar los conjuntos de datos multidimensionales.
- **matplotlib**. Igual que *numpy* es la librería más usada para el almacenamiento de datos, *matplotlib* es la librería más usada para la creación de gráficas. En este proyecto se ha usado, principalmente, para representar los resultados obtenidos a la hora de comprimir.
- **pycblosc2**. Es una interfaz que permite utilizar *Blosc* desde *Python*. Se ha creado al inicio de esta estancia en prácticas usando la librería CFFI, que permite la interacción desde *Python* con código escrito en *C*.
- **time**. Es un módulo que forma parte de *Python* y que permite el uso de distintas funciones relacionadas con el tiempo. En este proyecto se ha usado para controlar los tiempos de ejecución de los códigos desarrollados.

2.2.3. Jupyter Notebook

Jupyter Notebook es una aplicación web que permite crear y compartir documentos que contienen simultáneamente tanto código y sus resultados como texto llano. Además, aparte de *Markdown*, soporta LaTeXde manera nativa.

Debido a esto, se han utilizado los notebooks para recopilar todo el código desarrollado durante cada quincena y poder compartirlo tanto con el supervisor como con mis tutoras añadiendo anotaciones y/o ejemplos.

2.2.4. Git

Git es un software de control de versiones utilizado para mantener actualizado el código desarrollado. Se ha usado para realizar un seguimiento del desarrollo del proyecto, así como para compartir el código con mi supervisor.

2.3. Objetivos

El objetivo principal del proyecto consiste en implementar un algoritmo que transforme los datos antes de comprimirlos para mejorar los tiempos de descompresión.

Otros objetivos secundarios son:

- Realizar un paquete que permita la interacción desde *Python* con el compresor *Blosc*.
- Aprender a usar todos los paquetes necesarios para el uso de *Blosc* y el análisis de los filtros.
- Analizar un filtro de *Blosc*, 'shuffle', cuando es iterativo, es decir, se usa de manera reiterada.
- Implementar, en caso de ser necesario, la transformación creada en el lenguaje de programación *C*.

2.4. Tareas realizadas

En esta sección se va a describir todas las tareas realizadas durante la estancia en prácticas, las cuales están agrupadas de la siguiente manera: creación del paquete pycblosc2, análisis del filtro 'shuffle' iterativo e implementación de un particionado multidimensional de datos para una lectura más eficiente a través de *Blosc*.

2.4.1. Creación del paquete pycblosc2

Lo primero que realicé durante la estancia en prácticas fue implementar una interfaz que permitiera utilizar *Blosc* desde el lenguaje de programación *Python*. El motivo de realizar esta tarea era doble.

Por una parte, se conseguía poder usar *Blosc* desde *Python*. Esto facilitaría la realización de todos los tests que se necesitarían durante la estancia, pues *Python* tiene diversos paquetes creados especialmente para la realización de tests, así como para mostrar los resultados de una forma más visual que el lenguaje de programación C (lenguaje en el que está escrito *Blosc*).

Por otra parte, la realización de esta tarea, en vistas a que se tendría que usar algunas de sus características en las siguientes tareas, serviría como una primera toma de contacto con *Blosc* y para familiarizarse con su funcionamiento.

Después de consultar con mi supervisor y consultar varias alternativas, decidí implementar la interfaz usando la librería de *Python*, *CFFI*. Esta elección vino motivada por la simplicidad de su uso en este caso en concreto.

Básicamente, el trabajo consistió en crear una función para cada método implementado en *Blosc* y, en caso de ser necesario, modificar el tipo de los datos para hacerlos compatibles con el lenguaje de programación *C*. El código de la interfaz se encuentra en el anexo A.1.1.

Después de implementar toda la interfaz, se creó una batería de tests (anexo A.1.2) para comprobar que el funcionamiento de esta era el correcto. Para ello, se utilizó el framework *unittest*.

2.4.2. Análisis del filtro 'shuffle' iterativo

Una vez realizada la interfaz con la cual poder usar *Blosc* desde *Python*, mi segunda tarea fue analizar el funcionamiento del filtro 'shuffle' de *Blosc* cuando es iterativo.

El filtro 'shuffle' aplica una transformación a los datos antes de comprimirlos. Esta transformación consiste en separar los bytes de cada dato agrupándolos respecto a su nivel de significación.

En la Figura 2.1, se puede observar un ejemplo de esta transformación. Cada casilla representa un byte, el color representa el nivel de significación del byte y el número de cada casilla representa al dato que pertenece cada byte. Por tanto, se observa que en este ejemplo están representados cuatro datos de ocho bytes cada uno.

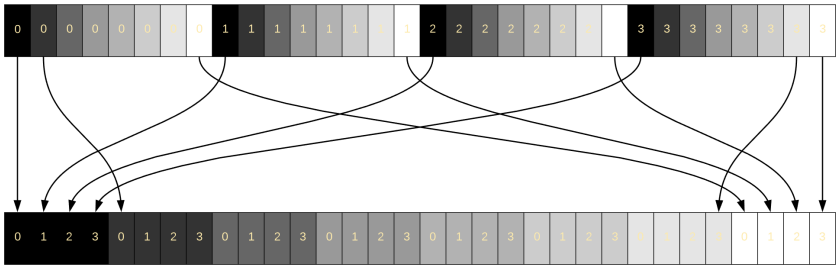


Figura 2.1: Esquema de cómo funciona el filtro 'shuffle'.

De esta forma, se consigue que los bytes de menor significación, que la mayoría de veces son iguales, estén agrupados todos juntos, repercutiendo favorablemente en la tasa de compresión y, muchas veces, también en la velocidad.

En esta tarea, se me pidió analizar cómo se comporta este filtro si se hace iterativo, es decir,

si se puede obtener mejoras en la tasa de compresión al repetirlo n veces.

Al igual que sucede con el filtro 'shuffle' simple, en la Figura 2.2 se representa el funcionamiento del filtro iterativo, que simplemente consiste en aplicar una y otra vez (n veces) el filtro a los datos.

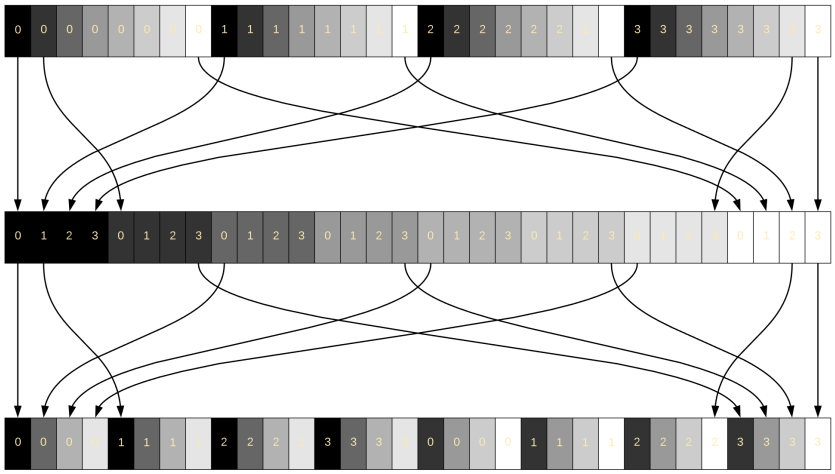


Figura 2.2: Esquema de cómo funciona el filtro 'shuffle' iterativo ($n = 2$).

En primer lugar, como este filtro puede considerarse una permutación de elementos[2], se estudió el orden de la permutación dependiendo del número de datos y del número de bytes de cada dato. Esto facilitaría el posterior análisis, pues ahorraría ir comprobando en cada paso si se ha obtenido la matriz de datos original o no. Esto evitaría la realización de un gran número de operaciones y, por tanto, mejoraría la velocidad de compresión.

El resultado obtenido fue que si $log_2(m)$ divide a $log_2(n)$, entonces $o = log_m(n) + 1$ y, por el contrario, si no lo divide, $o = log_2(n) + log_2(m)$ siendo n el número de datos, m el número de bytes de cada elemento y o el orden de la permutación. Por ejemplo, si $n = 16$ y $m = 4$, se tiene que $o = log_4(16) + 1 = 2 + 1 = 3$.

Después de implementar un método para la obtención del orden de la permutación, se pasó a analizar el comportamiento del filtro en cada una de las iteraciones hasta llegar a los datos iniciales, es decir, hacer o tests. Además, este experimento se reprodujo para diversos tipos de datos como matrices numéricas de enteros, datos climáticos o incluso imágenes.

Un ejemplo de estos experimentos, cuyo código se encuentra en el anexo A.2.1, queda reflejado en la Figura 2.3. En este caso, se ha analizado un conjunto de datos que contiene datos climáticos. Cada línea representa la tasa de compresión en cada iteración del filtro para cada

[2]Las permutaciones se estudiaron en la asignatura *MT1025 - Álgebra Abstracta*.

códec disponible en Blosc.

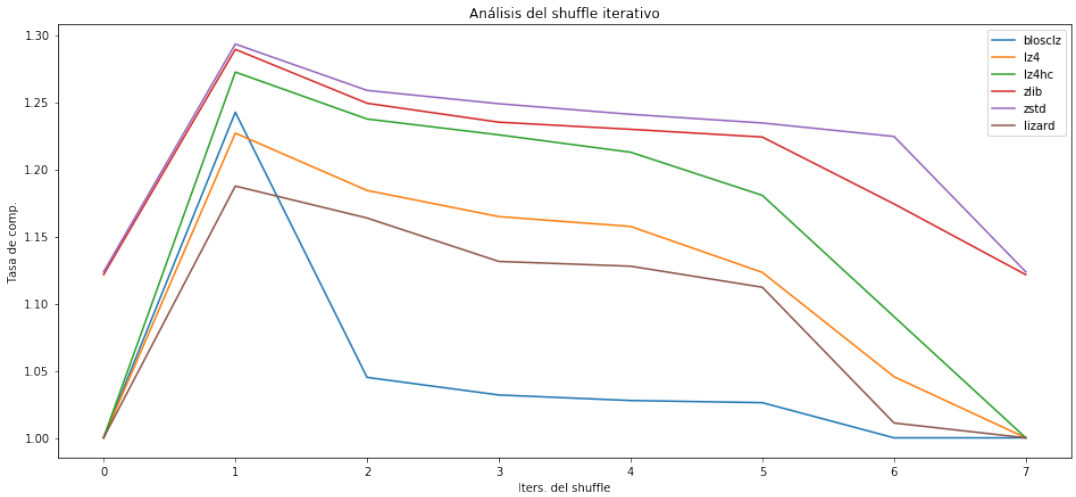


Figura 2.3: Resultado del análisis del 'shuffle' iterativo.

Se observa que la mejor tasa de compresión en cada uno de los códecs se obtiene al aplicar una vez el 'shuffle' a los datos. Este resultado se ha obtenido para la mayoría de las pruebas que se han realizado.

A partir de los resultados de los tests, se llega a la conclusión de que no vale la pena usar el filtro 'shuffle' de manera iterativa en la compresión de datos con *Blosc*. Esto se debe a que, por un parte, casi nunca mejora la tasa de compresión obtenida al aplicar el filtro una vez y, por otra parte, realizar varias iteraciones del 'shuffle' implica un incremento del tiempo empleado en la compresión.

2.4.3. Particionamiento de conjuntos de datos con Blosc

En esta tarea se pedía implementar una transformación a los datos antes de comprimirlos para, a la hora de descomprimir, obtener una mayor velocidad de descompresión.

Para entender la idea de la transformación hay que aclarar que, en memoria, los datos están representados de manera secuencial, es decir, se pierden las dimensiones. Por ejemplo, una imagen se representa como una secuencia de enteros y, por tanto, aunque algunos puntos están muy próximos en la imagen, en la memoria se encuentran alejados.

Otro concepto que hay que tener en cuenta para la transformación es una característica de Blosc. Este meta-compresor, a la hora de comprimir, no comprime todo el conjunto de datos junto, sino que va comprimiendo por bloques pudiéndose establecer el tamaño de estos (siempre

que sea una potencia de 2). Además, el comprimir por bloques permite que, si se quiere acceder a un elemento de un bloque, solo haga falta descomprimir este y no todos los datos, como tienen que hacer los otros compresores.

La idea de la transformación se basa en combinar estos dos conceptos. Por un lado, se crean particiones (todas del mismo tamaño) del conjunto de datos de manera que los elementos de ellas se encuentren próximos en el conjunto de datos original.

Posteriormente, se asigna el tamaño de la partición al tamaño de bloque de Blosc, consiguiendo que cada partición esté en un bloque. Así, cuando se desea descomprimir un subconjunto de los datos comprimidos, solo hay que descomprimir aquellos bloques, es decir, particiones, que contengan los datos deseados.

En la Figura 2.4, está representado lo mencionado anteriormente. El cubo grande representa el conjunto de datos originales, en este caso, de 3 dimensiones; los cubos pequeños representarían a cada una de las particiones realizadas sobre el conjunto de datos original; y, finalmente, el plano azul representaría al subconjunto de datos que se quiere obtener a partir de los datos comprimidos.

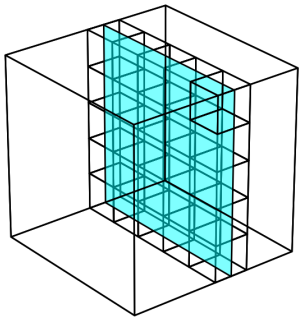


Figura 2.4: Representación de la idea de la transformación.

Por tanto, para obtener el plano azul, es decir, el subconjunto de datos deseado, como cada partición está en un bloque de Blosc, solo hará falta descomprimir aquellos bloques cuya partición contenga parte del subconjunto de datos. Sin embargo, si esto se realizara con otro compresor, habría que descomprimir todos los datos.

Algoritmo de compresión

Cabe destacar, que cada dimensión de los datos originales debe de ser divisible entre la misma dimensión de la partición. En caso contrario, habría particiones de distintos tamaños y no se podría usar en Blosc ya que los bloques deben de ser todos del mismo tamaño. Por

ejemplo, si la forma de los datos es $(8, 17)$ y la forma de las particiones es $(2, 2)$, existirían particiones cuya forma es $(2, 1)$ y, por tanto, distintas de las otras particiones con forma $(2, 2)$.

Para evitar esto, en el algoritmo implementado se creó una función que, dada la forma de las particiones, 'expande' los datos para obtener otro conjunto de datos cuya forma sí es divisible por la forma de las particiones. La función comprueba si el tamaño de una dimensión es divisible entre el tamaño de la misma dimensión en la partición y, en caso negativo, añade ceros (los ceros se comprimen especialmente bien) a esa dimensión hasta que se cumpla la condición.

En la Figura 2.5 se observa como la forma de los datos, $(10, 5)$, no es divisible entre la forma de la partición, $(4, 2)$, pues ni 4 divide a 10 ni 2 divide a 5. Por tanto, estos datos no se pueden particionar correctamente. Por este motivo, se tienen que 'expandir' los datos produciendo otros datos cuya forma sí es divisible, en este caso, $(12, 6)$.

0	1	2	3	4
5	6	7	8	9
10	11	12	13	14
15	16	17	18	19
20	21	22	23	24
25	26	27	28	29
30	31	32	33	34
35	36	37	38	39
40	41	42	43	44
45	46	47	48	49

0	1	2	3	4	0
5	6	7	8	9	0
10	11	12	13	14	0
15	16	17	18	19	0
20	21	22	23	24	0
25	26	27	28	29	0
30	31	32	33	34	0
35	36	37	38	39	0
40	41	42	43	44	0
45	46	47	48	49	0
0	0	0	0	0	0
0	0	0	0	0	0

Figura 2.5: Representación de la 'expansión' de los datos y sus particiones.

Una vez adecuado el conjunto de datos, en el algoritmo implementado, se asigna el tamaño de bloque adecuado a Blosc y se comprimen los datos con él.

Algoritmo para descomprimir

Para descomprimir, se tuvo que crear otra función. Esta calcula los índices de los bloques de Blosc que contienen parte de los datos necesarios y se descomprimen los bloques en un nuevo conjunto de datos. Finalmente, a partir del conjunto de datos descomprimido se seleccionan solo los datos deseados.

Siguiendo el mismo ejemplo anterior, si se tienen ya los datos comprimidos y se quiere extraer la sexta fila de los datos originales, simplemente habrá que descomprimir aquellos bloques que contengan estos datos. Esto está representado en la Figura 2.6.

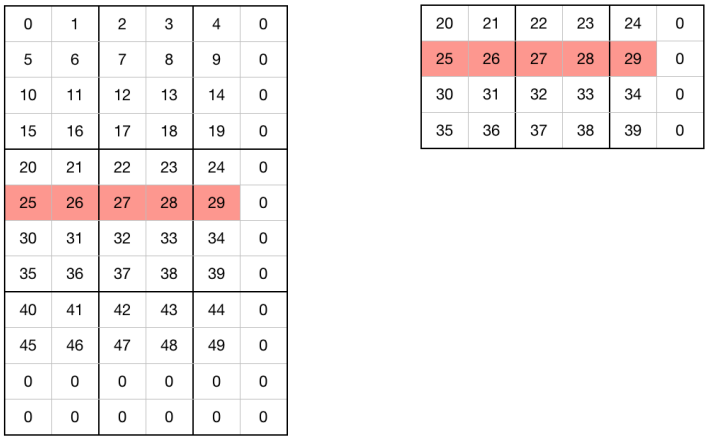


Figura 2.6: Representación de la descompresión de los datos.

Teniendo en cuenta esto último, se observa que definir la forma de las particiones es muy importante, pues de ella dependerá cuantos datos se tendrán que descomprimir sin ser necesarios. Esto se debe a que, por ejemplo, si a un conjunto de datos siempre se suele acceder por filas, será más conveniente hacer particiones que contengan muchas columnas y pocas filas que viceversa.

En el ejemplo usado en la Figura 2.6, si las particiones, en lugar de ser $(4, 2)$, fueran $(2, 6)$, solo habría que descomprimir un bloque en lugar de los tres que han sido necesarios. Además, de esta forma solo se descomprimirían 7 elementos no deseados frente a los 19 elementos que se descomprimirían en el ejemplo original.

Resultados y conclusiones

En el anexo A.3.4 se encuentra un código de ejemplo usando esta transformación que compara el tiempo empleado para descomprimir los datos de la forma usual y el tiempo empleado para descomprimir los datos si, previamente, se ha usado el particionamiento. Este código sigue los pasos anteriormente descritos.

En primer lugar se leen los datos que se van a comprimir y se crea una copia de los datos aplicando la transformación (particionamiento) de los datos.

Después de esto, se comprimen ambas copias de los datos. Por un lado se comprimen los datos originales y, por el otro lado, se asigna como tamaño de bloque de *Blosc* el tamaño de

las particiones creadas y se comprimen los datos particionados. De esta forma se asigna cada partición a un bloque de *Blosc*.

Una vez comprimidos los datos, se define cual es el subconjunto de datos y se descomprimen ambas copias de los datos. La copia que está particionada solo tendrá que descomprimir los bloques necesarios y, luego, seleccionar los datos que se desean. Sin embargo, la copia original tendrá que descomprimir todos los bloques y, al igual que la copia particionada, seleccionar los datos deseados. En cada descompresión se calcula el tiempo empleado.

En el resultado del ejemplo descrito en A.3.4, se observa que el tiempo empleado para la descompresión de los datos particionados es 150 veces (aprox.) más pequeño que el tiempo empleado en la descompresión de los datos comprimidos de la forma habitual.

Después de realizar este test con diversos conjuntos de datos y descomprimiendo diferentes subconjuntos, cabe destacar que la mejora del tiempo de descompresión oscila entre un factor de 0.5 y 1000 veces más rápido. A partir de estos resultados, se concluye que la diferencia entre los tiempos es más grande cuando más pequeño es el subconjunto deseado comparado con el conjunto de datos.

Por tanto, se observa que si se desea descomprimir solo un parte de un conjunto de datos comprimidos conviene aplicar la transformación descrita anteriormente, pues el tiempo de descompresión en este caso es mucho más corto.

2.5. Grado de consecución de los objetivos propuestos

Los objetivos planteados inicialmente se han podido completar casi en su totalidad.

Con respecto a la implementación de la interfaz *pycblosc2*, cabe destacar que se completó totalmente y está disponible para su uso y descarga en el repositorio de Blosc[3].

También se completó el análisis del filtro 'shuffle' al hacerlo iterativo. La conclusión que se extrajo fue que casi nunca se mejoraba la tasa de compresión de la primera iteración y que, por tanto, no hacía falta usarlo.

Finalmente, en lo que se refiere a la transformación de los datos y su integración con *Blosc*, aunque se pudo escribir todo en el lenguaje de programación *C* y funcionaba, se quedó el algoritmo separado en módulos y solo faltó integrarlos en el propio *Blosc*.

[3]https://github.com/Blosc/pycblosc2

2.6. Conclusiones

Esta estancia en prácticas junto con Francesc Alted me ha permitido aprender nuevas metodologías de trabajo. Además, he aprendido a usar varios paquetes de software para trabajar con grandes cantidades de datos.

Por otra parte, aunque he estado supervisado en todo momento por Francesc, la libertad que me ha dado para la realización de las tareas me ha permitido mejorar mis habilidades a la hora de enfrentarme a los problemas.

En resumen, ha sido un primer contacto con el mundo laboral muy satisfactorio en el que he mejorado mis habilidades relacionadas con la programación y con la síntesis y resolución de problemas.

Capítulo 3

Teoría del TFG: Técnicas matemáticas en la compresión de imágenes

3.1. Motivación y Objetivos

Como se ha descrito en la introducción, la compresión de datos, en la actualidad, es un campo que está tomando mucha importancia.

El aumento de los dispositivos electrónicos y las nuevas formas de recolección de datos han implicado que, en el día a día, se generen enormes cantidades de datos, las cuales se tienen que almacenar y tratar de la forma más eficiente posible. Es por ello por lo que la compresión de datos resulta crucial en este ámbito.

Pero la compresión no solo se usa para optimizar el almacenamiento de datos. También tiene suma importancia en el campo de la transmisión de datos, pues si en lugar de datos sin comprimir se transmiten datos comprimidos, la velocidad de transmisión de estos puede aumentar de forma considerable.

Además, como mi estancia en prácticas ha estado relacionada con la compresión, he trabajado durante casi toda la estancia en un filtro para el meta-compresor *Blosc*, he decidido realizar esta parte del TFG sobre técnicas matemáticas utilizadas en la compresión de imágenes.

Primero, en la sección 3.2 se realizará una introducción a la compresión de imágenes. Luego, en la sección 3.3 se desarrollarán de algunas de las transformaciones matemáticas más usa-

das en la compresión. Finalmente, la sección 3.4 se dedicará a la cuantización de imágenes o discretización de sus valores.

3.2. Introducción a la compresión de imágenes

En esta sección, en primer lugar, se va a realizar una descripción general de la compresión, en concreto, de la compresión de imágenes. Posteriormente, se introducirá una de las técnicas más usadas en la compresión de imágenes: la codificación por transformación. Cabe destacar que todo el contenido de esta sección está basado en Tekalp (1995).

La compresión de datos consiste en representar una determinada información empleando una menor cantidad de espacio. Para ello se intenta eliminar toda la redundancia contenida en los datos a comprimir.

Si estos datos son imágenes, en ellos se pueden encontrar dos tipos de redundancias: la estadística y la visual o perceptual. La redundancia estadística está presente en las imágenes a causa de que existen patrones que son más probables de que salgan que otros. En cambio, la redundancia visual (o perceptual) está presente en las imágenes a causa de que hay ciertos tipos de frecuencias de color que el ojo humano no es capaz de diferenciar.

En general, la compresión de imágenes se puede estructurar en tres grandes bloques (Figura 3.1): el transformador, el cuantizador y el codificador.

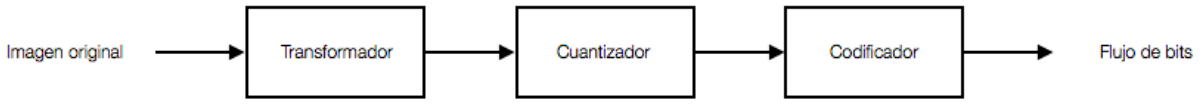


Figura 3.1: Diagrama de bloques de un sistema de compresión de imágenes.

- El transformador aplica, a la imagen original, una transformación uno a uno, es decir, si se aplica la transformación a n elementos, el resultado también será de n elementos. Su función consiste en transformar los datos para obtener otra organización de estos que tenga más redundancia y, así, se pueda comprimir mejor.
- El cuantizador genera un número finito de símbolos que sustituyen a los datos transformados. Es una función muchos a uno y es irreversible. Hay dos tipos de cuantizadores: escalares y vectoriales.
- El codificador asigna un bloque de bits, código, a cada símbolo proporcionado por la salida del cuantizador. El codificador puede usar o bien códigos cuya longitud sea fija o bien códigos cuya longitud sea variable.

Tanto en el bloque de transformación como en el bloque de codificación, teóricamente, no se pierde información. Sin embargo, en el bloque cuantizador sí que se pierde información que luego no se puede recuperar. Debido a esto, los métodos de compresión de imágenes pueden clasificarse en dos grupos:

- Métodos de compresión sin pérdida de información. En estos métodos el bloque cuantizador no se usa. Intentan minimizar el *bitrate*, la tasa promedio de bits, sin perder información de la imagen.
- Métodos de compresión con pérdida de información. Estos métodos, o bien intentan minimizar el *bitrate* dado un nivel máximo de pérdida de información, o bien intentan maximizar la fidelidad del resultado respecto a la imagen original dado un *bitrate* fijo.

3.2.1. Codificación por transformación

La codificación por transformación (Figura 3.2) es una técnica de compresión de imágenes usada en la mayoría de los métodos de compresión con pérdida de información.

Consiste en particionar la imagen en cuadrados más pequeños de $N \times N$ elementos y, a estos, se les aplica la transformación para producir otra matriz de $N \times N$ elementos con los resultados de la transformación. Una vez transformados, estos son cuantificados dependiendo del grado de energía que almacenan y, finalmente, son codificados.

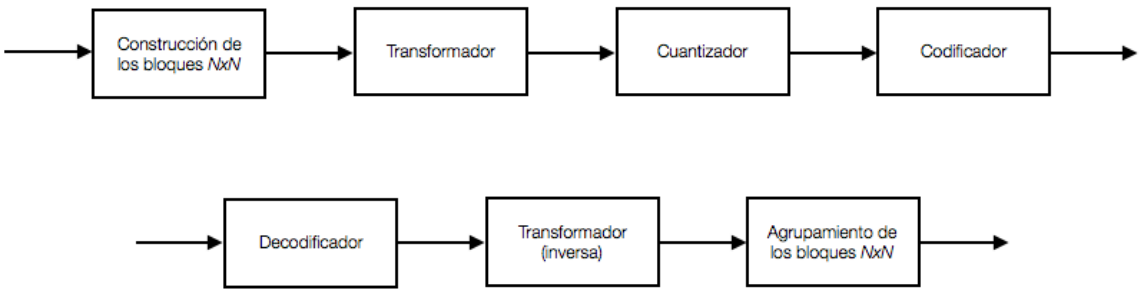


Figura 3.2: Diagrama de bloques de la codificación por transformación.

En los siguientes puntos de este documento se va a realizar un desarrollo más profundo de los dos primeros bloques de la codificación por transformación, es decir, el bloque de la transformación y de la cuantización. En el apartado 3.3 se explicarán las condiciones que debe cumplir la transformación usada en la compresión y, además, se van a describir dos de las más importantes: la transformada discreta de Fourier y la transformada discreta del coseno. En cambio, en el apartado 3.4 se explicará el bloque de la cuantización. En concreto, se realizará

una introducción a ella y se describirá dos tipos de cuantizadores: los cuantizadores escalares y los cuantizadores vectoriales.

3.3. El transformador

En esta sección, la introducción se ha basado en Tekalp (1995). Sin embargo, el contenido de los apartados 3.3.1, 3.3.2 y 3.3.3 está basado, en la mayor parte, en Jain (1989).

La mejor transformación, desde el punto de vista estadístico, para que después el cuantizador sea efectivo, consiste en producir unos coeficientes que estén incorrelacionados y que, además, agrupen la máxima cantidad de información en el menor número de coeficientes.

La transformación que cumple estos dos puntos es la transformación de Karhunen–Loeve (KLT), también conocida como análisis de componentes principales (PCA)[1].

En la KLT se asume que los datos de cada bloque están obtenidos de manera aleatoria y se define la matriz de varianzas covarianzas como $R_x = E(xx^t) = \frac{1}{n}\sum_{i=0}^{n} x_i x_i^t$, donde x_i es un vector obtenido a partir de los píxeles de cada bloque y n el número total de bloques en los que se descompone la imagen.

Al ser R_x una matriz de varianzas-covarianzas y, por tanto, simétrica y definida positiva, esta se puede escribir como $R_x = B^t A B$ donde B es la matriz de los vectores ortonormales de R_x y A es una matriz diagonal con los valores propios de esta. Por cómo esta definida B se sabe que es una matriz unitaria, es decir, $B^t = B^{-1}$.

A partir de todo esto, se puede definir la transformación de Karhunen–Loeve como $y = Bx$ y es fácil comprobar que esta transformación cumple las dos condiciones necesarias:

- Los coeficientes de y están incorrelacionados, pues

$$R_y = E(yy^t) = E(Bx(Bx)^t) = BE(xx^t)B^t = BR_xB^t = A$$

  es una matriz diagonal.

- La pérdida de información a causa de la cuantización se conserva en la KLT, ya que

$$E(||y - \dot{y}||^2) = E((x - \dot{x})^t B^t B(x - \dot{x})) = E((x - \dot{x})^t (x - \dot{x})) = E(||x - \dot{x}||^2)$$

[1] El PCA, análisis de componentes principales, se ha visto en la asignatura *MT1033 - Fundamentos Estadísticos de la Minería de Datos*

donde $\dot{}$ representa la variable cuantificada.

De aquí se obtiene que el error de truncamiento es igual a la suma de la variabilidad que representan los coeficientes descartados (Wintz, 1972).

Sin embargo, esta transformación presenta diversos problemas:

- Sus funciones base dependen de la matriz de varianzas-covarianzas de la imagen. Por tanto, se necesita calcular esta matriz para cada imagen y adjuntarla a cada bloque $N \times N$ para poder realizar la transformación.
- No existen algoritmos eficientes para su cálculo.

Es por estos motivos por lo que, en la práctica, no se usa como transformación el PCA. Se ha visto que mejores opciones para la transformación en la compresión de datos son las transformaciones ortonormales cuyas funciones base no dependen de los datos. Dos de estas funciones son: la transformada discreta de Fourier y la transformada discreta del coseno.

3.3.1. Transformaciones ortogonales y unitarias

En este apartado se va a introducir tanto la ecuación de una transformación ortogonal y unitaria como la ecuación de su inversa.

La notación que se va a usar es la siguiente: se denotará con letras mayúsculas y en negrita a las matrices (por ejemplo, $\mathbf{A}$), con letras minúsculas y en negrita a los vectores (por ejemplo, $\mathbf{u}$) y con letras minúsculas y sin negrita a los elementos (por ejemplo, $v(2)$ o $a_2(5)$).

Transformaciones unitarias en una dimensión

Sea una secuencia de datos (representada como vector)

$$\mathbf{u} = \begin{bmatrix} u(0) & u(1) & \cdots & u(N-1) \end{bmatrix}^T$$

y sea una matriz unitaria

$$\mathbf{A} = \begin{bmatrix} \mathbf{a}_0 \\ \mathbf{a}_1 \\ \vdots \\ \mathbf{a}_{N-1} \end{bmatrix}^T = \begin{bmatrix} a_0(0) & a_0(1) & \cdots & a_0(N-1) \\ a_1(0) & a_1(1) & \cdots & a_1(N-1) \\ \vdots & \vdots & \ddots & \vdots \\ a_{N-1}(0) & a_{N-1}(1) & \cdots & a_{N-1}(N-1) \end{bmatrix}$$

cuya inversa es

$$\mathbf{A}^{-1} = \mathbf{A}^{*T} = \left[\begin{array}{cccc} \mathbf{a}_0^* & \mathbf{a}_1^* & \cdots & \mathbf{a}_{N-1}^* \end{array} \right] = \left[\begin{array}{cccc} a_0^*(0) & a_1^*(0) & \cdots & a_{N-1}^*(0) \\ a_0^*(1) & a_1^*(1) & \cdots & a_{N-1}^*(1) \\ \vdots & \vdots & \ddots & \vdots \\ a_0^*(N-1) & a_1^*(N-1) & \cdots & a_{N-1}^*(N-1) \end{array} \right]$$

Entonces, a partir del vector $\mathbf{u}$ y de la matriz $\mathbf{A}$, se puede definir una transformación unitaria como

$$\mathbf{v} = \mathbf{A}\mathbf{u} \Leftrightarrow v(k) = \sum_{n=0}^{N-1} a_k(n)u(n), \quad 0 \leq k \leq N-1 \tag{3.1}$$

y su inversa como

$$\mathbf{u} = \mathbf{A}^{*T}\mathbf{v} \Leftrightarrow u(n) = \sum_{k=0}^{N-1} a_k^*(n)v(k), \quad 0 \leq n \leq N-1 \tag{3.2}$$

A partir de la definición 3.2, se observa que $\mathbf{u}$ se puede representar como una serie. En concreto, el conjunto de vectores $\{\mathbf{a}_k* : 0 \leq k \leq N-1\}$, es decir, las columnas de la matriz $\mathbf{A}^{*T}$, son los vectores que forman la base de la transformación y los $v(k)$, sus coeficientes.

Dependiendo de la forma de la matriz $\mathbf{A}$, las transformaciones tendrán distintas propiedades y, dependiendo de estas, serán útiles para unos u otros casos. En la compresión de datos, las dos transformaciones cuyas propiedades mejor se adaptan a la compresión de datos son: la transformada discreta de Fourier y la transformada discreta del coseno.

Transformaciones unitarias y ortogonales en dos dimensiones

Lo visto en el anterior apartado se puede extrapolar a dos dimensiones. De esta forma, una transformación ortogonal y unitaria para dos dimensiones se define como

$$v(k_1, k_2) = \sum_{n_1=0}^{N-1} \sum_{n_2=0}^{N-1} a_{k_1,k_2}(n_1, n_2)u(n_1, n_2), \quad 0 \leq k_1, k_2 \leq N-1 \tag{3.3}$$

y su inversa como

$$u(n_1, n_2) = \sum_{k_1=0}^{N-1} \sum_{k_2=0}^{N-1} a_{k_1,k_2}^*(n_1, n_2)v(k_1, k_2), \Leftrightarrow \quad 0 \leq n_1, n_2 \leq N-1 \tag{3.4}$$

De la misma forma que en una dimensión, si $\mathbf{U}$ es la matriz formada por los elementos $\{u(n_1, n_2) : 0 \leq n_1, n_2 \leq N-1\}$ y $\mathbf{A}^*_{k_1,k_2}$ es la matriz formada por $\{a *_{k_1,k_2} (n_1, n_2) : 0 \leq n_1, n_2 \leq N-1\}$, la matriz $\mathbf{U}$ se puede representar como una serie, donde el conjunto de matrices $\{\mathbf{A}^*_{k_1,k_2} : 0 \leq k_1, k_2 \leq N-1\}$ son las matrices que conforman la base de la transformación y los $v(k_1, k_2)$, sus coeficientes.

El conjunto de matrices $\mathbf{A}_{k_1,k_2}$, ha de ser un conjunto ortonormal y completo que debe cumplir las siguientes propiedades:

- Ortonormalidad

$$\sum_{n_1=0}^{N-1} \sum_{n_2=0}^{N-1} a_{k_1,k_2}(n_1, n_2) a^*_{k'_1,k'_2}(n_1, n_2) = \delta(k_1 - k'_1, k_2 - k'_2) \tag{3.5}$$

- Completitud

$$\sum_{k_1=0}^{N-1} \sum_{k_2=0}^{N-1} a_{k_1,k_2}(n_1, n_2) a^*_{k_1,k_2}(n'_1, n'_2) = \delta(n_1 - n'_1, n_2 - n'_2) \tag{3.6}$$

La ortonormalidad asegura que la truncación de cualquier serie de la forma

$$u_{P,Q}(n_1, n_2) = \sum_{k_1=0}^{P-1} \sum_{k_2=0}^{Q-1} a^*_{k_1,k_2}(n_1, n_2) v(k_1, k_2), \quad P, Q \leq N, \quad 0 \leq n_1, n_2 \leq N-1$$

minimizará la suma de los errores al cuadrado

$$\sigma_e^2 = \sum_{n_1=0}^{N-1} \sum_{n_2=0}^{N-1} (u(n_1, n_2) - u_{P,Q}(n_1, n_2))^2$$

siempre que los $v(k, l)$ hayan sido obtenidos a partir de la Ecuación 3.3. Además, la completitud asegura que el error será 0 cuando $P = Q = N$.

Transformaciones unitarias separables

Si se calcula la transformación de la ecuación 3.3 o su inversa (descrita en la ecuación 3.4), el número de operaciones a realizar es $O(N^4)$. Sin embargo, este número se puede reducir a $O(N^3)$ si se usa una transformación separable, es decir,

$$a_{k_1,k_2}(n_1, n_2) = a_{k_1}(n_1) b_{k_2}(n_2) \tag{3.7}$$

donde $\{\mathbf{a}_{k_1} : 0 \leq k_1 \leq N-1\}$ y $\{\mathbf{b}_{k_2} : 0 \leq k_2 \leq N-1\}$ son conjuntos de vectores ortonormales como los que se han visto para una dimensión. Además, las condiciones 3.5 y 3.6 muestran que tanto $\mathbf{A}$ (sus filas son el conjunto de vectores $\{\mathbf{a}_{k_1}^T\}$) como $\mathbf{B}$ (sus filas son el conjunto de vectores $\{\mathbf{b}_{k_2}^T\}$) son unitarias por sí mismas.

Si se escoge que $\mathbf{B} = \mathbf{A}$, entonces la transformación resultante es

$$v(k_1, k_2) = \sum_{n_1=0}^{N-1} \sum_{n_2=0}^{N-1} a_{k_1}(n_1) u(n_1, n_2) a_{k_2}(n_2), \quad 0 \leq k_1, k_2 \leq N-1 \Leftrightarrow \mathbf{V} = \mathbf{A}\mathbf{U}\mathbf{A}^T \tag{3.8}$$

y su inversa

$$u(n_1, n_2) = \sum_{k_1=0}^{N-1} \sum_{k_2=0}^{N-1} a_{k_1}^*(n_1) v(k_1, k_2) a_{k_2}^*(n_2), \quad 0 \leq n_1, n_2 \leq N-1 \Leftrightarrow \mathbf{U} = \mathbf{A}^{*T}\mathbf{V}\mathbf{A}^* \tag{3.9}$$

Ahora se define

$$\mathbf{A}_{k_1,k_2}^* = \mathbf{a}_{k_1}^* \mathbf{a}_{k_2}^{*T}, \quad 0 \leq k_1, k_2 \leq N-1 \tag{3.10}$$

así como el producto interno entre dos matrices

$$< \mathbf{F}, \mathbf{G} >= \sum_{n_1=0}^{N-1} \sum_{n_2=0}^{N-1} f(n_1, n_2) g^*(n_1, n_2) \tag{3.11}$$

Entonces, basándose en las definiciones 3.8 y 3.9, se puede representar $\mathbf{U}$ como una serie de la siguiente forma

$$\mathbf{U} = \sum_{k_1=0}^{N-1} \sum_{k_2=0}^{N-1} v(k_1, k_2) \mathbf{A}_{k_1,k_2}^* = \sum_{k_1=0}^{N-1} \sum_{k_2=0}^{N-1} < \mathbf{U}, \mathbf{A}_{k_1,k_2}^* > \mathbf{A}_{k_1,k_2}^* \tag{3.12}$$

De lo anterior se deduce que cualquier imagen $\mathbf{U}$ es una combinación lineal de las matrices $\mathbf{A}_{k_1,k_2}^*$, también llamadas, imágenes base. Además, los coeficientes de esta combinación lineal son los productos internos entre la imagen y la (k_1, k_2)-ésima imagen base.

De esta forma, se puede expresar cualquier imagen $N \times N$ en conjuntos de N^2 imágenes base. En la compresión, la ventaja de estas transformaciones es que, para recuperar la imagen original, solo hace falta el conjunto de imágenes base y la transformada de la imagen real.

Propiedades deseables de las transformaciones en la compresión de datos

Las propiedades que deben cumplir las transformaciones para su uso en la compresión de datos son:

1. Conservación de la energía. Al aplicar la transformación sobre los datos, ni se aumenta ni se disminuye la cantidad de información que estos contienen.
2. Compactación de la energía. La mayoría de la información está almacenada en pocos coeficientes.
3. Incorrelación. Los coeficientes de la transformación tienden a estar incorrelacionados.

Tanto la transformada discreta de Fourier como la transformada discreta del coseno, que se van a introducir a continuación, cumplen estas propiedades. En ambas transformaciones las funciones base son sinusoidales. Así pues, estas transformadas se localizan en el espectro de frecuencias.

3.3.2. Transformada discreta de Fourier (DFT)

La primera transformada que se va a deducir, usando lo visto anteriormente, va a ser la transformada discreta de Fourier o DFT.

DFT en una dimensión

Para poder obtener la transformada discreta de Fourier unidimensional, se necesita el conjunto de funciones base descrito por

$$a_k^*(n) = \frac{1}{\sqrt{N}} e^{\frac{i2\pi}{N}kn}, \quad 0 \leq k \leq N-1 \tag{3.13}$$

con $0 \leq n \leq N-1$.

Es decir, el conjunto de las funciones base estará formado por N elementos. En este caso, en el que la transformación es discreta, cada elemento de la base será un vector de N elementos $\mathbf{a}_k^* = \left[\begin{array}{cccc} a_k^*(0) & a_k^*(1) & \cdots & a_k^*(N-1) \end{array} \right]^T$ siendo $0 \leq k \leq N-1$.

La representación gráfica de la base de esta transformación se encuentra en la sección 4.1.1.

A partir de las definiciones de transformada descritas en 3.1 y 3.2, se puede definir la transformada discreta de fourier, DFT, como

$$v(k) = \sum_{n=0}^{N-1} u(n) \frac{1}{\sqrt{N}} e^{\frac{-i2\pi}{N}kn}, \quad 0 \leq k \leq N-1 \tag{3.14}$$

y su inversa, IDFT, como

$$u(n) = \sum_{k=0}^{N-1} v(k) \frac{1}{\sqrt{N}} e^{\frac{i2\pi}{N}kn}, \quad 0 \leq n \leq N-1 \tag{3.15}$$

Se comprueba fácilmente que la DFT es una transformación unitaria y ortogonal. En cuanto a las propiedades deseables, el teorema de Parseval (Kammler, 2007) garantiza que la DFT conserva toda la información (propiedad 1).

DFT en dos dimensiones, 2-DFT

En la sección 3.3.1 se ha visto que la base de las transformaciones unitarias viene dada por las N^2 imágenes $\mathbf{A}^*_{k_1,k_2}$ definidas en 3.10. Por ello, las funciones base para la 2-DFT están descritas por

$$a^*_{k_1,k_2}(n_1, n-2) = \frac{1}{N} e^{\frac{i2\pi}{N}(k_1 n_1 + k_2 n_2)}, \quad 0 \leq k \leq N-1 \tag{3.16}$$

con $0 \leq n \leq N-1$.

Ahora la base de esta transformación tendrá N^2 elementos siendo cada uno de estos una matriz compleja. En concreto, cada elemento de las base puede escribirse como $\mathbf{A}^*_{k_1,k_2} = \{a^*_{k_1,k_2}(n_1, n_2) : 0 \leq n_1, n_2 \leq N-1\}$.

La representación gráfica de los elementos de esta base, también se encuentra en la sección 4.1.1.

Basándose en las definiciones 3.8 y 3.9 y en las funciones base definidas en 3.13, se puede definir la transformada discreta de Fourier bidimensional, 2-DFT, como

$$v(k_1, k_2) = \sum_{n_1=0}^{N-1} \sum_{n_2=0}^{N-1} \frac{1}{N} e^{\frac{-i2\pi}{N}(k_1 n_1 + k_2 n_2)}, \quad 0 \leq k \leq N-1 \tag{3.17}$$

y su inversa, 2-IDFT, como

$$u(n_1, n_2) = \sum_{k_1=0}^{N-1} \sum_{k_2=0}^{N-1} \frac{1}{N} e^{\frac{i2\pi}{N}(k_1 n_1 + k_2 n_2)}, \quad 0 \leq k \leq N-1 \tag{3.18}$$

Como puede verse en estas dos ecuaciones, la 2-DFT es separable.

3.3.3. Transformada discreta del coseno (DCT)

Una vez visto cómo se obtiene la transformada discreta de Fourier, se va a obtener otra de las transformaciones más usadas, la transformada discreta del coseno.

DCT en una dimensión

En la transformada discreta del coseno en una dimensión, las funciones base vienen dadas por

$$a_k^*(n) = \begin{cases} \sqrt{\frac{1}{N}}, & k = 0 \\ \sqrt{\frac{2}{N}} \cos\left(\frac{\pi(2n+1)k}{2N}\right), & 1 \leq k \leq N-1 \end{cases} \tag{3.19}$$

con $0 \leq n \leq N-1$.

La base generada tendrá N elementos y cada uno de estos es un vector de la forma $\mathbf{a}_k^* = \begin{bmatrix} a_k^*(0) & a_k^*(1) & \cdots & a_k^*(N-1) \end{bmatrix}^T$ siendo $0 \leq k \leq N-1$.

Al igual que sucede con la transformada discreta de Fourier, la representación gráfica de esta base se encuentra en la sección 4.1.2.

A partir de las definiciones de transformada descritas en 3.1 y 3.2, se puede definir la transformada discreta del coseno, DCT, como

$$v(k) = \sum_{n=0}^{N-1} \gamma(k) u(n) \cos\left(\frac{\pi(2n+1)k}{2N}\right), \quad 0 \leq k \leq N-1 \tag{3.20}$$

y su inversa, IDCT, como

$$u(n) = \sum_{k=0}^{N-1} \gamma(k) v(k) \cos\left(\frac{\pi(2n+1)k}{2N}\right), \quad 0 \leq n \leq N-1 \tag{3.21}$$

siendo $\gamma(k) = \begin{cases} \sqrt{\frac{1}{N}}, & k = 0 \\ \sqrt{\frac{2}{N}}, & 1 \leq k \leq N-1 \end{cases}$.

Es interesante notar que la DCT es equivalente a la DFT de aproximadamente el doble de longitud, cuando opera sobre datos reales con simetría par.

DCT en dos dimensiones, 2-DCT

En este caso, por lo descrito al obtener la transformada de Fourier, las funciones base para la 2-DCT vienen dadas por

$$a^*_{k_1,k_2}(n_1,n_2) = \begin{cases} \frac{1}{N}, & k_1 = 0 \text{ y } k_2 = 0 \\ \frac{\sqrt{2}}{N}\cos\left(\frac{\pi(2n_1+1)k_1}{2N}\right), & k_2 = 0 \text{ y } 1 \leq k_1 \leq N-1 \\ \frac{\sqrt{2}}{N}\cos\left(\frac{\pi(2n_2+1)k_2}{2N}\right), & k_1 = 0 \text{ y } 1 \leq k_2 \leq N-1 \\ \frac{2}{N}\cos\left(\frac{\pi(2n_1+1)k_1}{2N}\right)\cos\left(\frac{\pi(2n_2+1)k_2}{2N}\right), & 1 \leq k_1, k_2 \leq N-1 \end{cases} \tag{3.22}$$

con $0 \leq n_1, n_2 \leq N-1$.

La base resultante consta de N^2 elementos. Cada uno de estos es una matriz $N \times N$ de la forma $\mathbf{A}^*_{k_1,k_2} = \{a^*_{k_1,k_2}(n_1,n_2) : 0 \leq n_1, n_2 \leq N-1\}$.

Como sucede con la representación gráfica de la base de la DCT, la representación gráfica de esta nueva base también se encuentra en 4.1.2.

Basándose en las definiciones 3.8 y 3.9 y en las funciones base definidas en 3.19, se puede definir la transformada discreta del coseno bidimensional, 2-DCT, como

$$v(k_1,k_2) = \sum_{n_1=0}^{N-1}\sum_{n_2=0}^{N-1}\gamma(k_1)\gamma(k_2)u(n_1,n_2)\cos\left(\frac{\pi(2n_1+1)k_1}{2N}\right)\cos\left(\frac{\pi(2n_2+1)k_2}{2N}\right) \tag{3.23}$$

y su inversa, 2-IDCT, como

$$u(n_1,n_2) = \sum_{k_1=0}^{N-1}\sum_{k_2=0}^{N-1}\gamma(k_1)\gamma(k_2)v(k_1,k_2)\cos\left(\frac{\pi(2n_1+1)k_1}{2N}\right)\cos\left(\frac{\pi(2n_2+1)k_2}{2N}\right) \tag{3.24}$$

siendo $\gamma(k) = \begin{cases} \frac{1}{\sqrt{N}}, & k = 0 \\ \sqrt{\frac{2}{N}}, & 1 \leq k \leq N-1 \end{cases}$.

Cabe destacar que la transformada discreta del coseno es la transformación más usada en la compresión de datos. En concreto se usa la 2-DCT con bloques de 8×8 o 16×16 elementos.

La 2-DCT es una transformación ortonormal cuyo conjunto de funciones base es fijo y existen algoritmos eficientes para su cálculo. Comparada con la transformada de Fourier, los coeficientes de esta transformación siempre son reales, mientras que la mayoría de los coeficientes de la DFT son números complejos.

Esta transformación también ofrece un buena compactación de energía, ya que los valores de la imagen varían lentamente entre píxeles consecutivos y, por tanto, la mayoría de las frecuencias espaciales tienen una amplitud casi cero (propiedad 2).

Además, usando la 2-DCT se obtiene una buena cantidad de incorrelación entre los coeficientes de la trnasformación. Esta propiedad se puede ver en el artículo de Clarke (1981), donde se demuestra cómo esta transformación, si se cumplen una serie de condiciones, se aproxima mucho a los resultados ofrecidos por la KLT.

3.4. El cuantizador

La cuantización consiste en transformar un conjunto de datos continuos, es decir, que hay infinitos valores posibles, en otro conjunto de datos en el que solo haya un número finito de valores para estos datos.

En esta sección se va a profundizar en el bloque cuantizador de la transformación por codificación. En primer lugar, se explicará el cuantizador escalar, que aplica la cuantización dato a dato; y, después, se explicará el cuantizador vectorial, que la aplica sobre vectores de datos. Todo este desarrollo se ha hecho a partir de Tekalp (1995)

3.4.1. El cuantizador escalar

Un cuantizador escalar se puede definir como una función, Q, que está definida por unos niveles de decisión, d_i, y por unos niveles de reconstrucción, r_i. Esta función viene dada por

$$Q(s) = r_i \quad \text{si} \quad s \in (d_{i-1}, d_i], \quad 1 \leq i \leq L \tag{3.25}$$

siendo L el número de niveles de valores (o estados) posibles de los datos cuantizados.

Esto significa que, por ejemplo, si el valor de s está entre d_2 y d_3, el valor $Q(s)$, también llamado $\dot{s}$, será igual a r_3.

Si se considera $Q(s) = \dot{s}$, se llama error de cuantización a $e = s - \dot{s}$. El rendimiento de un cuantizador viene dado por una medida de distorsión de los datos, llamada D, que es una función que depende de e y, por ende, depende de los d_i y los r_i. Si los datos provienen de una fuente aleatoria, se puede definir la función D como el error cuadrático medio (MSE) de la cuantización, o dicho de otra forma

$$D = E((s - r_i)^2) \tag{3.26}$$

siendo E la esperanza.

Aunque el error cuadrático medio sea la medida de distorsión más usada, en el caso de imágenes hay otra medida que puede ser mejor que el MSE. Es el caso del error medio perceptual, que trata de calcular el error desde un punto de vista perceptual ya que, luego, vamos a ser los humanos quienes vayamos a ver estas imágenes.

De esta forma, dada una medida de distorsión de los datos, D, que no necesariamente tiene que ser la descrita en 3.26, y dada una función de densidad, $f(s)$, hay dos formas de crear cuantificadores escalares óptimos:

- Fijado un número de niveles L, se procede a encontrar los d_i y los r_i para poder minimizar la función D. Estos cuantizadores, cuando minimizan el error cuadrático medio, se llaman cuantizadores de Lloyd-Max y se encuentran descritos en Lloyd (1982) y Max (1960).
- Fijada una entropía de salida, $H(\dot{s}) = C$, se tiene que encontrar los d_i y los r_i, siendo L desconocido, para poder minimizar la medida de distorsión D escogida. Tal y como se observa en Wood (1969), estos cuantizadores se llaman cuantizadores de entropía constante.

Cuantización no uniforme

Como se ha explicado antes, para poder determinar un cuantizador de Lloyd-Max se necesita minimizar

$$E((s-r_i)^2) = \sum_{i=1}^{L} \int_{d_{i-1}}^{d_i} (s-r_i)^2 f(s) ds \tag{3.27}$$

respecto a d_i y r_i.

En Jain (1989) y en Lim (1990) se demuestra que, para poder minimizar 3.27, son necesarias las siguientes condiciones

$$r_i = \frac{\int_{d_{i-1}}^{d_i} s f(s) ds}{\int_{d_{i-1}}^{d_i} f(s) ds}, \quad 1 \leq i \leq L \tag{3.28}$$

$$d_i = \begin{cases} -\infty, & i = 0 \\ \frac{r_i + r_{i+1}}{2}, & 1 \leq i \leq L-1 \\ +\infty & i = L \end{cases} \tag{3.29}$$

De la condición 3.28 se deduce que los r_i son los centroides de $f(s)$ en los intervalos $(d_{i-1}, d_i]$. Además, de la condición obtenida en 3.29 se obtiene que los d_i son los puntos medios entre r_i y r_{i+1} excepto para d_0 y d_L.

La solución de estas ecuaciones, al ser no uniformes, se ha de obtener mediante el uso de métodos iterativos, como puede ser el algoritmo de Newton-Raphson. Los d_i y r_i obtenidos no están igualmente espaciados y, a consecuencia de esto, se llama a estos cuantizadores no uniformes.

Cuantización uniforme

Un cuantizador uniforme es aquel en que los niveles de decisión están igualmente espaciados, es decir, que

$$r_{i+1} - r_i = \theta, \quad 1 \leq i \leq L-1 \tag{3.30}$$

donde θ es una constante llamada tamaño de paso.

Es muy fácil de ver que un cuantizador de Lloyd-Max se convierte en uno uniforme si la función de densidad, $f(s)$, está distribuida uniformemente en un intervalo finito, $[A, B]$. Esta función viene dada por

$$f(s) = \begin{cases} \frac{1}{A-B}, & s \in [A, B] \\ 0, & s \notin [A, B] \end{cases} \tag{3.31}$$

A partir de $f(s)$, es muy fácil diseñar este cuantizador uniforme

$$\theta = \frac{A-B}{L} \tag{3.32}$$

$$d_i = A + i\theta, \quad 0 \leq i \leq L \tag{3.33}$$

$$r_i = d_{i-1} + \frac{\theta}{2}, \quad 1 \leq i \leq L \tag{3.34}$$

A parte de obtener un cuantizador uniforme cuando $f(s)$ es uniforme en un intervalo, también se puede obtener otro cuantizador uniforme cuando $f(s)$ tiene una cola infinita, como es el caso de la distribución de Laplace. Para obtener este cuantizador habría que añadir la condición 3.30 junto con las condiciones anteriores, 3.28 y 3.29, y, de esta forma, minimizar la ecuación respecto a una sola variable, en este caso, θ.

Relaciones simétricas

Tanto si el cuantizador es uniforme como no uniforme, si la función de densidad $f(s)$ es par, existen relaciones simétricas entre los r_i y los d_i. Además, debido a la paridad de la función de densidad, se puede asumir, sin pérdida de información, que su media es 0.

A continuación, se van a estudiar estas relaciones simétricas dependiendo del valor de L, es decir, dependiendo de si L es par o si L es impar:

- Si L es par, se obtiene que

$$\tilde{r}_i = \begin{cases} r_{i+L/2+1}, & -L/2 \leq i \leq 1 \\ r_{i+L/2}, & 1 \leq i \leq L/2 \end{cases}$$
$$\tilde{d}_i = d_{i+L/2}, \quad -L/2 \leq i \leq L/2$$

  y sus relaciones simétricas son

$$\tilde{r}_i = -\tilde{r}_{-i}, \quad 1 \leq i \leq L/2$$
$$\tilde{d}_i = -\tilde{d}_{-i}, \quad 1 \leq i \leq L/2$$
$$\tilde{d}_0 = 0$$
$$\tilde{d}_{L/2} = \infty$$

- Si L es impar, se obtiene que

$$\tilde{r}_i = r_{i+(L+1)/2}, \quad -(L-1)/2 \leq i \leq (L-1)$$
$$\tilde{d}_i = \begin{cases} d_{i+(L+1)/2}, & -(L+1)/2 \leq i \leq 1 \\ d_{i+(L-1)/2}, & 1 \leq i \leq (L+1)/2 \end{cases}$$

  y sus relaciones simétricas son

$$\tilde{r}_i = -\tilde{r}_{-i}, \quad 1 \leq i \leq (L-1)/2$$
$$\tilde{d}_i = -\tilde{d}_{-i}, \quad 1 \leq i \leq (L+1)/2$$
$$\tilde{r}_0 = 0$$
$$\tilde{d}_{(L+1)/2} = \infty$$

Estas relaciones suelen tenerse en cuenta a la hora de diseñar un cuantizador ya que reducen el número de incógnitas en caso de que la función de densidad, $f(s)$, sea par.

3.4.2. El cuantizador vectorial

Al contrario que la cuantización escalar, que trabaja elemento a elemento, la cuantización vectorial opera con vectores de elementos.

El primer paso para poder aplicar la cuantización vectorial a una imagen, es descomponer esta en un conjunto de vectores $\{\mathbf{s}\}$, donde cada vector $\mathbf{s} = \begin{bmatrix} s_0 & s_1 & \cdots & s_L \end{bmatrix}$ puede representar un bloque de píxeles de la imagen o un bloque de coeficientes de la transformación.

El espacio vectorial formado por todas las combinaciones posibles de vectores, se divide en L regiones R_i, llamadas también regiones de Voronoi (Figura 3.3). Los vectores que caen en la región R_i se representan, en la cuantización, todos por el mismo vector $\mathbf{r}_i$. Al conjunto $C = \{\mathbf{r}_i : 1 \leq i \leq L\}$ se le llama 'codebook'.

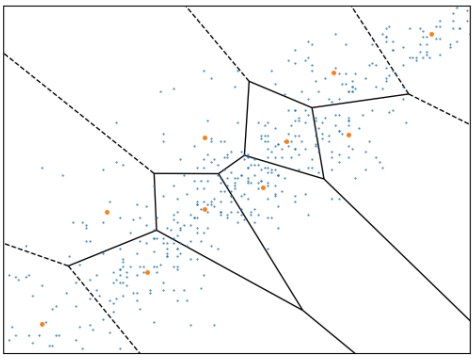


Figura 3.3: Particionamiento de un conjunto de vectores bidimensionales en regiones de Voronoi con sus respectivos centroides.

De esta forma, dado un 'codebook' preestablecido, se puede definir la cuantización vectorial como

$$VQ(\mathbf{s}) = \mathbf{r}_i \quad \text{si} \quad \mathbf{s} \in R_i, \quad 0 \leq i \leq L \tag{3.35}$$

siendo $\mathbf{s}$ el vector a cuantizar.

Diseño del 'codebook' en la cuantización vectorial

Como no es eficiente diseñar un 'codebook' para cada imagen a cuantizar, en la práctica, se diseña un solo 'codebook' para una colección de imágenes parecidas a partir de un conjunto de imágenes representativas.

Por tanto, para diseñar un cuantizador vectorial, dado un conjunto de vectores $\{\mathbf{s}\}$ obtenidos a partir del particionamiento de las imágenes usadas para crear el 'codebook' y dado el tamaño del 'codebook' L, hay que obtener las regiones R_i y los vectores r_i siendo $1 \leq i \leq L$ de tal forma

que se minimice la distorsión dada, por ejemplo, por

$$D = E(||s - r_i||^2) \tag{3.36}$$

Para ello, se usa el algoritmo LGB (se denomina así por sus autores, Linde, Buzo y Gray (Linde et al., 1980)), que es muy similar al método k-means[2], y consiste en los siguientes pasos:

1. Se crea el contador $j = 0$ y se inicializa el 'codebook', $C^j = \{r_1, r_2, \cdots, r_L\}$.
2. Se determinan las regiones R_i, asignando los vectores a su r_i más cercano.
3. Si $\frac{D^{j-1}-D^j}{D^j} < \epsilon$ (D^j es la medida de dispersión definida en 3.36 en la iteración j-ésima), concluye el algoritmo.
4. Se incrementa j, se recalculan los vectores de C^j usando los centroides de R_i y se vuelve al paso 2.

Aunque en cada iteración del algoritmo la distorsión total disminuye, normalmente se suele llegar a un mínimo local. Es por ello que la forma en que se inicializan los vectores del 'codebook' es muy importante. Algunas de estas formas de inicialización son: escoger los L primeros vectores de la muestra o separar aleatoriamente la muestra en L grupos y calcular los centroides de cada uno.

En Jayant and Noll (1990) y Gersho and Gray (1991) se demuestra que la cuantización vectorial ofrece varias ventajas frente a la cuantización escalar. Por ejemplo, la cuantización vectorial puede hacer uso de las correlaciones entre los datos a la hora de cuantizarlos. También ofrece, fijado el número de vectores del 'codebook', menor distorsión respecto a la escalar o, fijado un umbral de distorsión, se necesita un menor número de vectores. En contraposición, la cuantización escalar es más sencilla, pero obtener buenos resultados con ella depende del paso previo, de la transformación.

Implementaciones prácticas de la cuantización vectorial

Tal y como se ha descrito antes, para obtener el vector del 'codebook' que menor distorsión ofrece frente una muestra, hay que comparar la muestra con todos los vectores del 'codebook'. Es por ello, que en la práctica, se utilizan otros esquemas más eficientes como la cuantización vectorial basada en árboles estructurados o el producto de cuantizadores vectoriales.

[2]El método de clasificación no supervisada k-means se ha visto en la asignatura *MT1033 - Fundamentos Estadísticos de la Minería de Datos.*

Capítulo 4

Experimentos realizados a partir del desarrollo teórico

4.1. Representación de las bases de las transformaciones

En esta sección se van a representar gráficamente las funciones base descritas en las secciones 3.3.2 y 3.3.3, es decir, las funciones base de la transformada discreta de Fourier y de la transformada discreta del coseno.

Para poder representarlas, se ha implementado el código descrito en en anexo A.4.

4.1.1. Transformada discreta de Fourier

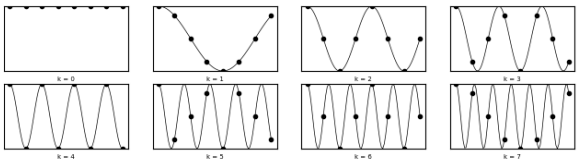
En las Figuras 4.1 y 4.2 están representadas las bases de la transformada discreta de Fourier tanto para una dimensión como para dos (2-DFT).

DFT

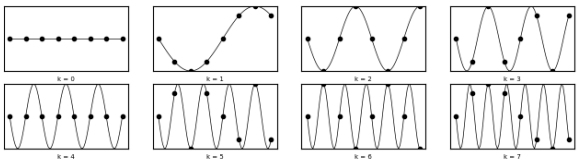
En la Figura 4.3 están representadas las funciones base de la DFT. En concreto, en la Figura 4.1a se encuentra representada la parte real de las funciones base y, en la Figura 4.1b, la parte imaginaria.

Los conjuntos de puntos son los vectores que forman la base de la transformada en caso de

que esta sea discreta. Sin embargo, las líneas son la representación de las funciones que forman la base en caso de que la transformada de Fourier sea continua.



(a) Parte real.

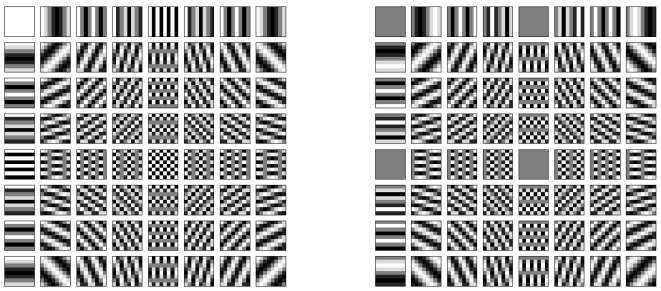


(b) Parte imaginaria.

Figura 4.1: Representación de las funciones base de la DFT para $N = 8$.

2-DFT

En este caso, se ha representado el conjunto de funciones base (matrices) de la transformada discreta de Fourier como un conjunto de imágenes. En la Figura 4.2 están representadas el conjunto de 64 imágenes que forman la base. En concreto, en la Figura 4.2a se encuentra la parte real de estas funciones (o imágenes) y en la Figura 4.2b, la imaginaria.



(a) Parte real. (b) Parte imaginaria.

Figura 4.2: Representación de las funciones base de la 2-DFT para $N = 8$.

4.1.2. Transformada discreta del coseno

Igual que se ha hecho con la trasformada discreta de Fourier, en las Figuras 4.3 y 4.4 se encuentran representadas gráficamente las funciones que forman la base de esta transformada.

DCT

En la Figura 4.3 se encuentra la representación de las funciones base de la DCT. Los conjuntos de puntos son una representación de los vectores de la base de la DCT. Las líneas, sin embargo, son una representación de las funciones de la base en caso de que la transformada sea continua, es decir, de la transformada del coseno.

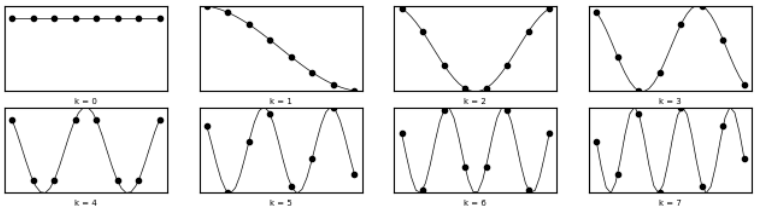


Figura 4.3: Representación de las funciones base de la DCT para $N = 8$.

2-DCT

Como se ha hecho en la 2-DFT, en la Figura 4.4 se han representado las funciones base de la 2-DCT como imágenes.

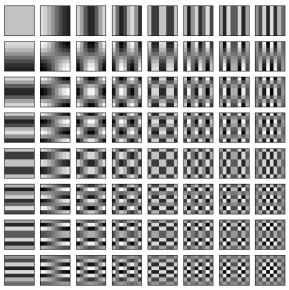


Figura 4.4: Representación de las funciones base de la 2-DCT para $N = 8$.

4.2. Ejemplo de la DCT y la DFT

En esta sección se va representar gráficamente cómo la transformada discreta de Fourier y la transformada discreta del coseno compactan toda la energía de los datos en unos pocos coeficientes. Para ello, se va a usar, como datos de entrada, la imagen de la Figura 4.5, que es una de las imágenes más usadas en el ámbito de la compresión.

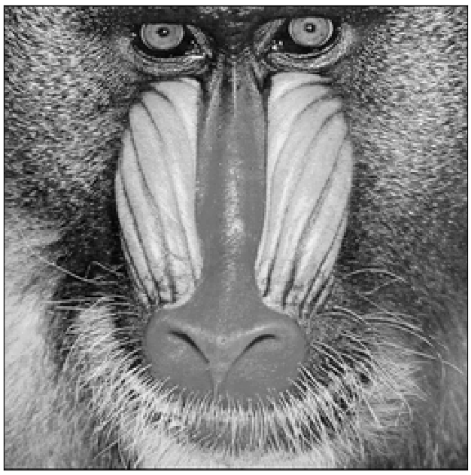


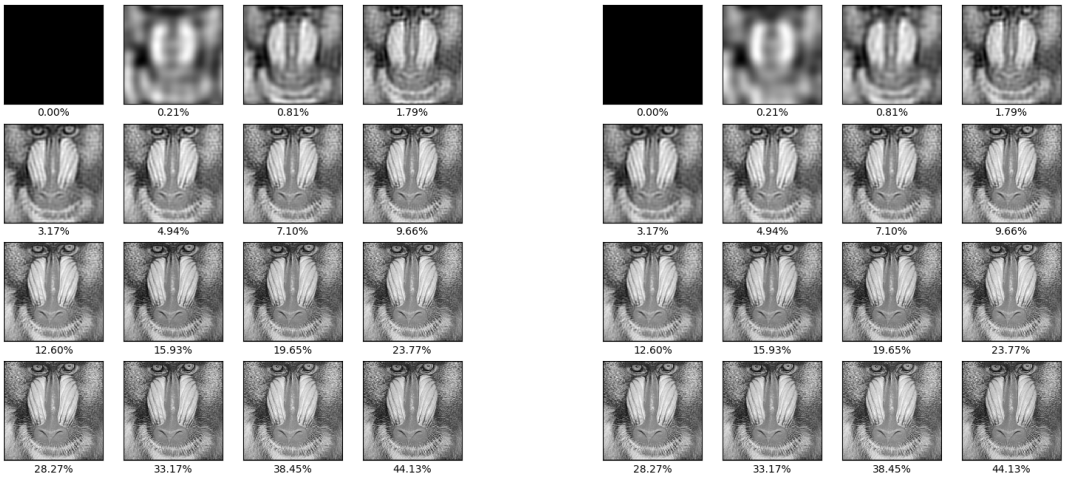
Figura 4.5: Imagen usada en los ejemplos de este documento.

En primer lugar, se va a aplicar, a la imagen, las dos transformaciones por separado: por un lado, la DFT y, por el otro, la DCT. Una vez obtenidas las dos transformaciones, se procederá de manera igual para ambas.

Para cada transformación, se escogerá un porcentaje de los primeros coeficientes y, los otros, se pondrán a cero, es decir, se truncarán los coeficientes restantes. Una vez realizado esto, se aplicará la inversa a la transformación modificada para intentar recuperar la imagen original.

Los resultados de este proceso se muestran en la Figura 4.6. En ella, se observa que los resultados obtenidos usando la DFT (Figura 4.6a) y los resultados obtenidos usando la DCT (Figura 4.6b) son muy parecidos. También queda reflejado que con solo un 10 % de los coeficientes de la transformación se puede obtener la imagen original con un alto grado de fidelidad.

Todo esto se ha implementado en el lenguaje de programación Python y cuyo código se encuentra en el anexo A.5.



(a) DFT (b) DCT

Figura 4.6: Inversa de las transformaciones usando solo un porcentaje de coeficientes.

4.3. Ejemplo de la cuantificación vectorial

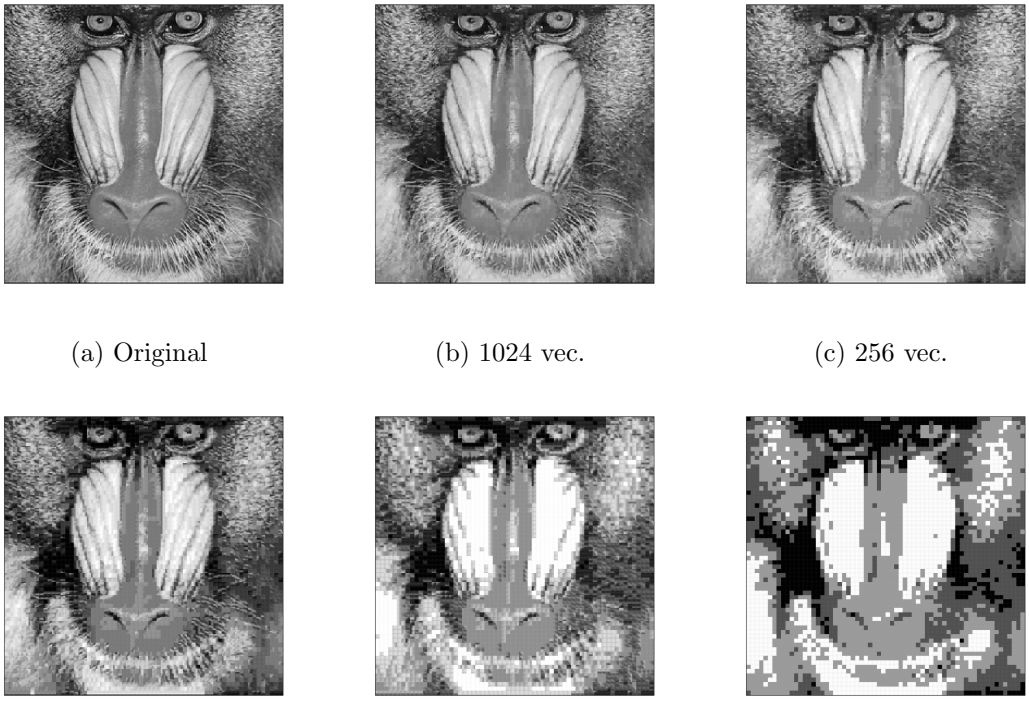
En esta sección se va a analizar los resultados gráficos obtenidos en la implementación de la cuantización vectorial, cuyo código se encuentra en el anexo A.6. Para ello se ha usado, al igual que en el ejemplo anterior, la imagen de la Figura 4.5 para realizar todos los experimentos.

El algoritmo implementado consiste, en primer lugar, en leer la imagen y particionarla en bloques de 4×4 usando el algoritmo implementado en la estancia en prácticas. Cada bloque de estos se representa como un vector.

Después se determina el número de vectores que contendrá el 'codebook' y se aplica a los vectores escogidos el método de clasificación k-means. La implementación usada de este método devuelve los vectores del 'codebook' así como una lista con la clasificación de los vectores creados anteriormente.

Finalmente, se sustituye cada vector por su correspondiente vector del 'codebook' y se aplica la inversa del algoritmo implementado en la estancia en prácticas para poder representar el resultado final.

Un ejemplo de este algoritmo se encuentra en la Figura 4.7. En ella se ha aplicado la cuantización vectorial a la imagen de la Figura 4.5 escogiendo distintos tamaños del 'codebook'.



(a) Original (b) 1024 vec. (c) 256 vec.

(d) 64 vec. (e) 16 vec. (f) 4 vec.

Figura 4.7: Cuantización vectorial en función del tamaño del 'codebook'.

En las Figuras 4.7b y 4.7c, se ha usado un 'codebook' con 1024 y 256 vectores respectivamente. En ambas, la pérdida de detalle casi no se aprecia. Sin embargo, en la Figura 4.7e y, sobretodo, en la Figura 4.7f, se aprecia mucho más la pérdida de información al aplicar la cuantización vectorial. En esta última figura, se ha usado un 'codebook' de solo 4 vectores.

Capítulo 5

Conclusiones

Durante el transcurso de la Estancia en Prácticas y durante el desarrollo del Proyecto de Fin de Grado se ha tratado la compresión de datos desde dos perspectivas: por un lado, en la estancia en prácticas, se ha visto la compresión desde el punto de vista de un informático y, por otro lado, en el desarrollo de la parte teórica del TFG, se ha visto cómo funciona la compresión desde un punto de vista matemático.

Por una parte, en la estancia en prácticas se ha estado trabajando en un compresor de datos desarrollado por Francesc Alted, *Blosc*. En concreto, se ha desarrollado completamente una interfaz para poder usar el compresor desde el lenguaje de programación *Python*. También se ha implementado una transformación de datos para *Blosc* previa a la compresión, la cual mejora la velocidad de descompresión en el caso de que se tenga que descomprimir solo un subconjunto de los datos.

Por otra parte, el desarrollo teórico del TFG se ha dedicado a explicar algunas técnicas matemáticas usadas en la compresión de datos, en concreto en la compresión de imágenes. La elección de este tema se debió a que, como se ha mencionado antes, se ha trabajado durante la estancia en prácticas en la compresión vista desde un punto de vista informático. En primer lugar, se ha explicado qué es la compresión y sus conceptos más básicos. También se ha explicado una de las técnicas de compresión de imágenes que más se usa: la codificación por transformación. Posteriormente, se han definido las transformaciones unitarias tanto para una dimensión como para dos y se han mencionado las propiedades deseables que estas deben cumplir para poder ser usadas en la compresión de los datos. En esta sección también se han definido la transformada discreta de Fourier y la transformada discreta del coseno. Finalmente, en la última sección del desarrollo teórico, se ha visto la cuantización de datos escalar y la cuantización de datos vectorial.

Por último, debido a que este Proyecto de Fin de Grado se enmarca dentro del Grado de Matemática Computacional, se han implementado, usando los conocimientos adquiridos en la parte más informática del grado, algunos ejemplos de lo visto en el desarrollo teórico. En concreto, se han representado gráficamente las funciones base de las dos transformaciones explicadas. También se ha comprobado que tanto la DFT como la DCT tienen un grado elevado de compactación de energía en sus coeficientes, pues con usar solo un 10 % de sus coeficientes, ya se obtiene la imagen original sin una pérdida de información apreciable. Finalmente, también se ha implementado, usando la transformación implementada en las prácticas, un ejemplo de la cuantización vectorial.

En este trabajo, nos hemos centrado en las transformaciones más básicas usadas en la compresión y, por tanto, como trabajo futuro, se podría hacer más hincapié en otras transformaciones que también se aplican a la hora de la compresión de imágenes. Estas podrían ser, por ejemplo, las wavelets, que son usadas por el nuevo estándar de compresión JPEG 2000.

Bibliografía

Clarke, R. J. (1981). Relation between the Karhunen-Loève and cosine transforms. *Communications, Radar and Signal Processing, IEE Proceedings F*, 128(6):359–360.

Gersho, A. and Gray, R. M. (1991). *Vector Quantization and Signal Compression*. Kluwer Academic Publishers, Norwell, MA, USA.

Jain, A. K. (1989). *Fundamentals of Digital Image Processing*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA.

Jayant, N. S. and Noll, P. (1990). *Digital Coding of Waveforms: Principles and Applications to Speech and Video*. Prentice Hall Professional Technical Reference.

Kammler, D. W. (2007). *A first course in Fourier analysis*. Cambridge University Press.

Lim, J. S. (1990). *Two-dimensional Signal and Image Processing*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA.

Linde, Y., Buzo, A., and Gray, R. (1980). An algorithm for vector quantizer design. *IEEE Transactions on Communications*, 28(1):84–95.

Lloyd, S. (1982). Least squares quantization in PCM. *IEEE Transactions on Information Theory*, 28(2):129–137.

Max, J. (1960). Quantizing for minimum distortion. *IRE Transactions on Information Theory*, 6(1):7–12.

Salomon, D. and Motta, G. (2009). *Handbook of Data Compression*. Springer Publishing Company, Incorporated, 5th edition.

Tekalp, A. M. (1995). *Digital Video Processing*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA.

VanderPlas, J. (2016). *Python Data Science Handbook: Essential Tools for Working with Data*. O'Reilly Media, Inc., 1st edition.

Wintz, P. A. (1972). Transform picture coding. *Proceedings of the IEEE*, 60(7):809–820.

Wood, R. (1969). On optimum quantization. *IEEE Transactions on Information Theory*, 15(2):248–252.

Anexo A

Código desarrollado en la estancia en prácticas

A.1. Creación del paquete pycblosc2

A.1.1. Código de pycblosc2

En el siguiente código está implementada la interfaz de *Blosc* para poder ser usado desde *Python*.

```
# Simple CFFI wrapper for the C-Blosc2 library

from pkg_resources import get_distribution, DistributionNotFound
try:
    __version__ = get_distribution(__name__).version
except DistributionNotFound:
    # package is not installed
    from setuptools_scm import get_version
    __version__ = get_version()


from cffi import FFI

ffi = FFI()
ffi.cdef(
    """
```

```
void blosc_init(void);

void blosc_destroy(void);

int blosc_compress(int clevel, int doshuffle, size_t typesize,
    size_t nbytes, const void* src, void* dest, size_t destsize);

int blosc_decompress(const void* src, void* dest, size_t,
    destsize);

int blosc_getitem(const void* src, int start, int nitems, void*
    dest);

int blosc_get_nthreads(void);

int blosc_set_nthreads(int nthreads);

char* blosc_get_compressor(void);

int blosc_set_compressor(const char* compname);

void blosc_set_delta(int dodelta);

int blosc_compcode_to_compname(int compcode, char** compname);

int blosc_compname_to_compcode(const char* compname);

char* blosc_list_compressors(void);

char* blosc_get_version_string(void);

int blosc_get_complib_info(char* compname, char** complib, char
    ** version);

int blosc_free_resources(void);

void blosc_cbuffer_sizes(const void* cbuffer, size_t* nbytes,
    size_t* cbytes, size_t* blocksize);

void blosc_cbuffer_metainfo(const void* cbuffer, size_t*
    typesize, int* flags);
```

```
void blosc_cbuffer_versions(const void* cbuffer, int* version,
    int* versionlz);

char* blosc_cbuffer_complib(const void* cbuffer);

enum {
  BLOSC_MAX_FILTERS = 5,
};

typedef struct blosc2_context_s blosc2_context;

typedef struct {
  int compcode;
  int clevel;
  int use_dict;
  size_t typesize;
  uint32_t nthreads;
  size_t blocksize;
  void* schunk;
  uint8_t filters[BLOSC_MAX_FILTERS];
  uint8_t filters_meta[BLOSC_MAX_FILTERS];
} blosc2_cparams;

typedef struct {
  int32_t nthreads;
  void* schunk;
} blosc2_dparams;

blosc2_context* blosc2_create_cctx(blosc2_cparams cparams);

blosc2_context* blosc2_create_dctx(blosc2_dparams dparams);

void blosc2_free_ctx(blosc2_context* context);

int blosc2_compress_ctx(blosc2_context* context, size_t nbytes,
    void* src, void* dest, size_t destsize);

int blosc2_decompress_ctx(blosc2_context* context, const void*
    src, void* dest, size_t destsize);

int blosc2_getitem_ctx(blosc2_context* context, const void* src,
     int start, int nitems, void* dest);
```

```
typedef struct {
  uint8_t version;
  uint8_t flags1;
  uint8_t flags2;
  uint8_t flags3;
  uint8_t compcode;
  uint8_t clevel;
  uint32_t typesize;
  int32_t blocksize;
  uint32_t chunksize;
  uint8_t filters[BLOSC_MAX_FILTERS];
  uint8_t filters_meta[BLOSC_MAX_FILTERS];
  int64_t nchunks;
  int64_t nbytes;
  int64_t cbytes;
  uint8_t* filters_chunk;
  uint8_t* codec_chunk;
  uint8_t* metadata_chunk;
  uint8_t* userdata_chunk;
  uint8_t** data;
  //uint8_t* ctx;
  blosc2_context* cctx;
  blosc2_context* dctx;
  uint8_t* reserved;
} blosc2_schunk;

blosc2_schunk* blosc2_new_schunk(blosc2_cparams cparams,
    blosc2_dparams dparams);

int blosc2_free_schunk(blosc2_schunk* sheader);

size_t blosc2_append_buffer(blosc2_schunk* sheader, size_t
    nbytes, void* src);

int blosc2_decompress_chunk(blosc2_schunk* sheader, size_t
    nchunk, void* dest, size_t nbytes);

int blosc_get_blocksize(void);

void blosc_set_blocksize(size_t blocksize);
```

```
    void blosc_set_schunk(blosc2_schunk* schunk);
    """
)

C = ffi.dlopen("blosc")


def blosc_init():
    return C.blosc_init()


def blosc_destroy():
    return C.blosc_destroy()


def blosc_compress(clevel, doshuffle, typesize, nbytes, src,dest,
   destsize):
    src = ffi.from_buffer(src)
    dest = ffi.from_buffer(dest)
    return C.blosc_compress(clevel, doshuffle, typesize, nbytes, src
        , dest, destsize)


def blosc_decompress(src, dest, destsize):
    src = ffi.from_buffer(src)
    dest = ffi.from_buffer(dest)
    return C.blosc_decompress(src, dest, destsize)


def blosc_getitem(src, start, nitems, dest):
    src = ffi.from_buffer(src)
    dest = ffi.from_buffer(dest)
    return C.blosc_getitem(src, start, nitems, dest)


def blosc_get_nthreads():
    return C.blosc_get_nthreads()


def blosc_set_nthreads(nthreads):
    return C.blosc_set_nthreads(nthreads)
```

```
def blosc_get_compressor():
    return ffi.string(C.blosc_get_compressor()).decode("utf-8")


def blosc_set_compressor(compname):
    if type(compname) == str:
        compname = ffi.new("char[]", compname.encode("utf-8"))
    else:
        compname = ffi.new("char[]", compname)
    return C.blosc_set_compressor(compname)


def blosc_set_delta(dodelta):
    return C.blosc_set_delta(dodelta)


def blosc_compcode_to_compname(compcode):
    compname = ffi.new("char **")
    return (C.blosc_compcode_to_compname(compcode, compname),
            ffi.string(compname[0]).decode("utf-8"))


def blosc_compname_to_compcode(compname):
    if type(compname) == str:
        compname = ffi.new("char[]", compname.encode("utf-8"))
    else:
        compname = ffi.new("char[]", compname)
    return C.blosc_compname_to_compcode(compname)


def blosc_list_compressors():
    return ffi.string(C.blosc_list_compressors()).decode('utf8')


def blosc_get_version_string():
    return ffi.string(C.blosc_get_version_string()).decode('utf8')


def blosc_get_complib_info(compname):
    if type(compname) == str:
        compname = ffi.new("char[]", compname.encode("utf-8"))
```

```
    else:
        compname = ffi.new("char[]", compname)
    complib = ffi.new("char **")
    version = ffi.new("char **")
    return (C.blosc_get_complib_info(compname, complib, version),
        ffi.string(complib[0]).decode("utf-8"), ffi.string(version
        [0]))


def blosc_free_resources():
    return C.blosc_free_resources()


def blosc_cbuffer_sizes(cbuffer):
    nbytes = ffi.new("size_t *")
    cbytes = ffi.new("size_t *")
    blocksize = ffi.new("size_t *")
    cbuffer = ffi.from_buffer(cbuffer)
    C.blosc_cbuffer_sizes(cbuffer, nbytes, cbytes, blocksize)
    return nbytes[0], cbytes[0], blocksize[0]


def blosc_cbuffer_metainfo(cbuffer):
    typesize = ffi.new("size_t *")
    flags = ffi.new("int *")
    cbuffer = ffi.from_buffer(cbuffer)
    C.blosc_cbuffer_metainfo(cbuffer, typesize, flags)
    return typesize[0], list((0, 1)[flags[0] >> j & 1] for j in
        range(3, -1, -1))


def blosc_cbuffer_versions(cbuffer):
    version = ffi.new("int *")
    versionlz = ffi.new("int *")
    cbuffer = ffi.from_buffer(cbuffer)
    C.blosc_cbuffer_versions(cbuffer, version, versionlz)
    return (version[0], versionlz[0])


def blosc_cbuffer_complib(cbuffer):
    cbuffer = ffi.from_buffer(cbuffer)
```

```
    return ffi.string(C.blosc_cbuffer_complib(cbuffer)).decode('utf8
        ')


######################
# STRUCTS FUNCTIONS #
######################
def blosc2_create_cparams(compcode, clevel, use_dict, typesize,
    nthreads, blocksize, schunk, filters, filters_meta):
    cp = ffi.new("blosc2_cparams*")
    cp.compcode = compcode
    cp.clevel = clevel
    cp.use_dict = use_dict
    cp.typesize = typesize
    cp.nthreads = nthreads
    cp.blocksize = blocksize
    cp.schunk = ffi.NULL if schunk is None else schunk
    cp.filters = filters
    cp.filters_meta = filters_meta
    return cp[0]


def blosc2_create_dparams(nthreads, schunk):
    dp = ffi.new("blosc2_dparams*")
    dp.nthreads = nthreads
    dp.schunk = ffi.NULL if schunk is None else schunk
    return dp[0]


def blosc2_create_cctx(cparams):
    return C.blosc2_create_cctx(cparams)


def blosc2_create_dctx(dparams):
    return C.blosc2_create_dctx(dparams)


def blosc2_free_ctx(context):
    return C.blosc2_free_ctx(context)


def blosc2_compress_ctx(context, nbytes, src, dest, destsize):
```

```
    src = ffi.from_buffer(src)
    dest = ffi.from_buffer(dest)
    return C.blosc2_compress_ctx(context, nbytes, src, dest,
        destsize)


def blosc2_decompress_ctx(context, src, dest, destsize):
    src = ffi.from_buffer(src)
    dest = ffi.from_buffer(dest)
    return C.blosc2_decompress_ctx(context, src, dest, destsize)


def blosc2_getitem_ctx(context, src, start, nitems, dest):
    src = ffi.from_buffer(src)
    dest = ffi.from_buffer(dest)
    return C.blosc2_getitem_ctx(context, src, start, nitems, dest)


def blosc2_new_schunk(cparams, dparams):
    return C.blosc2_new_schunk(cparams, dparams)


def blosc2_free_schunk(schunk):
    return C.blosc2_free_schunk(schunk)


def blosc2_append_buffer(schunk, nbytes, src):
    src = ffi.from_buffer(src)
    return C.blosc2_append_buffer(schunk, nbytes, src)


def blosc2_decompress_chunk(schunk, nchunk, dest, nbytes):
    dest = ffi.from_buffer(dest)
    return C.blosc2_decompress_chunk(schunk, nchunk, dest, nbytes)


def blosc_get_blocksize():
    return C.blosc_get_blocksize()


def blosc_set_blocksize(blocksize):
    return C.blosc_set_blocksize(blocksize)
```

```
def blosc_set_schunk(schunk):
    return C.blosc_set_schunk(schunk)
```

A.1.2. Tests

En el siguiente código hay una batería de tests para comprobar el correcto funcionamiento de la interfaz mencionada anteriormente.

```
import pycblosc2 as cb2
import numpy as np
import unittest


class TestUM(unittest.TestCase):

    def setUp(self):
        self.arr_1 = np.random.randint(5, size=(1000, 1000), dtype=
            np.int32)
        self.arr_2 = np.random.randint(5, size=(1000, 1000), dtype=
            np.int32)
        self.arr_3 = np.random.randint(5, size=(1000, 1000), dtype=
            np.int32)
        self.arr_aux = np.random.randint(5, size=(100, 100), dtype=
            np.int32)

    def test_compress_decompress(self):
        cb2.blosc_compress(5, 1, 4, 1000000 * 4, self.arr_1, self.
            arr_2, 1000000 * 4)
        cb2.blosc_decompress(self.arr_2, self.arr_3, 1000000 * 4)
        np.testing.assert_array_equal(self.arr_1, self.arr_3)

    def test_compress_getitem(self):
        cb2.blosc_compress(5, 0, 4, 1000000 * 4, self.arr_1, self.
            arr_2, 1000000 * 4)
        cb2.blosc_getitem(self.arr_2, 1000, 10000, self.arr_aux)
        arr_1 = self.arr_1[1:11, :].reshape(100, 100)
        np.testing.assert_array_equal(arr_1, self.arr_aux)

    def test_threads(self):
```

```
        cb2.blosc_set_nthreads(2)
        n = cb2.blosc_get_nthreads()
        self.assertEqual(n, 2)

    def test_set_compressor(self):
        cb2.blosc_set_compressor('lz4hc')
        n = cb2.blosc_get_compressor()
        self.assertEqual(n, 'lz4hc')
        cb2.blosc_set_compressor(b'lz4')
        n = cb2.blosc_get_compressor()
        self.assertEqual(n, 'lz4')

    def test_blosc_options(self):
        cb2.blosc_compress(5, 0, 4, 1000000 * 4, self.arr_1, self.
            arr_2, 1000000 * 4)
        _, flag = cb2.blosc_cbuffer_metainfo(self.arr_2)
        self.assertEqual(flag, [0, 0, 0, 0])
        cb2.blosc_set_delta(1)
        cb2.blosc_compress(5, 1, 4, 1000000 * 4, self.arr_1, self.
            arr_2, 1000000 * 4)
        _, flag = cb2.blosc_cbuffer_metainfo(self.arr_2)
        self.assertEqual(flag, [1, 0, 0, 1])
        cb2.blosc_set_delta(0)
        cb2.blosc_compress(5, 2, 4, 1000000 * 4, self.arr_1, self.
            arr_2, 1000000 * 4)
        _, flag = cb2.blosc_cbuffer_metainfo(self.arr_2)
        self.assertEqual(flag, [0, 1, 0, 0])

    def test_context(self):
        cparams = cb2.blosc2_create_cparams(compcode=1, clevel=5,
            use_dict=0, typesize=4, nthreads=1, blocksize=0, schunk=
            None, filters=[0, 0, 0, 0, 1], filters_meta=[0, 0, 0, 0,
            0])
        cctx = cb2.blosc2_create_cctx(cparams)
        dparams = cb2.blosc2_create_dparams(nthreads=1, schunk=None)
        dctx = cb2.blosc2_create_dctx(dparams)
        cb2.blosc2_compress_ctx(cctx, 1000000 * 4, self.arr_1, self.
            arr_2, 1000000 * 4)
        cb2.blosc2_decompress_ctx(dctx, self.arr_2, self.arr_3,
            1000000 * 4)
        np.testing.assert_array_equal(self.arr_1, self.arr_3)
```

```
    cb2.blosc2_getitem_ctx(dctx, self.arr_2, 1000, 10000, self.
        arr_aux)
    arr_1 = self.arr_1[1:11, :].reshape(100, 100)
    np.testing.assert_array_equal(arr_1, self.arr_aux)

def test_schunk(self):
    cparams = cb2.blosc2_create_cparams(compcode=1, clevel=5,
        use_dict=0, typesize=4, nthreads=1, blocksize=0, schunk=
        None, filters=[0, 0, 0, 0, 1], filters_meta=[0, 0, 0, 0,
        0])
    cparams.typesize = 4
    cparams.filters[0] = 3
    cparams.clevel = 9
    dparams = cb2.blosc2_create_dparams(nthreads=1, schunk=None)
    schunk = cb2.blosc2_new_schunk(cparams, dparams)
    nchunks = cb2.blosc2_append_buffer(schunk, 4 * 1000000, self
        .arr_1)
    cb2.blosc2_decompress_chunk(schunk, nchunks - 1, self.arr_3,
         4 * 1000000)
    np.testing.assert_array_equal(self.arr_1, self.arr_3)

def test_blocksize(self):
    n = cb2.blosc_get_blocksize()
    self.assertEqual(n, 0)
    cb2.blosc_set_blocksize(32 * 1024)
    m = cb2.blosc_get_blocksize()
    self.assertEqual(m, 32 * 1024)

def test_string_decode(self):
    cb2.blosc_compress(5, 1, 4, 1000000 * 4, self.arr_1, self.
        arr_2, 1000000 * 4)
    cb2.blosc_decompress(self.arr_2, self.arr_3, 1000000 * 4)

    compcode1 = cb2.blosc_compname_to_compcode(b'lizard')
    compcode2 = cb2.blosc_compname_to_compcode('lizard')
    self.assertEqual(compcode1, compcode2)
    compname = cb2.blosc_compcode_to_compname(1)[1]
    self.assertEqual(compname, 'lz4')
    complib1 = cb2.blosc_get_complib_info(b'lz4')
    complib2 = cb2.blosc_get_complib_info('lz4')
    self.assertEqual(complib1, complib2)
```

```
if __name__ == '__main__':
    unittest.main()
```

A.2. Análisis del filtro 'shuffle' iterativo

A.2.1. Test de ejemplo

En este notebook se va a analizar cómo afecta aplicar el filtro 'shuffle' de forma iterativa a la tasa de compresión de un conjunto de datos real. Primero, se van a leer estos datos y establecer los parámetros necesarios de Blosc. Después, para cada códec se va a calcular la tasa de compresión en cada iteración del filtro. Finalmente, se va a mostrar una gráfica con los resultados obtenidos.

Paquetes necesarios

```
In [1]: import pandas as pd
        import tables as tb
        import matplotlib.pyplot as plt
        import pycblosc2 as cb2
        import numpy as np
        from math import log
        import time as t
```

Definición de los parámetros de Blosc

```
In [4]: KB = 1024
        MB = 1024 * KB
        GB = 1024 * MB

        BLOSC_MAX_FILTERS = 5
        BLOSC_BLOCKSIZE = 16 * KB

        cparams = cb2.blosc2_create_cparams(compcode=1,
                                            clevel=5,
                                            use_dict=0,
```

```
                                                typesize=4,
                                                nthreads=4,
                                                blocksize=BLOSC_BLOCKSIZE,
                                                schunk=None,
                                                filters=[0, 0, 0, 0, 1],
                                                filters_meta=[0, 0, 0, 0, 2])

        dparams = cb2.blosc2_create_dparams(nthreads=1, schunk=None)
```

Definición de los datos

```
In [2]: def file_reader(filename, dataset):
            with tb.open_file(filename) as f:
                child = f.root
                for element in dataset.split('/'):
                    child = child.__getattr__(element)
                return child[:]


        # Lectura del dataset

        PATH = '/home/aleix/datasets/'

        filename = 'WRF_India.h5'

        dataset = 'U'

        data = file_reader(PATH + filename, dataset)

        data = np.resize(data, (2**9, 2**5, 2**4, 2**7))

        isize = data.size * data.dtype.itemsize
```

Cálculo del orden de la permutación

```
In [5]: TAM_DATA = data.dtype.itemsize

        DATA_SIZE = BLOSC_BLOCKSIZE/TAM_DATA
```

```
        if  log(DATA_SIZE, 2) % log(TAM_DATA, 2) == 0:
            orden = int(log(DATA_SIZE, TAM_DATA) + 1)
        else:
            orden = int(log(DATA_SIZE, 2) + log(TAM_DATA, 2))
```

Cálculo de los ratios de compresión

```
In [8]: fig = plt.figure(figsize=(16, 7))

        for code in [0, 1, 2, 4, 5, 6]:

            cparams.compcode = code

            n = []
            ratio = []
            speed = []

            cparams.filters[BLOSC_MAX_FILTERS - 1] = 0

            schunk = cb2.blosc2_new_schunk(cparams, dparams)
            nchunks = cb2.blosc2_append_buffer(schunk, isize, data)
            nbytes = schunk.nbytes
            cbytes = schunk.cbytes

            cb2.blosc2_free_schunk(schunk)

            r = (1. * nbytes) / cbytes
            n.append(0)
            ratio.append(r)

            cparams.filters[BLOSC_MAX_FILTERS - 1] = 1

            for i_iter in range(orden):

                cparams.filters_meta[BLOSC_MAX_FILTERS - 1] = i_iter

                schunk = cb2.blosc2_new_schunk(cparams, dparams)
                nchunks = cb2.blosc2_append_buffer(schunk, isize, data)
                nbytes = schunk.nbytes
                cbytes = schunk.cbytes
```

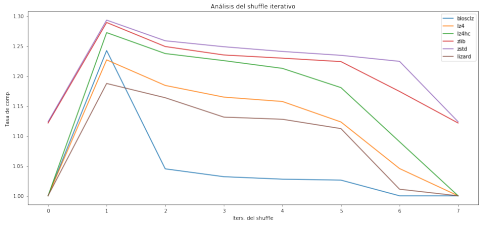
```
            cb2.blosc2_free_schunk(schunk)

            r = (1. * nbytes) / cbytes
            n.append(i_iter + 1)
            ratio.append(r)

        plt.plot(n, ratio, label=cb2.blosc_compcode_to_compname(code)[1])

    plt.ylabel("Tasa de comp.")
    plt.xlabel("Iters. del shuffle")
    plt.title("Análisis del shuffle iterativo")
    plt.legend(loc = "upper right")

    plt.show()
```



A.3. Particionamiento de conjuntos de datos con Blosc

A.3.1. Transformación en *C*

En el siguiente código se encuentra implementada la transformación que particiona los datos.

```
'''
Creates CFFI library to implement data transformation and
    compression algorithms.


calculate_j(k, dim, s, sb)

    Calculate the value of j and returns it.

    Parameters
    ----------
```

k: int
dim: int[]
s: int[]
sb: int[]

Returns

j: int

tData(src, dest, typesize, shape, pad_shape, sub_shape, size, dimension, inverse)

Resize data of needed dimensions and transform it.

Parameters

*src: char**
Location of data to transform pointer.
*dest: char**
Location of data transformed pointer.
typesize: int
Data element size.
shape: int[]
Data shape.
pad_shape: int[]
Data transformed shape.
sub_shape: int[]
Data partition shape.
dimension: int
Data dimension.
inverse: int

tData_simple(src, dest, typesize, sub_shape, shape, dimension, inverse)

Calculate the data transformation in case that the algorithm needed is simple.

Parameters

src: char*
    Location of data to transform pointer.
dest: char*
    Location of data transformed pointer.
typesize: int
    Data element size.
sub_shape: int[]
    Data partition shape.
shape: int[]
    Data shape.
dimension: int
    Data dimension.
inverse: int

padData(src, dest, typesize, shape, pad_shape, dimension)

Resize orignial data padding it with 0 to obtain expandend data
    wich dimension is multiple
    of partition dimension.

Parameters
---

src: char*
    Location of data to transform pointer.
dest: char*
    Location of data transformed pointer.
typesize: int
    Data element size.
shape: int[]
    Original data shape.
shape: int[]
    Expanded data shape.
dimension: int
    Data dimension.

decompress_trans(comp, dest, trans_shape, part_shape, dimensions,
  sub_trans, dimension, b_size,
          typesize)

Decompress partitioned data decompressing only the desired data.

```
    Parameters
    ----------
        comp: char*
            Location of compressed data pointer.
        dest: char*
            Location of data transformed pointer.
        trans_shape: int[]
            Partitioned data shape.
        part_shape: int[]
            A data partition shape.
        dimensions: int[]
            Defines data desired subset.
        sub_trans: int[]
            Desired data shape.
        dimension: int
            Data dimension.
        b_size: int
            Data partition size.
        typesize: int
            Data element size.
'''

from cffi import FFI

ffibuilder = FFI()

ffibuilder.set_source("tData",
    '''
    #include <stdio.h>
    #include <stdint.h>
    #include <blosc.h>
    #include <time.h>

    int calculate_j(int k, int dim[], int s[], int sb[]) {

    int j = dim[0]*((k)%sb[0]
    +
    k/(sb[0]*sb[1]*sb[2]*sb[3]*sb[4]*sb[5]*sb[6]*sb[7])%(s[0]/sb[0])
        *sb[0])

    +
```

```
dim[1]*(k/(sb[0])%sb[1]*s[0]
+
k/(s[0]*sb[1]*sb[2]*sb[3]*sb[4]*sb[5]*sb[6]*sb[7])%(s[1]/sb[1])*
    s[0]*sb[1])

+

dim[2]*(k/(sb[0]*sb[1])%sb[2]*s[1]*s[0]
+
k/(s[0]*s[1]*sb[2]*sb[3]*sb[4]*sb[5]*sb[6]*sb[7])%(s[2]/sb[2])*s
    [0]*s[1]*sb[2])

+

dim[3]*(k/(sb[0]*sb[1]*sb[2])%sb[3]*s[0]*s[1]*s[2]
+
k/(s[0]*s[1]*s[2]*sb[3]*sb[4]*sb[5]*sb[6]*sb[7])%(s[3]/sb[3])*s
    [0]*s[1]*s[2]*sb[3])

+

dim[4]*(k/(sb[0]*sb[1]*sb[2]*sb[3])%sb[4]*s[0]*s[1]*s[2]*s[3]
+
k/(s[0]*s[1]*s[2]*s[3]*sb[4]*sb[5]*sb[6]*sb[7])%(s[4]/sb[4])*s
    [0]*s[1]*s[2]*s[3]*sb[4])

+

dim[5]*(k/(sb[0]*sb[1]*sb[2]*sb[3]*sb[4])%sb[5]*s[0]*s[1]*s[2]*s
    [3]*s[4]
+
k/(s[0]*s[1]*s[2]*s[3]*s[4]*sb[5]*sb[6]*sb[7])%(s[5]/sb[5])*s
    [0]*s[1]*s[2]*s[3]*s[4]*sb[5])

+

dim[6]*(k/(sb[0]*sb[1]*sb[2]*sb[3]*sb[4]*sb[5])%sb[6]*s[0]*s[1]*
    s[2]*s[3]*s[4]*s[5]
+
k/(s[0]*s[1]*s[2]*s[3]*s[4]*s[5]*sb[6]*sb[7])%(s[6]/sb[6])*s[0]*
    s[1]*s[2]*s[3]*s[4]*s[5]*sb[6])
```

```
+

dim[7]*(k/(sb[0]*sb[1]*sb[2]*sb[3]*sb[4]*sb[5]*sb[6]) %sb[7]*s
    [0]*s[1]*s[2]*s[3]*s[4]*s[5]*s[6]
+
k/(s[0]*s[1]*s[2]*s[3]*s[4]*s[5]*s[6]*sb[7]) %(s[7]/sb[7])*s[0]*s
    [1]*s[2]*s[3]*s[4]*s[5]*s[6]*sb[7]);

return j;
}

void padData(char* src, char* dest, int typesize, int shape[],
    int pad_shape[], int dimension) {

int max_dim = 8;
int dim = dimension;

int s[max_dim], ps[max_dim];

for (int i = 0; i < max_dim; i++) {
if (i < dim) {
s[max_dim + i - dim] = shape[i];
ps[max_dim + i - dim] = pad_shape[i];
} else {
s[max_dim - i - 1] = 1;
ps[max_dim - i - 1] = 1;
}
}

int k, k2;

for (int a = 0; a < s[0]; a++) {
for (int b = 0; b < s[1]; b++) {
for (int c = 0; c < s[2]; c++) {
for (int d = 0; d < s[3]; d++) {
for (int e = 0; e < s[4]; e++) {
for (int f = 0; f < s[5]; f++) {
for (int g = 0; g < s[6]; g++) {

k = g*s[7]
+ f*s[7]*s[6]
```

```
+ e*s[7]*s[6]*s[5]
+ d*s[7]*s[6]*s[5]*s[4]
+ c*s[7]*s[6]*s[5]*s[4]*s[3]
+ b*s[7]*s[6]*s[5]*s[4]*s[3]*s[2]
+ a*s[7]*s[6]*s[5]*s[4]*s[3]*s[2]*s[1];

k2 = g*ps[7]
+ f*ps[7]*ps[6]
+ e*ps[7]*ps[6]*ps[5]
+ d*ps[7]*ps[6]*ps[5]*ps[4]
+ c*ps[7]*ps[6]*ps[5]*ps[4]*ps[3]
+ b*ps[7]*ps[6]*ps[5]*ps[4]*ps[3]*ps[2]
+ a*ps[7]*ps[6]*ps[5]*ps[4]*ps[3]*ps[2]*ps[1];

memcpy(&dest[k2 * typesize], &src[k *  typesize], s[7] *
    typesize);
}
}
}
}
}
}
}
}

void tData_simple(char* src, char* dest, int typesize, int
    sub_shape[], int shape[], int dimension, int inverse) {

int MAX_DIM = 8;
int DIM = dimension;

int s[MAX_DIM], sb[MAX_DIM];

for (int i = 0; i < MAX_DIM; i++) {
if (i < DIM) {
s[MAX_DIM + i - DIM] = shape[i];
sb[MAX_DIM + i - DIM] = sub_shape[i];
} else {
s[MAX_DIM - i - 1] = 1;
sb[MAX_DIM - i - 1] = 1;
}
}
```

```
int dim[MAX_DIM], shp[MAX_DIM], sub[MAX_DIM];

for (int i = 0; i < MAX_DIM; i++) {
if (i < DIM) {
dim[i] = 1;
shp[DIM - i - 1] = shape[i];
sub[DIM - i - 1] = sub_shape[i];
} else {
dim[i] = 0;
shp[i] = 1;
sub[i] = 1;
}
}

int k, j;

if (inverse == 0) {

for (int a = 0; a < s[0]; a++) {
for (int b = 0; b < s[1]; b++) {
for (int c = 0; c < s[2]; c++) {
for (int d = 0; d < s[3]; d++) {
for (int e = 0; e < s[4]; e++) {
for (int f = 0; f < s[5]; f++) {
for (int g = 0; g < s[6]; g++) {
for (int h = 0; h < s[7]; h+=sb[7]) {

k = h
+ g*s[7]
+ f*s[7]*s[6]
+ e*s[7]*s[6]*s[5]
+ d*s[7]*s[6]*s[5]*s[4]
+ c*s[7]*s[6]*s[5]*s[4]*s[3]
+ b*s[7]*s[6]*s[5]*s[4]*s[3]*s[2]
+ a*s[7]*s[6]*s[5]*s[4]*s[3]*s[2]*s[1];

j = calculate_j(k, dim, shp, sub);

memcpy(&dest[k * typesize], &src[j * typesize], typesize * sb
    [7]);
```

```
}
}
}
}
}
}
}
}
}

else {

for (int a = 0; a < s[0]; a++) {
for (int b = 0; b < s[1]; b++) {
for (int c = 0; c < s[2]; c++) {
for (int d = 0; d < s[3]; d++) {
for (int e = 0; e < s[4]; e++) {
for (int f = 0; f < s[5]; f++) {
for (int g = 0; g < s[6]; g++) {
for (int h = 0; h < s[7]; h+=sb[7]) {

k = h
+ g*s[7]
+ f*s[7]*s[6]
+ e*s[7]*s[6]*s[5]
+ d*s[7]*s[6]*s[5]*s[4]
+ c*s[7]*s[6]*s[5]*s[4]*s[3]
+ b*s[7]*s[6]*s[5]*s[4]*s[3]*s[2]
+ a*s[7]*s[6]*s[5]*s[4]*s[3]*s[2]*s[1];

j = calculate_j(k, dim, s, sb);

memcpy(&dest[j * typesize], &src[k * typesize], typesize * sb
    [7]);

}
}
}
}
}
}
}
}
```

```
}
}
}

void tData(char* src, char* dest, int typesize, int shape[], int
    pad_shape[], int sub_shape[], int size,
int dimension, int inverse) {

char* src_aux = (char *) calloc (size, typesize);

padData(src, src_aux, typesize, shape, pad_shape, dimension);

tData_simple(src_aux, dest, typesize, sub_shape, pad_shape,
   dimension, inverse);

}

void decompress_trans(char* comp, char* dest2, int shape[], int
   trans_shape[], int part_shape[], int final_s[],
int dimensions[], int dimension, int b_size, int typesize){

int MAX_DIM = 8;
int DIM = dimension;

// Calculate dimensions data

int subpl[MAX_DIM], ts[MAX_DIM], ps[MAX_DIM], dim[MAX_DIM], sd[
   MAX_DIM], fs[MAX_DIM], s[MAX_DIM];

for (int i = 0; i < MAX_DIM; i++) {
if (i < DIM) {
s[MAX_DIM + i - DIM] = shape[i];
ts[MAX_DIM + i - DIM] = trans_shape[i];
ps[MAX_DIM + i - DIM] = part_shape[i];
dim[MAX_DIM + i - DIM] = dimensions[i];
fs[MAX_DIM + i - DIM] = final_s[i];
sd[MAX_DIM + i - DIM] = trans_shape[i]/part_shape[i];
} else {
s[MAX_DIM - i - 1] = 1;
ts[MAX_DIM - i - 1] = 1;
ps[MAX_DIM - i - 1] = 1;
dim[MAX_DIM - i - 1] = -1;
```

```
sd[MAX_DIM - i - 1] = 1;
fs[MAX_DIM - i - 1] = 1;
}
}

int oi_s[MAX_DIM], oi_f[MAX_DIM];

for (int i = 0; i < MAX_DIM; i++) {
if (dim[i] != -1) {
subpl[i] = ps[i];
oi_s[i] = dim[i];
oi_f[i] = dim[i] + 1;
} else {
subpl[i] = ts[i];
oi_s[i] = 0;
oi_f[i] = ts[i];
}
}

int subpl_size = 1;

for (int i = 0; i < MAX_DIM; i++) {
subpl_size *= subpl[i];
}

// Malloc buffers

char *aux = malloc(b_size * typesize);
char *dest = malloc(subpl_size * typesize);

// Calculate index to decompress

int k, n;
int h2, g2, f2, e2, d2, c2, b2, a2;
int cont;

for (int a = oi_s[0]/ps[0]*ps[0]; a < oi_f[0]; a += ps[0]) {
for (int b = oi_s[1]/ps[1]*ps[1]; b < oi_f[1]; b += ps[1]) {
for (int c = oi_s[2]/ps[2]*ps[2]; c < oi_f[2]; c += ps[2]) {
for (int d = oi_s[3]/ps[3]*ps[3]; d < oi_f[3]; d += ps[3]) {
for (int e = oi_s[4]/ps[4]*ps[4]; e < oi_f[4]; e += ps[4]) {
for (int f = oi_s[5]/ps[5]*ps[5]; f < oi_f[5]; f += ps[5]) {
```

```
for (int g = oi_s[6]/ps[6]*ps[6]; g < oi_f[6]; g += ps[6]) {
for (int h = oi_s[7]/ps[7]*ps[7]; h < oi_f[7]; h += ps[7]) {

k = h
+ g*ts[7]
+ f*ts[7]*ts[6]
+ e*ts[7]*ts[6]*ts[5]
+ d*ts[7]*ts[6]*ts[5]*ts[4]
+ c*ts[7]*ts[6]*ts[5]*ts[4]*ts[3]
+ b*ts[7]*ts[6]*ts[5]*ts[4]*ts[3]*ts[2]
+ a*ts[7]*ts[6]*ts[5]*ts[4]*ts[3]*ts[2]*ts[1];

n = a/ps[0]*sd[1]*sd[2]*sd[3]*sd[4]*sd[5]*sd[6]*sd[7]
+ b/ps[1]*sd[2]*sd[3]*sd[4]*sd[5]*sd[6]*sd[7]
+ c/ps[2]*sd[3]*sd[4]*sd[5]*sd[6]*sd[7]
+ d/ps[3]*sd[4]*sd[5]*sd[6]*sd[7]
+ e/ps[4]*sd[5]*sd[6]*sd[7]
+ f/ps[5]*sd[6]*sd[7]
+ g/ps[6]*sd[7]
+ h/ps[7];

blosc_getitem(comp, n * b_size, b_size, aux);

h2 = k % ts[7] % subpl[7];
g2 = k / (ts[7]) % subpl[6];
f2 = k / (ts[7]*ts[6]) % subpl[5];
e2 = k / (ts[7]*ts[6]*ts[5]) % subpl[4];
d2 = k / (ts[7]*ts[6]*ts[5]*ts[4]) % subpl[3];
c2 = k / (ts[7]*ts[6]*ts[5]*ts[4]*ts[3]) % subpl[2];
b2 = k / (ts[7]*ts[6]*ts[5]*ts[4]*ts[3]*ts[2]) % subpl[1];
a2 = k / (ts[7]*ts[6]*ts[5]*ts[4]*ts[3]*ts[2]*ts[1]) % subpl[0];

// Copy block to final data

cont = 0;

for (int ra = a2; ra < a2 + ps[0]; ra++) {
for (int rb = b2; rb < b2 + ps[1]; rb++) {
for (int rc = c2; rc < c2 + ps[2]; rc++) {
for (int rd = d2; rd < d2 + ps[3]; rd++) {
    for (int re = e2; re < e2 + ps[4]; re++) {
        for (int rf = f2; rf < f2 + ps[5]; rf++) {
```

```
                for (int rg = g2; rg < g2 + ps[6]; rg++) {

                k = h2
                    + rg * subpl[7]
                    + rf * subpl[7]*subpl[6]
                    + re * subpl[7]*subpl[6]*subpl[5]
                    + rd * subpl[7]*subpl[6]*subpl[5]*subpl[4]
                    + rc * subpl[7]*subpl[6]*subpl[5]*subpl[4]*subpl
                        [3]
                    + rb * subpl[7]*subpl[6]*subpl[5]*subpl[4]*subpl
                        [3]*subpl[2]
                    + ra * subpl[7]*subpl[6]*subpl[5]*subpl[4]*subpl
                        [3]*subpl[2]*subpl[1];

                    memcpy(&dest[k * typesize], &aux[cont * typesize
                        ], ps[7] * typesize);
                    cont += ps[7];
                }
            }
        }
}
}
}
}

}
}
}
}
}
}
}
}

// Reduce final data

int ini[MAX_DIM], fin[MAX_DIM];

for (int i = 0; i < MAX_DIM - 1; i += 1) {
if (dim[i] != -1) {
ini[i] = dim[i] % ps[i];
fin[i] = dim[i] % ps[i] + 1;
```

```
}
else {
ini[i] = 0;
fin[i] = s[i];
}
}

if (dim[MAX_DIM - 1] != -1) {
ini[MAX_DIM - 1] = dim[MAX_DIM - 1] % ps[MAX_DIM - 1];
fin[MAX_DIM - 1] = dim[MAX_DIM - 1] % ps[MAX_DIM - 1] + 1 ;
} else {
ini[MAX_DIM - 1] = 0;
fin[MAX_DIM - 1] = 1;
}

cont = 0;

for (int a = ini[0]; a < fin[0]; a += 1) {
for (int b = ini[1]; b < fin[1]; b += 1) {
for (int c = ini[2]; c < fin[2]; c += 1) {
for (int d = ini[3]; d < fin[3]; d += 1) {
for (int e = ini[4]; e < fin[4]; e += 1) {
for (int f = ini[5]; f < fin[5]; f += 1) {
for (int g = ini[6]; g < fin[6]; g += 1) {
for (int h = ini[7]; h < fin[7]; h += 1) {

k = h
+ g * subpl[7]
+ f * subpl[7]* subpl[6]
+ e * subpl[7]* subpl[6]* subpl[5]
+ d * subpl[7]* subpl[6]* subpl[5]* subpl[4]
+ c * subpl[7]* subpl[6]* subpl[5]* subpl[4]* subpl[3]
+ b * subpl[7]* subpl[6]* subpl[5]* subpl[4]* subpl[3]* subpl[2]
+ a * subpl[7]* subpl[6]* subpl[5]* subpl[4]* subpl[3]* subpl[2]*
    subpl[1];

memcpy(&dest2[cont * typesize], &dest[k * typesize], fs[7] *
    typesize);
cont += fs[7];

}
}
```

```
    }
    }
    }
    }
    }
    }

    free(aux);
    free(dest);
    }

    ''', libraries=['blosc'])

ffibuilder.cdef(
    '''
    void tData(char* src, char* dest, int typesize, int shape[], int
        pad_shape[], int sub_shape[], int size,
        int dimension, int inverse);

    void tData_simple(char* src, char* dest, int typesize, int
       sub_shape[], int shape[], int dimension, int inverse);

    void padData(char* src, char* dest, int typesize, int shape[],
       int pad_shape[], int dimension);


    int calculate_j(int k, int dim[], int s[], int sb[]);

    void decompress_trans(char* comp, char* dest2, int shape[], int
       trans_shape[], int part_shape[], int final_s[],
        int dimensions[], int dimension, int b_size, int typesize);
    '''
)

if __name__ == "__main__":
    ffibuilder.compile(verbose=True)
```

A.3.2. Tests

En el siguiente código se encuentra implementada una colección de tests para verificar que la transformación implementada anteriormente funcione.

```
"""
Test para comprobar que el algoritmo implementado de descompresion
    funciona
"""

import transformData as td
import numpy as np
import pytest


@pytest.fixture(scope="module",
                params=[
                        ([16, 16, 16], [4, 4, 4], [5, -1, 12, -1,
                            -1, -1, -1, -1]),
                        ([128, 128, 128], [8, 8, 8], [-1, 127, -1,
                            -1, -1, -1, -1, -1]),
                        ([64, 64, 64, 64], [8, 8, 8, 8], [23, -1,
                            -1, 56, -1, -1, -1, -1]),
                        ([123, 124, 158], [32, 16, 32], [13, -1,
                            119, -1, -1, -1, -1, -1]),
                        ([4, 4, 4, 4], [2, 2, 4, 4], [-1, 3, 3, -1,
                            -1, -1, -1, -1]),
                        ([8, 8, 8, 8, 8, 8, 8, 8], [2, 2, 4, 4, 2,
                            4, 4, 2], [-1, 4, 6, -1, 5, -1, -1, 0]),
                        ([128, 135, 70, 200], [8, 16, 8, 32], [-1,
                            45, 67, 61, -1, -1, -1, -1]),
                        ([234, 54, 234], [16, 8, 16], [13, -1, 119,
                            -1, -1, -1, -1, -1]),
                        ([6, 8, 4, 9, 3, 5, 7, 6], [4, 2, 2, 8, 2,
                            4, 4, 4], [-1, 4, 2, 5, -1, 2, 3, 0])
                       ])
def shapes(request):
    shape, part_shape, index = request.param
    yield (shape, part_shape, index)
```

```
#
```

```
def test_algoritm(shapes):
    shape, part_shape, index = shapes

    # Create slices

    slices = []

    for i in range(len(shape)):
        if index[i] != -1:
            slices.append(slice(index[i], index[i] + 1))
        else:
            slices.append(slice(0, shape[i]))

    slices = tuple(slices)

    # Create indexes

    a = index[0]
    b = index[1]
    c = index[2]
    d = index[3]
    e = index[4]
    f = index[5]
    g = index[6]
    h = index[7]

    # Test

    size = np.prod(shape)

    src = np.arange(size, dtype=np.int32).reshape(shape)

    dtype = src.dtype

    src_trans = td.tData(src, part_shape)

    trans_shape = src_trans.shape
```

```
    comp = td.compress(src_trans, part_shape)

    dest = td.decompress_trans(comp, shape, trans_shape, part_shape,
        dtype, a, b, c, d, e, f, g, h)

    np.testing.assert_array_equal(src[slices].reshape(dest.shape),
        dest)
```

A.3.3. Interfaz para la transformación

En el siguiente código se encuentra implementada una interfaz que permite usar la transformación desde el lenguaje de programación *Python*.

```
"""
Implements functions to use tData library.
"""

import numpy as np
from tData import ffi, lib
import pycblosc2 as cb2


def tData(src, ps, inverse=False):
    """
    Apply a data transformation based in reorganize data partitions.

    Parameters
    ----------
    src : np.array
        Data to transform.
    ps: int[] or tuple
        Data partition shape.
    inverse: bool, optional

    Returns
    -------
    dest: np.array
        Data transformed.
    d: dict
        Dictionary with partitions indexation.
    """
```

```
    # Obtain src parameters

    typesize = src.dtype.itemsize
    shape = src.shape
    dimension = len(shape)

    # Calculate the extended dataset parameters

    ts = []

    for i in range(len(shape)):
        d = shape[i]
        if shape[i] % ps[i] != 0:
            d = shape[i] + ps[i] - shape[i] % ps[i]
        ts.append(d)

    size = np.prod(ts)

    # Create destination dataset

    dest = np.empty(size, dtype=src.dtype).reshape(ts)

    # Transform datasets to buffers (for use in cffi)

    src_b = ffi.from_buffer(src)
    dest_b = ffi.from_buffer(dest)

    # Execute the transformation

    inv = 1 if inverse else 0

    lib.tData(src_b, dest_b, typesize, shape, ts, ps, size,
       dimension, inv)

    return dest


# Compression/decompression functions


def compress(src, ps):
```

```
    """
    Compress data.

    Parameters
    ----------
    src : np.array
        Data to compress.
    ps: int[] or tuple
        Data partition shape.

    Returns
    -------
    dest : chunk
        Data compressed.
    """

    size = src.size
    itemsize = src.dtype.itemsize
    bsize = size * itemsize

    dest = np.empty(size, dtype=src.dtype)

    cb2.blosc_set_blocksize(np.prod(ps) * itemsize)

    cb2.blosc_compress(9, 1, itemsize, bsize, src, dest, bsize)

    return dest


def decompress(comp, s, dtype):
    """
    Decompress data.

    Parameters
    ----------
    comp : chunk
        Data compressed.
    s: int[] or tuple
        Original data shape.
    itemsize: int
        Data item size.
    dtype: np.type
```

```
        Data type.

    Returns
    -------
    dest : np.array
        Data decompressed.
    """

    size = np.prod(s)
    bsize = size * dtype.itemsize

    dest = np.empty(size, dtype=dtype).reshape(s)

    cb2.blosc_decompress(comp, dest, bsize)

    return dest


def decompress_trans(comp, s, ts, ps, dtype, a=-1, b=-1, c=-1, d=-1,
    e=-1, f=-1, g=-1, h=-1):
    """
    Decompress partitioned data.

    Parameters
    ----------
    comp : chunk
        Data compressed.
    s: int[] or tuple
        Original data shape.
    ts: int[] or tuple
        Partitioned data shape.
    ps: int[] or tuple
        Data partition shape.
    a, b, c, d, e, f, g, h: int, optional
        Defines the subset of data desired

    Returns
    -------
    dest : np.array
     Data decompressed.
    """
```

```
    dimension = len(ps)
    dim = [a, b, c, d, e, f, g, h][:dimension]
    b_size = np.prod(ps)

    # Calculate desired data shape

    fs = [1] * dimension

    for i in range(dimension):
        if dim[i] != -1:
            fs[i] = 1
        else:
            fs[i] = s[i]

    dest = np.empty(np.prod(fs), dtype=dtype).reshape(fs)

    dest_b = ffi.from_buffer(dest)
    comp_b = ffi.from_buffer(comp)

    lib.decompress_trans(comp_b, dest_b, s, ts, ps, fs, dim,
        dimension, b_size,
                         dest.dtype.itemsize)

    return dest
```

A.3.4. Test de ejemplo

En este notebook se va a analizar el rendimiento del algoritmo creado para poder comprimir los datos particionados y, de esta forma, poder descomprimir solo los datos necesarios.

Para ello se va a crear un conjunto de datos y se va a comprimir y descompirmir de la forma tradicional. Es decir, se va a descomprimir todo el conjunto de datos y, posteriormente, se seleccionará el subconjunto de datos deseado.

Por otra parte se va a particionar los datos y, luego, comprimirlos. A la hora de descomprimir, se calculará las particiones que contienen los datos deseados y sólo se descomprimirán estas.

Paquetes necesarios

```
In [1]: import transformData as td
        import matplotlib.pyplot as plt
        import tables as tb
        import numpy as np
        import time as t
```

Definición del conjunto de datos

```
In [2]: def file_reader(filename, dataset):
            with tb.open_file(filename) as f:
                child = f.root
                for element in dataset.split('/'):
                    child = child.__getattr__(element)
                return child[:]
```

Datos sin transformación

El conjunto de datos que se va a usar será de 4 dimensiones con la forma $(744, 39, 30, 251)$. Por tanto habrá $744 \cdot 39 \cdot 30 \cdot 251 = 218490480$ elementos.

```
In [3]: PATH = '/home/aleix/datasets/'

        filename = 'WRF_India.h5'

        dataset = 'U'

        src = file_reader(PATH + filename, dataset)

        SHAPE = src.shape

        SIZE = np.prod(SHAPE)

        ITEMSIZE = src.dtype.itemsize

        DTYPE = src.dtype
```

Datos transformados

El tamaño de las particiones tiene que ser más pequeño que el del conjunto de datos. En este caso, las particiones son de 32 o 16 elementos por cada dimensión lo que resulta que cada partición tendrá $32 \cdot 8 \cdot 8 \cdot 32 = 65536$ elementos.

```
In [4]: PART_SHAPE = [32, 8, 8, 32]

        src_part = td.tData(src, PART_SHAPE, inverse=False)

        TSHAPE = src_part.shape
```

Compresión y descompresión de los datos

Compresión

```
In [5]: # Datos particionados

        start = t.perf_counter()
        dest_trans = td.compress(src_part, PART_SHAPE)
        end = t.perf_counter()

        ct_t = end - start

        # Datos originales

        start = t.perf_counter()
        dest = td.compress(src, PART_SHAPE)
        end = t.perf_counter()

        c_t = end - start
```

Descompresión

Se quiere descomprimir el subconjunto que forman la fila 500 de la primera dimensión, todas las filas de la segunda y tercera dimensión y la fila 200 de la cuarta dimensión. Para ello, con los datos comprimidos particionados, solo habrá que descomprimir los bloques que contengan

parte de este subconjunto. Sin embargo, con los datos comprimidos sin particionar habrá que descomprimir todos los datos y luego seleccionar el subconjunto.

```
In [20]: # Datos particionados

         start = t.perf_counter()
         res_trans = td.decompress_trans(dest_trans, SHAPE,
                                         TSHAPE, PART_SHAPE,
                                         DTYPE, a=500, d=200)
         end = t.perf_counter()

         dt_t = end - start

         # Datos originales

         start = t.perf_counter()
         res = td.decompress(dest, SHAPE, DTYPE)[500, :, :, 200]
                 .reshape(res_trans.shape)
         end = t.perf_counter()

         d_t = end - start
```

Resultados y conclusiones

```
In [21]: try:
             np.testing.assert_array_equal(res_trans, res)
             print("OK. Las dos matrices son iguales.")
             print("Tiempo: {:.4f}".format(d_t/dt_t))

         except Exception:
             print("ERROR. Las dos matrices no coinciden.")
```

```
OK. Las dos matrices son iguales.
Tiempo: 155.0988
```

Se observa que el tiempo de de descompresión es aproximadamente 155 veces más rápido que el de la descompresión tradicional.

A.4. Representación de las funciones base

En el siguiente código se han representado gráficamente las funciones base de la transformada discreta de Fourier y de la transformada discreta del coseno.

```
from math import cos, sin,  sqrt, pi, inf, e
import numpy as np
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D


# Implementacion de la dft

def dft(u, n, N):
    return complex(cos(2*pi*u*n/N), -sin(2*pi*u*n/N))


# Implementacion de la dct

def dct(u, n, N):
    if u == 0:
        return sqrt(1/N)
    else:
        return sqrt(2/N)*cos(pi*(2*n+1)*u/(2*N))


# Creacion de las funciones base en 2 dimensiones

def basis_functions_2d(a, N, S):

    n = np.linspace(0, N - 1, S)

    res = []

    max = 0
    min = 0

    for u1 in range(N):
        for u2 in range(N):
            img = np.empty(S * S).reshape(S, S)

            for i, n1 in enumerate(n):
```

```
                for j, n2 in enumerate(n):
                    c1 = a(u1, n1, N)
                    c2 = a(u2, n2, N)
                    coef = (c1 * c2).imag
                    img[i, j] = coef
                    if max < coef:
                        max = coef
                    elif min > coef:
                        min = coef
            res.append(img)

    return res, max, min


# Creacion de las funciones base en 1 dimension

def basis_functions(a, N, S):

    n = np.linspace(0, N - 1, S)

    res = []

    max = 0
    min = 0

    for u1 in range(N):
        img = np.empty(S).reshape(1, S)

        for i, n1 in enumerate(n):
            coef = a(u1, n1, N).imag
            img[0, i] = coef
            if max < coef:
                max = coef
            elif min > coef:
                min = coef
        res.append(img)

    return res, max, min

# Plot de las funciones base en 2 dimensiones como imagenes

def plot_2d_img(a, N, name):
```

```
    res, max, min = basis_functions_2d(a, N, N)

    fig = plt.figure(figsize=(32, 32))

    for i, img in enumerate(res):
        plt.subplot(N, N, i + 1)
        plt.imshow(img, cmap=plt.cm.gray, vmin=min, vmax=max)
        plt.grid(False)
        plt.xticks([])
        plt.yticks([])

    plt.savefig(name)


# Plot de las funciones base en 2 dimensiones como funciones

def plot(a, N, name):

    res, max, min = basis_functions(a, N, N)
    res2, max2, min2 = basis_functions(a, N, N * 10)

    fig = plt.figure(figsize=(8, 2))

    for i, img in enumerate(res):
        plt.subplot(2, N/2, i + 1).set_ylim(min, max)
        plt.plot(range(N), img[0], 'k.')
        plt.plot(np.linspace(0, N - 1, N*10), res2[i][0], 'k-', lw
            =0.5)
        plt.grid(False)
        plt.xticks([])
        plt.yticks([])
        plt.xlabel('k = ' + str(i), fontsize=5)

    plt.savefig(name)

# Plot de las funciones base en 2 dimensiones como funciones

def plot_2d(a, N, name):

    res, max, min = basis_functions_2d(a, N, N)
    res2, max2, min2 = basis_functions_2d(a, N, N*4)
```

```
    yv, xv = np.meshgrid(np.linspace(0, N-1, N), np.linspace(0, N-1,
        N))
    yv2, xv2 = np.meshgrid(np.linspace(0, N-1, N*4), np.linspace(0,
       N-1, N*4))

    fig = plt.figure(figsize=(32, 32))

    for i, img in enumerate(res):
        ax = fig.add_subplot(N, N, i+1, projection='3d')
        ax.plot(xv.flatten(), yv.flatten(), img.flatten(), c="k",
            marker=".", ls="None")
        ax.plot_surface(xv2, yv2, res2[i], cmap=plt.cm.gray, alpha
            =0.5)
        ax.grid(False)

        ax.set_zlim((min, max))

        ax.set_xticks([])
        ax.set_yticks([])
        ax.set_zticks([])

    plt.savefig(name)


# Resultados

plot(dft, 8, 'dft-i.png')
```

A.5. Ejemplo de la DFT y DCT

En el siguiente código se ha analizado una de las propiedades de las transformadas: la compactación de la energía en unos pocos coeficientes. El experimento se ha realizado tanto para la DFT como para la DCT.

```
from PIL import Image
from scipy import fftpack
from math import sqrt, cos, pi
import numpy as np
import matplotlib.pyplot as plt
```

```
# Transformacion de la imagen a niveles de grises

def black_and_white(img_path, size=(256, 256)):
    img = Image.open(img_path)
    img = img.resize(size, 1)
    img = img.convert('L')
    img = np.array(img, dtype=np.float)
    return img


# Implementacion de la 2-DCT

def C(k):
    if k == 0:
        return 1/sqrt(2)
    else:
        return 1


def dct_2d(img, inverse=False):

    if not inverse:
        return fftpack.dct(fftpack.dct(img.T, norm='ortho').T, norm=
            'ortho')

    else:
        return fftpack.idct(fftpack.idct(img.T, norm='ortho').T,
            norm='ortho')


# Implementacion de la 2-DFT

def dft_2d(img, inverse=False):

    if not inverse:

        return fftpack.rfft(fftpack.rfft(img.T).T)

    else:
        return fftpack.irfft(fftpack.irfft(img.T).T)
```

```
# Se lee la imagen

img = black_and_white('lenna.png')

# Se aplica la transformacion

trans = dft_2d(img)

# Se restaura la imagen usando solo un porcentaje de coeficientes y
    se imprime

fig = plt.figure(figsize=(8, 8))

for k in range(0, 256, 16):

    buf = trans.copy()

    for i in range(k):
        buf[i, (k - i):] = 0
        buf[(k-i):, i] = 0
    buf[k:, k:] = 0

    recov = dft_2d(buf, inverse=True)

    plt.subplot(4, 4, k/16 + 1)
    plt.imshow(recov, cmap=plt.cm.gray)
    plt.grid(False)
    plt.xlabel('{:.2f}%'.format((k*(k+1)/2) * 100 / 256**2))
    plt.xticks([])
    plt.yticks([])

plt.savefig('res-dft.png')
```

A.6. Ejemplo de la cuantización vectorial

En este código se ha implementado la cuantización vectorial sobre una imagen.

```
import numpy as np
```

```
import scipy as sp
import matplotlib.pyplot as plt
import transformData as td

from PIL import Image
from scipy.cluster.vq import kmeans2

def black_and_white(img_path, size=(256, 256)):
    img = Image.open(img_path)
    img = img.resize(size, 1)
    img = img.convert('L')
    img = np.array(img, dtype=np.float)
    return img


# Se lee la imagen

face = black_and_white('lenna.png')
shape = face.shape

# Se crean los bloques de 4x4

n_block = 4
size_block = n_block * n_block

face_trans = td.tData(face, [n_block, n_block]).flatten()

vectors = []

for i in range(0, len(face_trans), size_block):
    v = face_trans[i: i + size_block]
    vectors.append(v)

fig = plt.figure(figsize=(8, 8))

for e, k_groups in enumerate([10000, 1000, 100, 10]):

    # Se aplica el k-means obteniendo un codebook y la clasificacion
        de cada vector

    codebook, labels = kmeans2(vectors, k_groups)
```

```
# Se asigna cada elemento del codebook a su vector

face_res = np.copy(face_trans)

for i in range(0, len(face_trans), size_block):
    face_res[i: i + 16] = codebook[labels[i // size_block]]

# Se juntan los bloques

face_recover = td.tData(face_res.reshape(shape), [n_block,
   n_block], inverse=True)

# Se imprime la imagen cuantizada

# plt.subplot(2, 2, e + 1)
plt.imshow(face_recover, cmap=plt.cm.gray)
plt.grid(False)
# plt.xlabel('Tam. del "codebook": {:d}'.format(k_groups))
plt.xticks([])
plt.yticks([])

plt.savefig('res-vq-{:d}.png'.format(k_groups))
```