UNIVERSITAT JAUME I

Grado en Matemática Computacional

Prácticas Externas y Proyecto de Final de Grado

El teorema de Stone-Weierstrass y su aplicación a las redes neuronales

Autor:
Oscar Guiñon Montolio

Supervisor:
Francesc Alted Abad
Tutores académicos:
Juan José Font Ferrandis
Sergio Macario Vives

Fecha de lectura: __ de _____ de 20__
Curso académico 2019/2020

Resumen

En este documento se detalla tanto la estancia en prácticas en la empresa de Francesc Alted como el desarrollo teórico del teorema de Stone-Weierstrass y su aplicación a las redes neuronales para aproximar funciones continuas.

En la primera parte del escrito se describe la estancia en prácticas. Mi labor en la empresa fue desarrollar una nueva versión de la librería *Caterva*, cuya función es preprocesar los datos antes de que sean tratados por el compresor *Blosc*, particionándolos para que la extracción de una parte de los datos comprimidos sea más rápida. La nueva funcionalidad que añadí a esta librería fue la de crear subparticiones que agilizaran aún más la descompresión.

En la segunda parte, correspondiente al desarrollo teórico del TFG, se da una demostración del teorema de Stone-Weierstrass que utiliza propiedades matemáticas elementales. Además, se usa el teorema para comprobar que algunas redes neuronales aproximan a las funciones continuas.

Palabras clave

Compresión, bloque, teorema de Stone-Weierstrass, red neuronal.

Keywords

Compression, blocking, Stone-Weierstrass, neural network.

Índice general

Capítulo 1

Introducción

1.1. Contexto y motivación del proyecto

En el Grado en Matemática Computacional, las Prácticas Externas y el Proyecto Final de Grado se desarrollan en el último curso.

En lo relativo a las prácticas externas, mi estancia tuvo lugar en la empresa de Francesc Alted, situada en el Grao de Castellón. Uno de los motivos que me llevaron a elegir esta empresa fue su ambiente acogedor, que se debe a su pequeño tamaño. Por otra parte, también me llamó la atención el trabajo que se me pedía realizar, ya que nunca había trabajado en el lenguaje *C* ni en el campo de la compresión de datos. Es por esto que la posibilidad de aprender sobre esto desde cero, con el apoyo y la supervisión de personas con gran experiencia, me parecía realmente atractiva.

La compresión de datos es un caso particular de la codificación que sirve para reducir el tamaño de los datos de forma que sean más sencillos de almacenar. Francesc Alted trabaja en este campo, aunque orientado concretamente a grandes bases de datos (*Big Data*).

En cuanto a mi Proyecto de Final de Grado, este versará acerca del Teorema de Stone-Weierstrass. En esta parte veremos una demostración diferente de las clásicas, que solo usa conceptos elementales de Análisis y Topología.

Además, el teorema se utilizará para comprobar cómo diferentes redes neuronales aproximan a las funciones continuas.

Capítulo 2

Estancia en prácticas

2.1. Introducción

Francesc Alted es un consultor freelance que trabaja tanto para el sector público (Universidad de Oslo, Universidad de Valencia) como para el privado (Kisters, The HDF Group, Engie...). Su principal ámbito de actuación radica en bases de datos *SQL* y *NoSQL* para *Big Data*, centrado en la programación en lenguajes *C* y *Python*.

Además, Francesc es el autor del compresor de datos de altas prestaciones *Blosc* implementado en *C* y que tiene gran utilidad para *Big Data*.

Por otra parte, *Caterva* es una librería de *C* que permite manejar conjuntos de datos comprimidos y de varias dimensiones fácilmente. Esta librería complementa a *Blosc*, permitiendo trabajar con *chunks* (pedazos) multi-dimensionales dentro del conjunto de datos. Esta funcionalidad optimiza la extracción de *slices* (porciones) a partir de contenedores multi-dimensionales.

2.2. Objetivos del proyecto formativo

Durante mi estancia en prácticas, mi tarea ha sido añadir una nueva funcionalidad a la librería *Caterva*. Por ello, aparece el concepto de *block* (bloque), que representa una nueva partición dentro de los *chunks*.

El objetivo de la estancia es desarrollar una nueva versión de los métodos principales de *Caterva* que tenga en cuenta estos *blocks* para que el tiempo de descompresión de *slices* descienda. Para realizar este objetivo será necesario actualizar los diferentes tests de la librería de manera que comprueben correctamente el funcionamiento de estos métodos.

2.3. Conceptos previos y software empleado

En primer lugar, el lenguaje de programación en el que están desarrollados *Blosc* y *Caterva* es *C* y el entorno de programación utilizado durante mi estancia es *CLion* de *JetBrains*. Además, para poder compartir la información online correctamente se usa *Git*.

2.3.1. Git

Git es un software de control de versiones diseñado para mantener actualizado el código desarrollado, registrando sus cambios y coordinando el trabajo que varias personas realizan sobre archivos compartidos.

Durante mi estancia he usado este software para mantener al día los cambios que realizaba sobre mi código y para que el resto de personas con las que trabajaba pudieran verlos al momento.

Con este objetivo en mente me registré en la plataforma *GitHub*, un portal que permite alojar proyectos usando *Git*, de modo que pude crear en mi cuenta un repositorio local de las

librerías *Blosc* y *Caterva* para trabajar con ellas en *CLion*, compartiendo los cambios mediante *Git*.

2.3.2. Blosc

Blosc es el compresor de datos sin pérdida de información que utiliza la librería *Caterva*. Este es un compresor de altas prestaciones y asimismo, está optimizado para datos binarios. Además, usa la técnica del *blocking*. Este sistema se basa en dividir los conjuntos de datos en blocks que sean suficientemente pequeños para caber en la caché L1 de los procesadores modernos y realizar las operaciones de compresión y descompresión allí.

Además, los algoritmos usados en *Blosc* se dedican principalmente a buscar redundancias en los datos, es decir, repeticiones de los mismos patrones de datos.

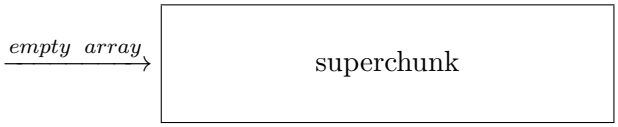
Por otra parte, *Blosc* tiene otras ventajas frente al resto de compresores de altas prestaciones, como la longitud máxima del buffer de destino, la variedad de filtros que se pueden usar antes de la compresión, el pequeño tamaño para el *overhead* de los datos no compresibles, el *multithreading* o el reemplazo de datos mediante *memcpy()*.

2.3.3. Caterva

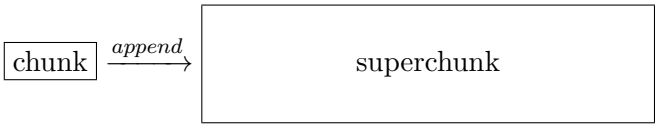
La función principal de esta librería complementaria a *Blosc* es transformar los datos antes de comprimirlos para mejorar los tiempos de descompresión. Con este objetivo se define una nueva estructura: el contenedor de *Caterva*. Este contenedor es multidimensional, admite datos comprimidos y tiene una gran cantidad de parámetros. Entre ellos cabe destacar el *superchunk* de *Blosc* (sc), los diferentes shapes y sizes, el número de dimensiones (ndim) y la cantidad de particiones utilizadas (nparts).

Es importante conocer los métodos principales de la librería, ya que deben ser editados para adaptar la nueva funcionalidad. Estos métodos son:

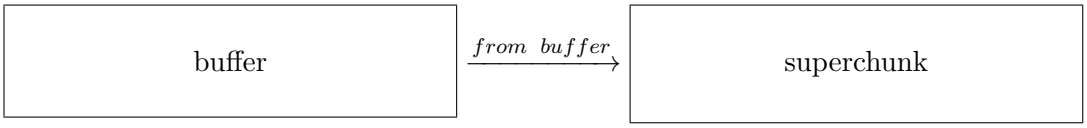
- **caterva_empty_array()**: crea un contenedor vacío de caterva y actualiza sus diferentes *shapes*.



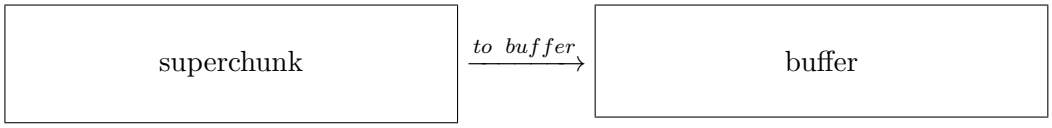
- **caterva_append()**: añade un *chunk* al contenedor de *Caterva* (si no está lleno).



- **caterva_from_buffer()**: crea un nuevo contenedor de *Caterva* a partir de los datos obtenidos de un buffer de *C*.

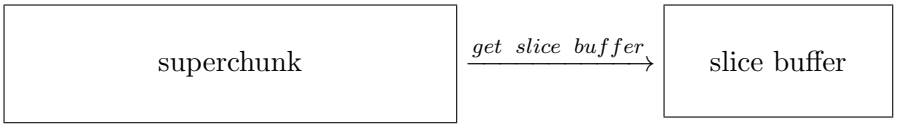


- **caterva_to_buffer()**: extrae la información de un contenedor de *Caterva* en un buffer de *C*.



- **caterva_get_slice()**: obtiene un *slice* de un contenedor de *Caterva* y lo introduce en un nuevo contenedor de *Caterva* vacío.

- **caterva_get_slice_buffer()**: obtiene un *slice* de un contenedor de *Caterva* y lo introduce en un buffer de *C*.



- **caterva_squeeze()**: elimina las dimensiones que no estén siendo utilizadas en un contenedor de *Caterva*.

2.4. Explicación detallada del proyecto realizado en la empresa

2.4.1. Metodología y definición de tareas

La mayoría de las tareas realizadas durante la estancia están relacionadas con el desarrollo de los métodos de la librería *Caterva* con el objetivo de que se adapten a la nueva funcionalidad de los *blocks*, que dividen a los *chunks*.

De esta manera, la tarea más importante en lo relativo a aprovechar la nueva funcionalidad para reducir los tiempos de descompresión es el desarrollo del método "caterva_get_slice_buffer_2". Esto es consecuencia de que ahora se deberá obtener un slice más específico, debido a la introducción de los *blocks*, así que su algoritmo será el más complejo a desarrollar. En cuanto al resto de métodos, en la mayoría de los casos, únicamente se deberán mantener las estructuras actualizadas con la nueva concepción de los *blocks*, para que puedan ser usadas como corresponde en el método "caterva_get_slice_buffer_2".

De igual forma, hay otras tareas secundarias más relacionadas con la instalación y gestión del software necesario para llevar a cabo las tareas principales, como por ejemplo los diferentes *forks*, *pulls* y *commits* que se realizan en *GitHub*, necesarios para mantener al día la información.

Por último, entre mis tareas finales también se incluye el desarrollo de una función paralela de *Blosc* que funcione con varios *threads* para aprovechar al máximo la nueva funcionalidad de *Caterva*.

2.4.2. Planificación temporal de las tareas

Primera semana

Periodo 11/11/19 – 24/11/19

En estas dos semanas mi labor consistió en la instalación de las diferentes herramientas necesarias para el proyecto y la comprensión del código de la librería a editar, todo ello con la ayuda de los trabajadores de la empresa, Francesc y Aleix.

En primer lugar, para poder trabajar necesitaba un entorno de programación, y elegí CLion por recomendación. Por esta razón, durante el período previo a mi incorporación a la empresa pedí una licencia online a JetBrains e instalé el programa. Una vez llegué a la empresa configuré los diferentes ajustes de CLion e instalé complementos útiles como VisualStudio (aunque finalmente usé WSL) o CMake, para posteriormente hacer un *fork* de la librería *Caterva* a CLion que me permitiera poder trabajar con ella localmente desde mi ordenador.

Una vez tenía el proyecto a mi disposición me dispuse a configurar GitHub para una correcta administración del código online con la intención de que al subir mis actualizaciones del programa el resto de trabajadores pudiesen ver los cambios.

Finalmente, ya con todas las herramientas necesarias para trabajar con el programa, comencé con la tarea de leer el código, comprendiendo lo que hacía cada función a base de hacerme esquemas y apuntar lo que se hacía en cada sección del código. Este cometido no fue trivial, ya que partía de una base muy limitada y el código era extenso y complejo, pero gracias a la ayuda de Francesc y Aleix fui entendiendo su funcionamiento.

Para la próxima quincena, la tarea que pretendía realizar era la de empezar a escribir el nuevo código, comenzando por las nuevas versiones de los métodos "caterva_empty_array" y "caterva_append", dado que son las más básicas.

Segunda semana

Periodo 25/11/19 – 8/12/19

En las dos semanas previas instalé y configuré las herramientas necesarias para mi tarea. Asimismo, leí y comprendí gran parte del código sobre el que tenía que trabajar.

En estas dos semanas mi actividad en la empresa daba comienzo con el desarrollo de un nuevo código que implementase las funcionalidades requeridas en mis objetivos. Para llevar a cabo este propósito, al final de cada sesión realicé un "*commit*", es decir, una operación de GitHub que consiste en guardar en la nube las actualizaciones realizadas en el código del repositorio local.

Para empezar, tuve que editar la estructura de la variable caterva_array en el header caterva.h, la cual representa un contenedor de datos de *Caterva*, para añadirle los nuevos parámetros que necesitaría un objeto de esta clase para su correcto funcionamiento.

Posteriormente, me dispuse a crear un método "caterva_empty_array_2" a partir del método "caterva_empty_array" previamente existente en la librería, que crea un "caterva_array" vacío e inicializa varios de sus parámetros. Este proceso no sólo consistió en añadir los nuevos parámetros e inicializarlos, sino que hice que ya se actualizaran sus valores para el caso concreto de cada llamada al método.

Una vez acabado este método más básico, dado que no había ningún test hecho para comprobar su eficacia, decidí escribir el siguiente método más básico, "caterva_append_2", a partir de "caterva_append", que sí que contaba con un test que me permitiría comprobar que ambos métodos funcionaban. La implementación de este método fue costosa, ya que si bien el método sólo añade una partición al "caterva_array", en la nueva versión debía reordenar el buffer del que se extraían los datos de la partición para que estuviesen subparticionados. Para llevar a cabo este proceso creé un nuevo método "caterva_repart_chunk" (al que llama "caterva_append") y que está basado en el código del método "caterva_from_buffer".

Una vez implementado el método "caterva_repart_chunk" creé un nuevo test "test_append_2"

basado en "test_append", que comprobaba que el método "caterva_append_2" no daba problemas. Sin embargo, este test no verificaba que "caterva_repart_chunk" cumpliese con su propósito, así que creé otro test "test_repart_chunk" que sí se encargaba de esto.

Una vez implementados y optimizados los dos tests para que aseguraran el correcto funcionamiento de los nuevos métodos los añadí al conjunto de tests de la librería. Realicé dicha función con la finalidad de que se ejecutasen junto a ellos y se comprobase continuamente que no se realizan cambios que provoquen errores en estos métodos.

Tercera semana

Periodo 9/12/19 – 22/12/19

En las dos semanas previas desarrollé los métodos "caterva_empty_array_2", "caterva_repart_chunk" y "caterva_append_2" junto con los tests "test_repart_chunk" y "test_append_2", ambos de la librería *Caterva*.

Durante este periodo, sin embargo, mi tarea fue comenzar a desarrollar el nuevo código del método "caterva_get_slice_2", basado en "caterva_get_slice", el cual descomprimía los *chunks* que contuvieran los datos solicitados y extraía estos datos de cada *chunk* descomprimido. El nuevo método debía descomprimir una cantidad menor de datos aprovechando la nueva estructura de subparticiones implementada. Esta parte del código era la más compleja algorítmicamente hablando, así que fue el método que más tiempo me llevó implementar. En ese momento, solamente descomprimía las subparticiones que contienían datos, no los *chunks* enteros, y desde ahí extraía los datos. Esta diferencia en el tamaño de descompresión es lo que se esperaba que marcara la diferencia en cuanto a la eficiencia del programa, ya que el proceso de descompresión es realmente costoso.

El método "caterva_get_slice" llama a otro método "caterva_get_slice_buffer", que es el que se encarga de hacer la mayor parte del trabajo, así que, para que mi "caterva_get_slice_2" funcionara, tuve que crear un nuevo "caterva_get_slice_buffer_2" basado en "caterva_get_slice_buffer".

Una vez implementado el método y comprobada su funcionalidad teórica sobre el papel, me dispuse a comprobar su funcionamiento en la práctica. Para ello creé un test "test_get_slice_buffer_2", basado en "test_get_slice_buffer" ya existente, que presentaba una variada batería de pruebas para el método. Sin embargo, para que el test funcionara bien también tuve que crear un método "caterva_from_buffer_2" basado en "caterva_from_buffer", ya que el test usaba este método y ahora debía hacerlo teniendo en cuenta las subparticiones.

Tuve muchos problemas para imprimir por pantalla los accesos a direcciones de memoria concretas, dado que no había programado en *C* antes de estas prácticas y el uso de punteros es un tanto confuso. No obstante, Francesc me animó a leer el libro "Understanding and Using C Pointers" de Richard Reese (5), el cual me ayudó a comprender mejor este tema. Gracias a su ayuda junto a la de Aleix pude resolver los problemas que me iban surgiendo.

Más allá de estos problemas conseguí implementar "caterva_get_slice_buffer_2" y hacerlo funcionar correctamente.

Cuarta semana

Periodo 6/1/20 – 19/1/20

En las dos semanas previas desarrollé el método "caterva_get_slice_2" que requería de nuevos métodos como "caterva_get_slice_buffer_2" y "caterva_from_buffer_2", y también desarrollé el test "test_get_slice_buffer_2", ya que el método "caterva_get_slice_buffer_2" es el que hace la mayor parte del trabajo y era primordial comprobar que funcionaba correctamente.

Mi tarea a realizar durante estos días fue la de desarrollar el test "test_get_slice_2", basado en "test_get_slice", que permite comprobar el correcto funcionamiento de "caterva_get_slice". Este método, a su vez, utiliza el método "caterva_to_buffer", así que tuve que desarrollar un nuevo "caterva_to_buffer_2" que tuviese en cuenta las subparticiones.

El método "caterva_to_buffer_2" crea un buffer a partir de un caterva_array, pero cuando des-

comprimimos un *chunk* de un caterva_array, éste está reordenado para tener en cuenta las subparticiones (*reparted chunk*). De este modo, era necesario un método inverso a "caterva_repart_chunk" que deshiciera este proceso, debido a lo cual creé "caterva_derepart_chunk", que se ocupa precisamente de esto.

Una vez desarrollado "caterva_derepart_chunk", también hice un test "test_derepart_chunk" para comprobar que funcionaba bien. Dado que "caterva_derepart_chunk" es el inverso de "caterva_repart_chunk", si se usa "caterva_repart_chunk" sobre un buffer, y "caterva_derepart_chunk" sobre el resultado, se debe obtener el buffer que se ha usado de *input*. Por tanto, para el test copié "test_repart_chunk" e hice que, después de usarse "caterva_repart_chunk", se usara el inverso sobre el resultado y otra vez "caterva_repart_chunk" sobre la salida. De esta forma, si daba el mismo resultado que en el test "test_repart_chunk", el método "caterva_derepart_chunk" funcionaba correctamente.

Quinta semana

Periodo 20/1/20 – 2/2/20

En las dos semanas previas desarrollé el test "test_get_slice_2", que además requería de nuevos métodos "caterva_to_buffer_2" y "caterva_derepart_chunk". A su misma vez, desarrollé tests para estos métodos.

En ese transcurso temporal, mi ocupación consistía en usar el test "test_get_slice_2" para comprobar el correcto funcionamiento de "caterva_get_slice_2". Al principio encontré abundantes errores y, mediante el uso del *debugging* para este test, descubrí que el origen de estos se encontraba en el método "caterva_append_2", ya que gracias a "caterva_repart_chunk" el *padding* de las subparticiones se trataba correctamente, pero el del propio *chunk* no.

El problema consistía en que el *input* que la función pedía era un *buffer* del tamaño de un *chunk*, de tal forma que este *buffer* ya debía venir con los 0s del *padding* añadidos por el usuario. Para resolver esta cuestión, añadí dos nuevos parámetros a la estructura "caterva_array":

"next_size" indica el tamaño que debe tener el siguiente *chunk* a añadir y "next_pshape" su forma. De esta manera, cuando se llama a "caterva_append_2", con "next_size" se comprueba que el *chunk* que se pasa como *input* tiene el tamaño correspondiente y con "next_pshape" se añaden 0s donde sea necesario.

Para comprobar que estos cambios en "caterva_append_2" tenían el efecto deseado creé un nuevo test "test_append_2". Gracias a dicha creación conseguí que el método funcionara debidamente, estando así más cerca de que "test_get_slice_2" superara todos sus tests.

Sexta semana

Periodo 3/2/20 – 16/2/20

En las dos semanas previas modifiqué el método "caterva_append_2" y desarrollé un test para comprobar que hacía lo que debía.

En estas dos semanas mi tarea fue usar el test "test_get_slice_2" para buscar los errores que hacían que "caterva_get_slice_2" no pasase sus tests y depurar el código de "caterva.c".

En primer lugar, mediante el uso del *debugging* para "test_get_slice_2", descubrí que los errores que encontraba se debían al método "caterva_squeeze", ya que no tenía en cuenta las subparticiones, así que creé un nuevo método "caterva_squeeze_2", que sí lo hacía. Una vez implementado este nuevo método, "test_get_slice_2" por fin pasó todas las pruebas, demostrando la efectividad de "caterva_get_slice_2", que era el último método que me quedaba por implementar.

A partir de este momento, mi nuevo objetivo fue depurar el código, borrar los comentarios innecesarios y añadir algunos que hacían falta y buscar formas de mejorar la eficiencia de algunos métodos. En cuanto a esto último, Aleix me dio una idea de cómo mejorar "caterva_to_buffer_2".

Inicialmente, "caterva_to_buffer_2" funcionaba de acuerdo al siguiente diagrama:

schunk $\xrightarrow{decompress}$ rchunk $\xrightarrow{derepart}$ chunk $\xrightarrow{memcpy}$ d_b

Como se puede ver en el esquema, se utilizaban tres *buffers* en este método: "rchunk", recipiente necesario para guardar la información resultante de descomprimir un *chunk*; "chunk", que contenía la partición de un *chunk* sin *padding*; "d_b", *buffer* final que guardaba todas las particiones.

De estos tres *buffers*, solamente dos eran necesarios, ya que "chunk" representaba un estado de transición entre los otros dos. Además, el programa descomprimía los *chunks* enteros en lugar de solamente las subparticiones. De esta forma, decidí crear un nuevo "caterva_to_buffer_3", que simplemente llama a "caterva_get_slice_buffer_2" con start el vector con todo 0s y stop el vector *shape*, cogiendo todo el hiperchunk.

Séptima semana

Periodo 17/2/20 – 1/3/2020

En las dos semanas previas conseguí que me funcionaran los tests del método "caterva_get_slice_2". A su vez, desarrollé un nuevo "caterva_to_buffer_3" con su correspondiente test.

En estas dos semanas mi tarea fue optimizar el "caterva_get_slice_buffer_2" haciendo *benchmarks* para conseguir la eficiencia deseada.

En una primera instancia, el *speedup* entre el programa nuevo y el original solamente era favorable cuando el *slice* era una recta, pero no tardamos en ver que esto se debía a que el método "blosc_getitem" tenía un tamaño de descompresión por defecto y no estaba descomprimiendo solo subparticiones, sino también datos innecesarios. Esto se solucionó cambiándole un parámetro para que descomprimiera bloques del tamaño adecuado. Además, para ahorrarnos que se hicieran mallocs en cada llamada al método, se decidió usar "blosc2_getitem_ctx" en su lugar.

Una vez conseguido un *speedup* más favorable frente al programa original secuencial, se comparó con el original paralelo, obteniendo, como es lógico, peores resultados. A la vista de la mejoría que implicaba la paralelización, se decidió desarrollar un nuevo "caterva_get_slice_buffer_3", que descomprimiera las subparticiones de cada *chunk* en paralelo. Para conseguir esto, al recorrer las subparticiones creé una máscara de *booleans* que indica qué subparticiones se deben descomprimir y cuáles no. Después se llama a "blosc2_schunk_decompress_chunk" pasándole la máscara como parámetro y la descompresión se lleva a cabo en paralelo.

Octava semana

Periodo 2/3/20 – 15/3/2020

En las dos semanas previas desarrollé un nuevo "caterva_get_slice_buffer_3" que descomprime las subparticiones de cada *chunk* en paralelo. Sin embargo, para que funcionara correctamente se debía editar la función paralela de c-blosc2 a la que este nuevo método llama y que se encarga de descomprimir.

El objetivo de estas semanas ha sido hacer que la función "t_blosc_do_job" de blosc.c, al ser llamada compruebe si hay una *maskout*, y, si la hay, la utilice para descomprimir únicamente los bloques que se le indican. Para esto en primer lugar hice un *fork* de c-blosc2 en mi repositorio y trabajé en la rama "parallel-decompress", para posteriormente crear una sentencia *if* que diferenciara el caso en que *maskout* esté vacía o tenga elementos, actuando en consecuencia.

2.4.3. Grado de consecución de los objetivos propuestos

Dados los objetivos que se plantearon al empezar la estancia, se puede decir que fueron completados satisfactoriamente, ya que los métodos principales de *Caterva* fueron editados y trabajan de forma correcta con la nueva funcionalidad de los *blocks*, superando todos los tests que se crearon específicamente para comprobar su funcionamiento. Esto es fácilmente visible en el repositorio de *GitHub* de *Caterva*, donde cualquiera puede acceder libremente al código y

ejecutarlo desde su ordenador:

`https://github.com/Blosc/Caterva`

Además, se ha comprobado que los nuevos métodos son totalmente compatibles con *Blosc*, y trabajan de forma conjunta sin ningún problema.

2.4.4. Resultados

Finalmente, una vez terminadas las tareas propuestas para la estancia llegó el momento de comprobar si la nueva versión de *Caterva* daba sus frutos y era realmente útil. Para ello, me dispuse a comparar las velocidades de descompresión de la nueva versión con la anterior para diferentes casos.

Con este objetivo en mente, desarrollé un ejemplo en *C* que utilizaba las dos versiones de "caterva_get_slice_buffer" para extraer diferentes slices de datos en distintos contenedores de caterva. Los resultados obtenidos no fueron del todo satisfactorios, ya que se obtuvieron tiempos notablemente mejores frente a la versión antigua secuencial, pero se quedaba atrás frente a la que usa varios *threads*.

En consecuencia, llegó el desarrollo de otra nueva versión del método "caterva_get_slice_buffer" que descomprimiera los bloques en paralelo. En conclusión, al añadir la concurrencia a la nueva funcionalidad, la nueva versión resultó ser notablemente más eficiente para la descompresión de *slices* que la anterior, mejorando la velocidad de descompresión de *Blosc* y otorgándole una nueva ventaja competitiva frente al resto de compresores de altas prestaciones, ya que no es frecuente que estos trabajen con subparticiones.

Para concluir, debo añadir que me sentí acogido en la empresa y se me ofreció ayuda con las diferentes dudas que me surgieron, y quedé satisfecho tanto con el trato recibido por parte de la empresa como con mi desempeño durante la estancia.

Capítulo 3

Memoria TFG

3.1. Motivación y Objetivos

Tal y como se describe en un artículo reciente de La Gaceta de la RSME (1), cuando entran en contacto argumentos de varias disciplinas matemáticas se suelen producir resultados de una singular potencia y elegancia. Tal es el caso del conocido Teorema de Stone-Weierstrass.

En 1885, K. Weierstrass demostró la densidad de los polinomios en el espacio de las funciones reales continuas en un intervalo compacto, C([a, b]). Más adelante, M. H. Stone lo generalizó, observando que la razón de dicha densidad está en las propiedades algebraicas de los polinomios, en las propiedades analíticas de las funciones continuas, y en la propiedad topológica de la compacidad de [a, b].

La mayoría de las pruebas del teorema de Stone-Weierstrass que podemos encontrar en la literatura sobre el tema se basan en el teorema clásico de Weierstrass de aproximación de funciones continuas mediante polinomios y en el hecho de que la clausura de una subálgebra del espacio de las funciones continuas sigue siendo una subálgebra.

En este proyecto presentamos una demostración del teorema de Stone-Weierstrass que únicamente utiliza propiedades elementales de los conjuntos compactos y de las funciones continuas, además de la desigualdad de Bernoulli.

Por otra parte, una vez enunciado y demostrado el teorema, lo utilizaremos para comprobar cómo algunas redes neuronales aproximan a las funciones continuas.

3.2. Teorema de Stone-Weierstrass

La demostración que va a verse en este proyecto está basada en las que se presentan en las tesis "Using Artificial Neural Networks in the Calculation of Mortgage Prepayment Risk" de Robben Riksen (5) y en el artículo "An elementary proof of the Stone-Weierstrass Theorem" de Bruno Brosowski y Frank Deutsch (2)

3.2.1. Conceptos previos

A continuación, se van a mostrar algunos conceptos previos que serán necesarios durante la demostración del teorema.

Definición de espacio topológico de Hausdorff

Se dice que dos puntos x e y de un espacio topológico X cumplen la propiedad de Hausdorff si existen dos entornos U_x de x y U_y de y tales que $U_x \cap U_y = \varnothing$.

Un espacio de Hausdorff es un espacio topológico en el que todo par de puntos distintos del espacio verifican la propiedad de Hausdorff, es decir, puntos distintos tienen entornos disjuntos.

Definición de recubrimiento abierto

Un recubrimiento abierto de un subconjunto A de un espacio topológico X es una familia de conjuntos abiertos $\{O_i\}_{i\in I}$ de X, tales que su unión cubre a A, es decir, $\cup_{i\in I}\, O_i \supseteq A$.

Conjuntos compactos

Un subconjunto A de un espacio topológico X se dice compacto si, dado un recubrimiento abierto de A cualquiera, existe un subrecubrimiento finito del mismo.

Además, en un espacio de Hausdorff compacto, todo subconjunto cerrado es también un conjunto compacto.

Teorema de Weierstrass

Sea T un conjunto compacto y $f : T \to \mathbb{R}$ una función continua. Entonces hay al menos dos puntos x_1, x_2 pertenecientes a T donde f alcanza valores extremos absolutos, es decir, $f(x_1) \leq f(x) \leq f(x_2)$, para todo $x \in T$.

Desigualdad de Bernoulli

Si $n \in \mathbb{N}$ y $x \geq -1$, entonces $(1 + x)^n \geq 1 + nx$.

A continuación, introducimos el concepto de álgebra de funciones que después usaremos para demostrar el Lema 1.

Definición y propiedades de una subálgebra de funciones continuas

Sea T un espacio topológico compacto de Hausdorff y $C(T)$ el espacio de las funciones continuas definidas sobre T y que toman valores reales. Un subconjunto $\mathbb{A}$ de $C(T)$ se dice una subálgebra si cumple las siguientes propiedades:

(i) Si $f, g \in \mathbb{A}$, $\alpha, \beta \in \mathbb{R} \implies \alpha f + \beta g \in \mathbb{A}$.
(ii) $f, g \in \mathbb{A} \implies fg \in \mathbb{A}$.

Y si, además, $\mathbb{A}$ cumple las propiedades expuestas a continuación, entonces se dice que $\mathbb{A}$ contiene las constantes y separa puntos:

(iii) La función constante $1 \in \mathbb{A}$ ($\mathbb{A}$ contiene las constantes).
(iv) Si $t_1, t_2 \in T$, entonces existe $f \in \mathbb{A}$ tal que $f(t_1) \neq f(t_2)$ ($\mathbb{A}$ separa puntos en T).

En el espacio $C(T)$ se define la norma $\|f\| := sup_{t \in T} |f(t)|$.

Una vez vistas estas propiedades continuamos con el Lema 1 y su correspondiente demostración, para posteriormente usarlo en la demostración del Lema 2.

3.2.2. Lemas previos

Lema 1

Sea T un espacio topológico compacto de Hausdorff y sea $\mathbb{A}$ una subálgebra de $C(T)$ que contiene las constantes y separa puntos.
Si $t_0 \in T$ y U es un entorno de t_0, entonces existe un entorno $V = V(t_0)$ de t_0 con $V \subset U$, donde, para cada $0 < \epsilon < 1$, existe $f \in \mathbb{A}$ tal que:
(1) $0 \leq f(t) \leq 1$, para $t \in T$.
(2) $f(t) < \epsilon$, para $t \in V$.
(3) $f(t) > 1 - \epsilon$, para $t \in T \setminus U$.

Demostración.

Como $\mathbb{A}$ separa puntos (propiedad (iv)), para cada $t \in T \setminus U$ existe $g_t \in \mathbb{A}$ tal que $g_t(t) \neq g_t(t_0)$. Ahora definimos $h_t := g_t - g_t(t_0) \cdot 1$. Por las propiedades (i) y (iii) deducimos que $h_t \in \mathbb{A}$. Además, por definición, $h_t(t) \neq h_t(t_0) = g_t(t_0) - g_t(t_0) = 0$; por lo que podemos definir ahora la función

$$p_t := \frac{h_t^2}{\|h_t^2\|}.$$

Parece claro que $p_t \in \mathbb{A}$, $p_t(t_0) = 0$, $p_t(t) > 0$ y $0 \leq p_t \leq 1$.

Sea $U(t) = \{s \in T \mid p_t(s) > 0\}$ que es un entorno abierto de t, ya que p_t es continua. Entonces, como $U(t)$ es abierto, $T \setminus U$ es un subconjunto cerrado de T y como, además, T es compacto, $T \setminus U$ también es compacto. Por tanto, por compacidad de $T \setminus U$, existen $t_1, t_2, ..., t_m \in T \setminus U$ tales que $T \setminus U \subset \cup_{i=1}^{m} U(t_i)$.
Por otra parte, definimos la función $p := \frac{1}{m} \sum_1^m p_{t_i}$. Es evidente que $p \in \mathbb{A}$, $0 \leq p \leq 1$, $p(t_0) = 0$ y $p > 0$ en $T \setminus U$.
Como p es continua y $T \setminus U$ es compacto, por el Teorema de Weierstrass existe $0 < \delta < 1$ tal que $p \geq \delta$ en $T \setminus U$. Sea $V := \left\{t \in T \mid p(t) < \frac{\delta}{2}\right\}$. Entonces $V \subset U$ es abierto y, como $p(t_0) = 0$, entonces V es entorno abierto de t_0.
Sea k el menor número entero mayor que $1/\delta$. Entonces

$$k - 1 \leq 1/\delta,$$

es decir

$$k \leq \frac{1+\delta}{\delta} < \frac{2}{\delta},$$

y, por tanto,

$$1 < k\delta < 2.$$

Para $n = 1, 2, ...$ definimos las funciones

$$q_n(t) := (1 - p^n(t))^{k^n}.$$

Claramente $q_n \in \mathbb{A}$, $0 \leq q_n \leq 1$ y $q_n(t_0) = 1$.
Como $kp(t) < \frac{k\delta}{2} < 1$ para todo $t \in V$, entonces, por la desigualdad de Bernoulli sabemos que,

para todo $t \in V$,

$$q_n(t) = (1 - p^n(t))^{k^n} \geq 1 - (kp(t))^n \geq 1 - \left(k\frac{\delta}{2}\right)^n,$$

y este último término tiende a 1. Por tanto, podemos afirmar que $q_n > 1 - \epsilon$ en V para n suficientemente grande.
Por otra parte, para $t \in T \setminus U$, $kp(t) \geq k\delta > 1$ y, usando la desigualdad de Bernoulli,

$$\begin{aligned} q_n(t) &= \frac{1}{k^n p^n(t)}(1 - p^n(t))^{k^n} k^n p^n(t) \leq \frac{1}{(kp(t))^n}(1 - p^n(t))^{k^n}(1 + k^n p^n(t)) \\ &\leq \frac{1}{(kp(t))^n}(1 - p^n(t))^{k^n}(1 + p^n(t))^{k^n} = \frac{1}{(kp(t))^n}(1 - p^{2n}(t))^{k^n} \leq \frac{1}{(k\delta)^n} \end{aligned}$$

y, como este último término tiende a 0, entonces, para n suficientemente grande, $q_n < \epsilon$ en $T\backslash U$.
Finalmente, tomando $f := 1 - q_n$, con n suficientemente grande, obtenemos:

- Para $t \in T\backslash U$, $1 - f = q_n < \epsilon$ y, por tanto, $f > 1 - \epsilon$.
- Para $t \in V$, $1 - f = q_n > 1 - \epsilon$, es decir, $f < \epsilon$.

□

Lema 2

Sean T un espacio topológico compacto de Hausdorff y $\mathbb{A}$ una subálgebra de $C(T)$ que contiene las constantes y separa puntos. Sean A y B dos subconjuntos cerrados disjuntos de T. Entonces para todo $0 < \epsilon < 1$, existe $f \in \mathbb{A}$ tal que:
(1) $0 \leq f(t) \leq 1$, $t \in T$;
(2) $f(t) < \epsilon$, $t \in A$;
(3) $f(t) > 1 - \epsilon$, $t \in B$.

Demostración.

Sea $U = T \setminus B$. Para cada $t \in A$, elegimos el entorno $V(t)$ de t como en el Lema 1. Por compacidad de A, existen $t_1, t_2, ..., t_m \in A$ tales que $A \subset \cup_{i=1}^m V(t_i)$. Además, por la elección de $V(t_i)$, existe $f_i \in \mathbb{A}$ $(i = 1, 2, ..., m)$ con $0 \leq f_i \leq 1$, $f_i < \epsilon/m$ en $V(t_i)$ y $f_i > 1 - \epsilon/m$ en $T\backslash U = B$.

Entonces la función $f = f_1 \cdot f_2 \dots f_m$ está en $\mathbb{A}$, $0 \leq f \leq 1$, $f < \epsilon/m \leq \epsilon$ en $\cup_{i=1}^{m} V(t_i) \supset A$ y, usando la desigualdad de Bernoulli, obtenemos $f > (1 - \epsilon/m)^m \geq 1 - \epsilon$ en B.

□

3.2.3. Enunciado y demostración del teorema de Stone-Weierstrass

Sea T un espacio topológico compacto de Hausdorff. Si $\mathbb{A}$ es una subálgebra de $C(T)$ que contiene las constantes y separa puntos, entonces los elementos de $C(T)$ pueden ser uniformemente aproximados por los elementos de $\mathbb{A}$.

Es decir, dados $f \in C(T)$ y $\epsilon > 0$, existe $g \in \mathbb{A}$ tal que $sup_{t \in T}|f(t) - g(t)| < \epsilon$.

Demostración.

Sean $f \in C(T)$ y $\epsilon > 0$. Veamos que existe $g \in \mathbb{A}$ tal que $|f(t) - g(t)| < 2\epsilon$ para todo $t \in T$. Sustituyendo f por $f + \|f\|$, podemos asumir, sin pérdida de generalidad, que $f \geq 0$. También podemos asumir que $\epsilon < \frac{1}{3}$.

Elegimos un número entero n tal que $(n-1)\epsilon \geq \|f\|$ y definimos los conjuntos A_j, B_j $(j = 0, 1, \dots, n)$ de la siguiente manera:

$$A_j = \left\{ t \in T \mid f(t) \leq (j - \frac{1}{3})\epsilon \right\},$$

$$B_j = \left\{ t \in T \mid f(t) \geq (j + \frac{1}{3})\epsilon \right\}.$$

Cabe destacar que A_j y B_j son conjuntos cerrados y disjuntos en T que cumplen

$$\emptyset = A_0 \subset A_1 \subset \cdots \subset A_n = T,$$

$$B_0 \supset B_1 \supset \cdots \supset B_n = \emptyset.$$

Para cada $j = 1, \ldots, n$, por el Lema 2, existe $f_j \in \mathbb{A}$ con $0 \leq f_j \leq 1$, $f_j < \epsilon/n$ en A_j y $f_j > 1 - \epsilon/n$ en B_j.
A continuación, definimos la siguiente función en $\mathbb{A}$:

$$g(t) = \epsilon \sum_{i=0}^{n} f_i(t).$$

Para todo $t \in T = A_n$ tenemos que $t \in A_j \setminus A_{j-1}$ para algún $j \geq 1$ y, por tanto,

$$t \in \left\{ t \in T \mid f(t) \leq \left(j - \frac{1}{3}\right)\epsilon, f(t) > \left(j - 1 - \frac{1}{3}\right)\epsilon \right\}.$$

Es decir,

$$\left(j - \frac{4}{3}\right)\epsilon < f(t) \leq \left(j - \frac{1}{3}\right)\epsilon \tag{3.1}$$

y, como para todo $i \geq j$ se cumple que $t \in A_i$, entonces

$$f_i(t) < \epsilon/n, \quad \forall\ i \geq j. \tag{3.2}$$

Además, como $f(t) > \left(j - \frac{4}{3}\right)\epsilon$ y, para todo i $\leq j - 2$, tenemos

$$B_i \subset \left\{ t \in T \mid f(t) \geq \left(j - 2 + \frac{1}{3}\right)\epsilon \right\} = \left\{ t \in T \mid f(t) \geq \left(j - \frac{5}{3}\right)\epsilon \right\},$$

entonces $t \in B_i$ y, por tanto,

$$f_i(t) > 1 - \epsilon/n \quad \forall\ i \leq j - 2. \tag{3.3}$$

Usando (3.2) y sabiendo que, por el Lema 2, $0 \leq f_i \leq 1$, obtenemos

$$g(t) = \epsilon \sum_{i=0}^{n} f_i(t) = \epsilon \sum_{i=0}^{j-1} f_i(t) + \epsilon \sum_{i=j}^{n} f_i(t) \leq j\epsilon + \epsilon(n - j + 1)\epsilon/n \leq j\epsilon + \epsilon^2 < \left(j + \frac{1}{3}\right)\epsilon.$$

Usando (3.3), para $j \geq 2$ obtenemos

$$\begin{aligned} g(t) &= \epsilon \sum_{i=0}^{n} f_i(t) \geq \epsilon \sum_{i=0}^{j-2} f_i(t) \geq \epsilon \sum_{i=0}^{j-2} (1 - \epsilon/n) \geq (j-1)\epsilon(1 - \epsilon/n) \\ &= (j-1)\epsilon - (j-1)\frac{\epsilon^2}{n} > (j-1)\epsilon - \epsilon^2 > (j-1)\epsilon - \frac{1}{3}\epsilon > \left(j - \frac{4}{3}\right)\epsilon. \end{aligned}$$

Además, para $j = 1$

$$g(t) = \epsilon \sum_{i=0}^{n} f_i(t) \geq \epsilon \sum_{i=0}^{n} 0 = 0 \geq \left(j - \frac{4}{3}\right)\epsilon = -\frac{1}{3}\epsilon.$$

Por tanto, para $t \in A_j$ tenemos por (3.1):

$$\left(j - \frac{4}{3}\right)\epsilon \leq f(t) \leq \left(j - \frac{1}{3}\right)\epsilon,$$

y, como acabamos de ver, también

$$\left(j - \frac{4}{3}\right)\epsilon \leq g(t) \leq \left(j + \frac{1}{3}\right)\epsilon.$$

Finalmente, podemos concluir que

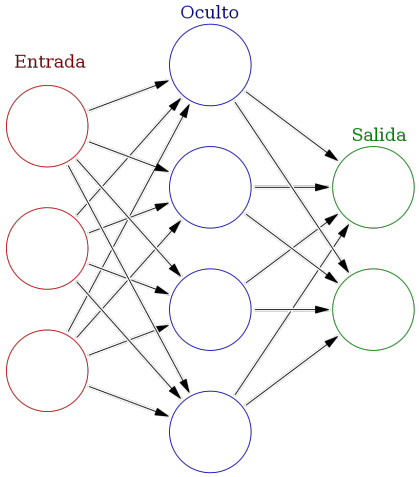
$$|f(t) - g(t)| \leq \left(j + \frac{1}{3}\right)\epsilon - \left(j - \frac{4}{3}\right)\epsilon = \frac{5}{3}\epsilon < 2\epsilon.$$

□

3.3. Aplicación del teorema de Stone-Weierstrass en redes neuronales

Una red neuronal artificial consiste en un conjunto de neuronas artificiales conectadas entre sí de manera que, al recibir un *input*, este atraviesa la red neuronal donde se le aplican diferentes operaciones dando lugar a un *output*.

Además, las neuronas se agrupan en distintas capas (*layers*), de forma que las diferentes neuronas de una misma capa realizan la misma operación sobre los diferentes datos que les llegan de la capa anterior. A partir de ahora se denotará como L al número de capas de una red y como d_l al número de neuronas de la capa l.



Las redes neuronales se suelen usar para aproximar una función (normalmente desconocida y complicada) $f : \mathbb{R}^{d_0} \to \mathbb{R}^{d_L}$. Por tanto, el objetivo es crear una red que, para cada *input* $(x_1, ..., x_{d_0})$ produzca un *output* $(y_1, ..., y_{d_L})$ que aproxime a $f(x_1, ..., x_{d_0})$.

Una forma de comprobar que una red neuronal aproxima a las funciones continuas de varias variables es estudiar si satisface las hipótesis del teorema de Stone-Weierstrass. Recordemos, según hemos visto en la sección anterior, que las hipótesis del teorema de Stone-Weierstrass son las siguientes:

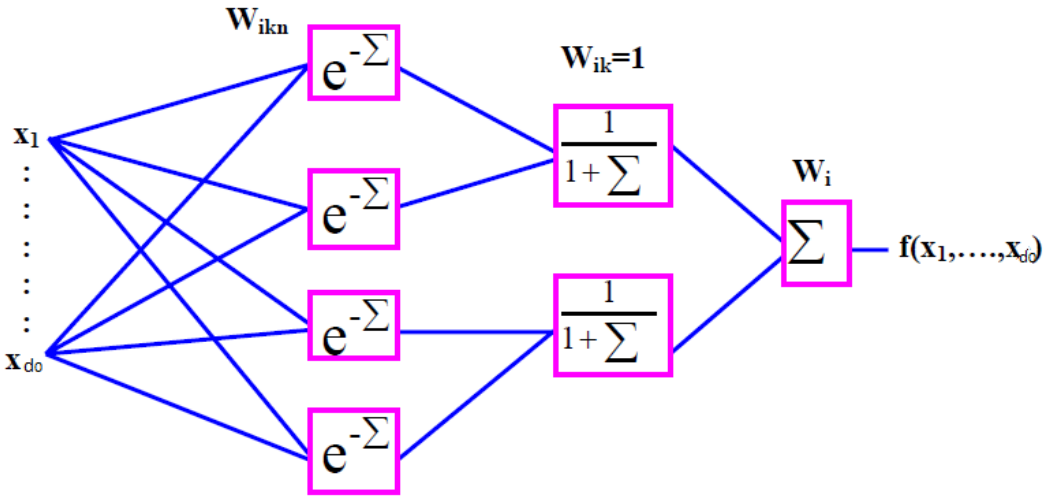
- **Función constante**: la red neuronal ha de ser capaz de computar la función constante $f(x) = 1$.
- **Separación de puntos**: la red neuronal tiene que poder computar funciones de forma que para diferentes puntos den lugar a diferentes valores.
- **Cerrada para la suma y el producto por un escalar**: la red neuronal debe tener la capacidad de computar sumas y productos por números reales.

- **Cerrada para el producto**: la red neuronal debe poder computar el producto de dos funciones.

Un tipo de red ampliamente utilizado es

$$f(x_1, ..., x_N) = \sum_{i=1}^{I} w_i \, \sigma \left(\sum_{n=1}^{N} w_{in} x_n \right).$$

Además, estas redes neuronales suelen tener estructura de árbol como la del siguiente ejemplo, de manera que muchas neuronas de una capa alimentan a una sola neurona de la capa siguiente:



3.3.1. Algunas redes neuronales que satisfacen el teorema de Stone-Weierstrass

A continuación, vamos a ver cinco ejemplos de redes neuronales que satisfacen las hipótesis del teorema de Stone-Weierstrass y, por tanto, aproximan funciones continuas de varias variables sobre conjuntos compactos. Estos ejemplos están inspirados en el artículo "The Stone-Weierstrass theorem and its application to neural networks" de Neil E. Cotter (3) y en la tesis "Neural Networks Satisfying Stone-weiestrass Theorem And Approximating" de Pinal Thakkar (6).

En este tipo de redes que nos disponemos a estudiar, la condición de ser cerrada para el producto, que debe cumplir una red neuronal para satisfacer el teorema de Stone-Weierstrass, va a requerir que sea capaz de transformar los productos en sumas. Las funciones genéricas que cumplen estos requisitos son, usualmente, las funciones exponenciales.

3.3.1.1. Redes exponenciales decrecientes

En primer lugar, se define $\mathcal{F}$ como el conjunto de las funciones definidas en el dominio compacto $D = [0,1]^N$ que tienen la siguiente forma:

$$\mathcal{F} := \left\{ f(x_1, ..., x_N) = \sum_{i=1}^{I} w_i \, exp\left(- \sum_{n=1}^{N} w_{in} x_n \right) : w_i, w_{in} \in \mathbb{R} \right\}.$$

A esta red se la conoce como red exponencial decreciente.

Ahora veamos que estas funciones forman una subálgebra de $C(D)$ que separa puntos y contiene las constantes, es decir, satisface las hipótesis del teorema de Stone-Weierstrass.

Para empezar, hay que comprobar que entre estas funciones con las que la red es capaz de trabajar se incluye la función constante $f(x_1, ..., x_N) = 1$. Vamos a comprobar qué sucede con una función f cuando $I = 1$, $w_1 = 1$ y $w_{1n} = 0$ para todo $n \in [1, N]$:

$$f(x_1, ..., x_N) = \sum_{i=1}^{1} 1 \, exp\left(- \sum_{n=1}^{N} 0 \, x_n \right) = exp\left(- \sum_{n=1}^{N} 0 \right) = 1.$$

Por tanto, la función constante sí es computable por esta red neuronal.

En segundo lugar, vamos a comprobar que la red separa puntos de D:

Dados dos puntos $\overline{x} = (x_1, ..., x_N)$ e $\overline{y} = (y_1, ..., y_N) \in D$ tales que $\overline{x} \neq \overline{y}$, entonces existe algún k tal que $x_k \neq y_k$. Para encontrar $f \in \mathcal{F}$ tal que $f(\overline{x}) \neq f(\overline{y})$ basta elegir $I = 1$, $w_1 = 1$, $w_{1k} = 1$ y $w_{1j} = 0$ para $j \neq k$, de manera que

$$f(x_1, ..., x_N) = \sum_{i=1}^{1} 1\, exp\left(-\sum_{n=1}^{N} w_{in} x_n\right) = exp\left(-x_k\right),$$

$$f(y_1, ..., y_N) = \sum_{i=1}^{1} 1\, exp\left(-\sum_{n=1}^{N} w_{in} y_n\right) = exp\left(-y_k\right),$$

de modo que, como $x_k \neq y_k$, entonces $f(x_1, ..., x_N) \neq f(y_1, ..., y_N)$.

Una vez comprobado esto, continuamos viendo que la red es cerrada para sumas y productos por números reales. Para comprobar esto, dadas dos funciones continuas f y g producidas por la red, hay que ver que su suma $f + g$ y su producto por un escalar tienen la forma de la red.

Dadas

$$f(x_1, ..., x_N) = \sum_{i=1}^{I} w_i\, exp\left(-\sum_{n=1}^{N} w_{in} x_n\right),$$

$$g(x_1, ..., x_N) = \sum_{j=1}^{J} z_j\, exp\left(-\sum_{n=1}^{N} z_{jn} x_n\right).$$

Entonces

$$\begin{aligned} af + bg &= \sum_{i=1}^{I} a w_i\, exp\left(-\sum_{n=1}^{N} w_{in} x_n\right) + \sum_{j=1}^{J} b z_j\, exp\left(-\sum_{n=1}^{N} z_{jn} x_n\right) \\ &= \sum_{k=1}^{I+J} w_k\, exp\left(-\sum_{n=1}^{N} w_{kn} x_n\right) \in \mathcal{F}, \end{aligned}$$

donde

$$w_k = \begin{cases} a w_i & 1 \leq k \leq I \\ b z_j & I + 1 \leq k \leq I + J \end{cases}$$

y

$$w_{kn} = \begin{cases} w_{in} & 1 \leq k \leq I \\ z_{jn} & I+1 \leq k \leq I+J. \end{cases}$$

Finalmente, vamos a ver que, dadas dos funciones g y h computables por la red, el producto gh forma parte de la red. En este caso, la capacidad de transformar productos en sumas es la clave, ya que para la multiplicación de exponenciales sucede lo siguiente:

Dados $x, y \in \mathbb{R}$,

$$exp(x)\, exp(y) = exp\,(x+y).$$

De acuerdo con la definición dada en $\mathcal{F}$, las funciones se definirían de la siguiente manera.

$$g(x_1, ..., x_N) := \sum_{i=1}^{I} w_i exp\left(-\sum_{n=1}^{N} w_{in} x_n\right),$$

$$h(x_1, ..., x_N) := \sum_{j=1}^{J} w_j exp\left(-\sum_{n=1}^{N} w_{jn} x_n\right).$$

Por tanto, el producto tendría la forma

$$\begin{aligned} gh &= \sum_{i=1}^{I} w_i exp\left(-\sum_{n=1}^{N} w_{in} x_n\right) \sum_{j=1}^{J} w_j exp\left(-\sum_{n=1}^{N} w_{jn} x_n\right) \\ &= \sum_{i=1}^{I}\sum_{j=1}^{J} w_i w_j\, exp\left(-\sum_{n=1}^{N} w_{in} x_n - \sum_{n=1}^{N} w_{jn} x_n\right) \\ &= \sum_{i=1}^{I}\sum_{j=1}^{J} w_i w_j\, exp\left(-\sum_{n=1}^{N} (w_{in} + w_{jn}) x_n\right) = \sum_{k=1}^{IJ} w_k exp\left(-\sum_{n=1}^{N} w_{kn} x_n\right) \in \mathcal{F}. \end{aligned}$$

En conclusión, la red cumple las cuatro hipótesis del teorema de Stone-Weierstrass.

3.3.1.2. Redes de Fourier

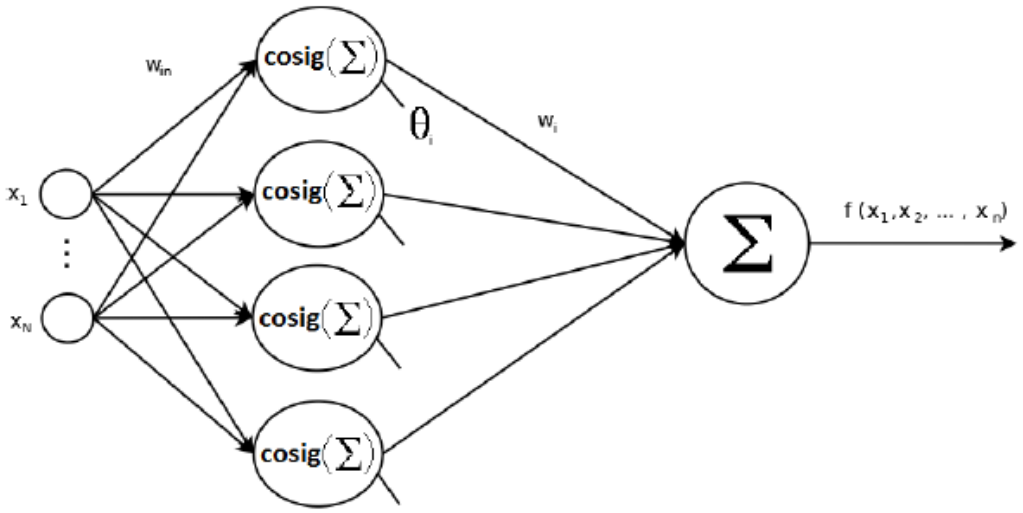
Se define $\mathcal{F}$ como el conjunto de las funciones definidas en el dominio compacto $D = [0,1]^N$ que tienen la siguiente forma:

$$\mathcal{F} := \left\{ f(x_1, ..., x_N) = \sum_{i=1}^{I} w_i \, cosig \left(\sum_{n=1}^{N} w_{in} \, x_n + \theta_i \right) : w_i, w_{in}, \theta_i \in \mathbb{R} \right\},$$

con

$$cosig(x) = \begin{cases} 0 & x \leq -\frac{1}{2} \\ \frac{1+cos(2\pi x)}{2} & -\frac{1}{2} < x < 0 \\ 1 & x \geq 0. \end{cases}$$

A esta red se la conoce como red de Fourier y su estructura en forma de árbol puede verse en la figura siguiente:



A continuación, veremos que estas redes satisfacen las hipótesis del teorema de Stone-Weierstrass.

En primer lugar, veamos que entre las funciones que forman $\mathcal{F}$ se encuentra la función constante $f(x_1, ..., x_N) = 1$. Para ello, elegimos los valores $I = 1$, $w_1 = 1$, $\theta_1 = 1$ y $w_{1n} = 0$ para todo $n \in [1, N]$:

$$f(x_1, ..., x_N) = \sum_{i=1}^{1} 1 \, cosig \left(\sum_{n=1}^{N} 0 \, x_n + 1 \right) = cosig\,(1) = 1.$$

Por tanto, la función constante sí es computable por esta red neuronal.

En segundo lugar, vamos a comprobar que la red separa puntos de D:

Dados dos puntos $\overline{x} = (x_1, ..., x_N)$ e $\overline{y} = (y_1, ..., y_N) \in D$ tales que $\overline{x} \neq \overline{y}$, entonces existe algún k tal que $x_k \neq y_k$. Para encontrar $f \in \mathcal{F}$ tal que $f(\overline{x}) \neq f(\overline{y})$ primero damos valores a ciertos parámetros: $I = 1$, $w_1 = 1$, $w_{1k} = 1$ y $w_{1j} = 0$ para $j \neq k$, de manera que

$$f(x_1, ..., x_N) = cosig \left(\sum_{n=1}^{N} w_{1n} x_n + \theta_1 \right) = cosig\,(x_k + \theta_1)\,,$$

$$f(y_1, ..., y_N) = cosig \left(\sum_{n=1}^{N} w_{1n} y_n + \theta_1 \right) = cosig\,(y_k + \theta_1)\,.$$

A continuación, basta elegir θ_1 para que se cumpla:

$$x_k + \theta_1 \leq -\frac{1}{2},$$

$$y_k + \theta_1 \geq -\frac{1}{2}.$$

De este modo, $cosig\,(x_k + \theta_1) = 0$ y $cosig\,(y_k + \theta_1) \neq 0$. Por tanto, $f(\overline{x}) \neq f(\overline{y})$.

Una vez comprobado esto, continuamos viendo que la red es cerrada para sumas y productos por números reales.

Dadas

$$f(x_1, ..., x_N) = \sum_{i=1}^{I} w_i \, cosig \left(\sum_{n=1}^{N} w_{in} \, x_n + \theta_i \right),$$

$$g(x_1, ..., x_N) = \sum_{j=1}^{J} z_j \, cosig \left(\sum_{n=1}^{N} z_{jn} \, x_n + \theta_j \right).$$

Entonces

$$\begin{aligned} af + bg &= \sum_{i=1}^{I} aw_i \, cosig\left(\sum_{n=1}^{N} w_{in}\, x_n + \theta_i\right) + \sum_{j=1}^{J} bz_j \, cosig\left(\sum_{n=1}^{N} z_{jn}\, x_n + \theta_j\right) \\ &= \sum_{k=1}^{I+J} w_k \, cosig\left(\sum_{n=1}^{N} w_{kn} x_n\right) \in \mathcal{F}, \end{aligned}$$

donde

$$w_k = \begin{cases} aw_i & 1 \leq k \leq I \\ bz_j & I+1 \leq k \leq I+J \end{cases}$$

y

$$w_{kn} = \begin{cases} w_{in} & 1 \leq k \leq I \\ z_{jn} & I+1 \leq k \leq I+J. \end{cases}$$

Finalmente, vamos a ver que, dadas dos funciones g y h computables por la red, el producto gh se puede escribir como una función de la red. En este caso, las redes de Fourier transforman el producto en suma mediante la siguiente propiedad trigonométrica:

$$cos(x)cos(y) = \frac{1}{2}\left(cos(x+y) + cos(x-y)\right).$$

Vamos a definir las funciones g y h en $\mathcal{F}$:

$$g(x_1, ..., x_N) := \sum_{i=1}^{I} w_i \, cosig\left(\sum_{n=1}^{N} w_{in}\, x_n + \theta_i\right),$$

$$h(x_1, ..., x_N) := \sum_{j=1}^{J} w_j \, cosig\left(\sum_{n=1}^{N} w_{jn}\, x_n + \theta_j\right).$$

Por tanto, el producto tendría la forma

$$
\begin{aligned}
gh &= \sum_{i=1}^{I} w_i \, cosig\left(\sum_{n=1}^{N} w_{in}\, x_n + \theta_i\right) \sum_{j=1}^{J} w_j \, cosig\left(\sum_{n=1}^{N} w_{jn}\, x_n + \theta_j\right) \\
&= \sum_{i=1}^{I}\sum_{j=1}^{J} w_i w_j \, cosig\left(\sum_{n=1}^{N} w_{in}\, x_n + \theta_i\right) \, cosig\left(\sum_{n=1}^{N} w_{jn}\, x_n + \theta_j\right) \\
&= \sum_{i=1}^{I}\sum_{j=1}^{J} w_{ij} \, cosig(A)\, cosig(B) \\
&= \begin{cases}
0 & si\ A \le -\frac{1}{2}\ o\ B \le -\frac{1}{2} \\
\sum_{i=1}^{I}\sum_{j=1}^{J} w_{ij} & si\ A \ge 0\ y\ B \ge 0 \\
\sum_{i=1}^{I}\sum_{j=1}^{J} w_{ij}\, cosig(B) = \sum_{i=1}^{I}\sum_{j=1}^{J} w_{ij}\, \frac{1+cos(2\pi B)}{2} & si\ A \ge 0\ y\ -\frac{1}{2} < B < 0 \\
\sum_{i=1}^{I}\sum_{j=1}^{J} w_{ij}\, cosig(A) = \sum_{k=1}^{IJ} w_k\, \frac{1+cos(2\pi A)}{2} & si\ -\frac{1}{2} < A < 0\ y\ B \ge 0 \\
\sum_{i=1}^{I}\sum_{j=1}^{J} w_{ij}\, \frac{1+cos(2\pi A)}{2}\, \frac{1+cos(2\pi B)}{2} & si\ -\frac{1}{2} < A < 0\ y\ -\frac{1}{2} < B < 0.
\end{cases}
\end{aligned}
$$

Este último término puede desarrollarse de la siguiente manera:

$$
\begin{aligned}
&\textstyle\sum_{i=1}^{I}\sum_{j=1}^{J} w_{ij}\, \frac{1+cos(2\pi A)}{2}\, \frac{1+cos(2\pi B)}{2} = \sum_{i=1}^{I}\sum_{j=1}^{J} \frac{w_{ij}}{4}\, (1+cos(2\pi A))(1+cos(2\pi B)) \\
&= \textstyle\sum_{i=1}^{I}\sum_{j=1}^{J} \frac{w_{ij}}{4}\, (1 + cos(2\pi A) + cos(2\pi B) + cos(2\pi A) cos(2\pi B)) \\
&= \textstyle\sum_{i=1}^{I}\sum_{j=1}^{J} \frac{w_{ij}}{4}\, \left(1 + cos(2\pi A) + cos(2\pi B) + \frac{1}{2}(cos(2\pi(A+B)) + cos(2\pi(A-B)))\right) \\
&= \textstyle\sum_{i=1}^{I}\sum_{j=1}^{J} \frac{w_{ij}}{2}\, \left(\frac{1+cos(2\pi A)}{2} + \frac{1+cos(2\pi B)}{2} - \frac{1}{2} + \frac{\frac{1+cos(2\pi(A+B))}{2} + \frac{1+cos(2\pi(A-B))}{2} - 1}{2}\right) \\
&= \textstyle\sum_{i=1}^{I}\sum_{j=1}^{J} \frac{w_{ij}}{2}\, \left(cosig(A) + cosig(B) - 1 + \frac{cosig(A+B) + cosig(A-B)}{2}\right) \in \mathcal{F}.
\end{aligned}
$$

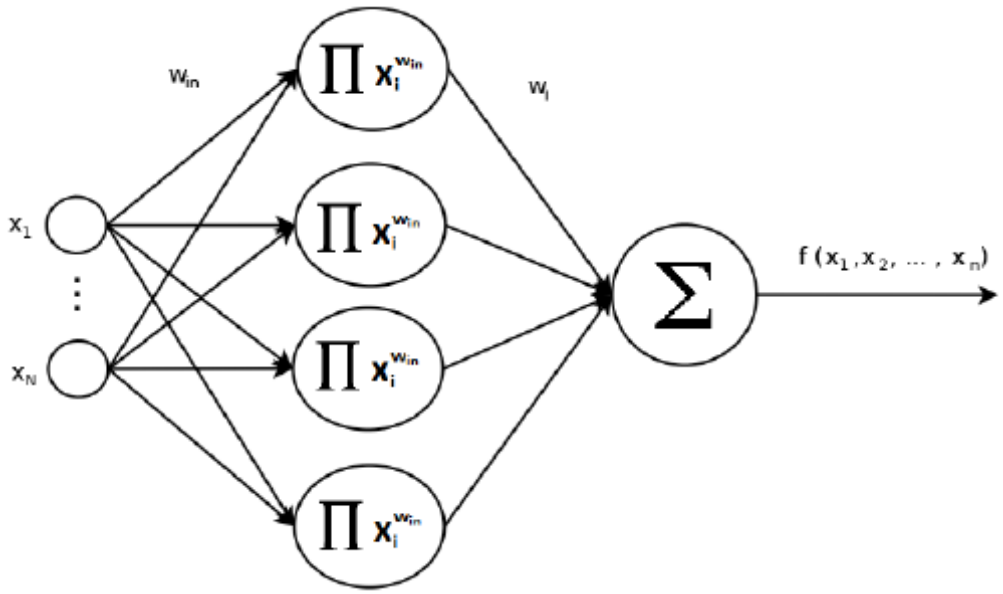
En conclusión, la red cumple las cuatro hipótesis y, por tanto, satisface el teorema de Stone-Weierstrass.

3.3.1.3. Redes Sigma-Pi modificadas y polinómicas

Se define $\mathcal{F}$ como el conjunto de las funciones definidas en el dominio compacto $D = [0,1]^N$ que tienen la siguiente forma:

$$\mathcal{F} := \left\{ f(x_1, ..., x_N) = \sum_{i=1}^{I} w_i \prod_{n=1}^{N} x_n^{w_{in}} : w_i, w_{in} \in \mathbb{R} \right\}.$$

Este tipo de red se conoce como red Sigma-Pi modificada y su estructura en forma de árbol puede verse en la figura siguiente:



Veamos que estas funciones cumplen las diferentes condiciones del teorema de Stone-Weierstrass.

Para empezar, hay que ver que la función constante $f(x_1, ..., x_N) = 1$ está entre las funciones que forman $\mathcal{F}$. Para ello, elegimos los valores $I = 1$, $w_1 = 1$, y $w_{1n} = 0$ para todo $n \in [1, N]$:

$$f(x_1, ..., x_N) = \sum_{i=1}^{1} 1 \prod_{n=1}^{N} x_n^0 = \prod_{n=1}^{N} 1 = 1.$$

Por tanto, esta red neuronal sí que puede computar la función constante.

En segundo lugar, vamos a comprobar que la red separa puntos de D:

Dados dos puntos $\overline{x} = (x_1, ..., x_N)$ e $\overline{y} = (y_1, ..., y_N) \in D$ tales que $\overline{x} \neq \overline{y}$, entonces existe algún k tal que $x_k \neq y_k$. Para encontrar $f \in \mathcal{F}$ tal que $f(\overline{x}) \neq f(\overline{y})$ basta elegir $I = 1$, $w_1 = 1$, $w_{1k} = 1$ y $w_{1j} = 0$ para $j \neq k$, de forma que

$$f(x_1, ..., x_N) = \sum_{i=1}^{1} 1 \prod_{n=1}^{N} x_n^{w_{in}} = x_k,$$

$$f(y_1, ..., y_N) = \sum_{i=1}^{1} 1 \prod_{n=1}^{N} y_n^{w_{in}} = y_k,$$

de modo que $f(x_1, ..., x_N) \neq f(y_1, ..., y_N)$.

Una vez comprobado esto, continuamos viendo que la red es cerrada para sumas y productos por números reales.

Dadas

$$f(x_1, ..., x_N) = \sum_{i=1}^{I} w_i \prod_{n=1}^{N} x_n^{w_{in}},$$

$$g(x_1, ..., x_N) = \sum_{j=1}^{J} z_j \prod_{n=1}^{N} x_n^{z_{jn}}.$$

Entonces

$$af + bg = \sum_{i=1}^{I} aw_i \prod_{n=1}^{N} x_n^{w_{in}} + \sum_{j=1}^{J} bz_j \prod_{n=1}^{N} x_n^{z_{jn}} = \sum_{k=1}^{I+J} w_k \prod_{n=1}^{N} x_n^{w_{kn}} \in \mathcal{F},$$

donde

$$w_k = \begin{cases} aw_i & 1 \leq k \leq I \\ bz_j & I+1 \leq k \leq I+J \end{cases}$$

y

$$w_{kn} = \begin{cases} w_{in} & 1 \leq k \leq I \\ z_{jn} & I+1 \leq k \leq I+J. \end{cases}$$

Por último, veamos que, dadas dos funciones g y h computables por la red, el producto gh se puede escribir como una función de la red. Para este caso, las redes transforman el producto en suma mediante propiedades de la exponencial:

$$x^n x^m = exp(n\, lnx)\, exp(m\, lnx) = exp((n+m)\, lnx) = x^{n+m}.$$

A continuación, definimos las funciones g y h en $\mathcal{F}$:

$$g(x_1, ..., x_N) := \sum_{i=1}^{I} w_i \prod_{n=1}^{N} x_n^{w_{in}},$$

$$h(x_1, ..., x_N) := \sum_{j=1}^{J} w_j \prod_{n=1}^{N} x_n^{w_{jn}}.$$

Por tanto, el producto tendría la forma

$$\begin{aligned} gh &= \left(\sum_{i=1}^{I} w_i \prod_{n=1}^{N} x_n^{w_{in}}\right)\left(\sum_{j=1}^{J} w_j \prod_{n=1}^{N} x_n^{w_{jn}}\right) = \sum_{i=1}^{I}\sum_{j=1}^{J} w_i w_j \prod_{n=1}^{N} x_n^{w_{in}} x_n^{w_{jn}} \\ &= \sum_{i=1}^{I}\sum_{j=1}^{J} w_i w_j \prod_{n=1}^{N} x_n^{w_{in}+w_{jn}} = \sum_{k=1}^{IJ} w_k \prod_{n=1}^{N} x_n^{w_{kn}} \in \mathcal{F}, \end{aligned}$$

con $w_k = w_i w_j$, $w_{kn} = w_{in} + w_{jn}$.

En conclusión, la red cumple las cuatro hipótesis del teorema de Stone-Weierstrass.

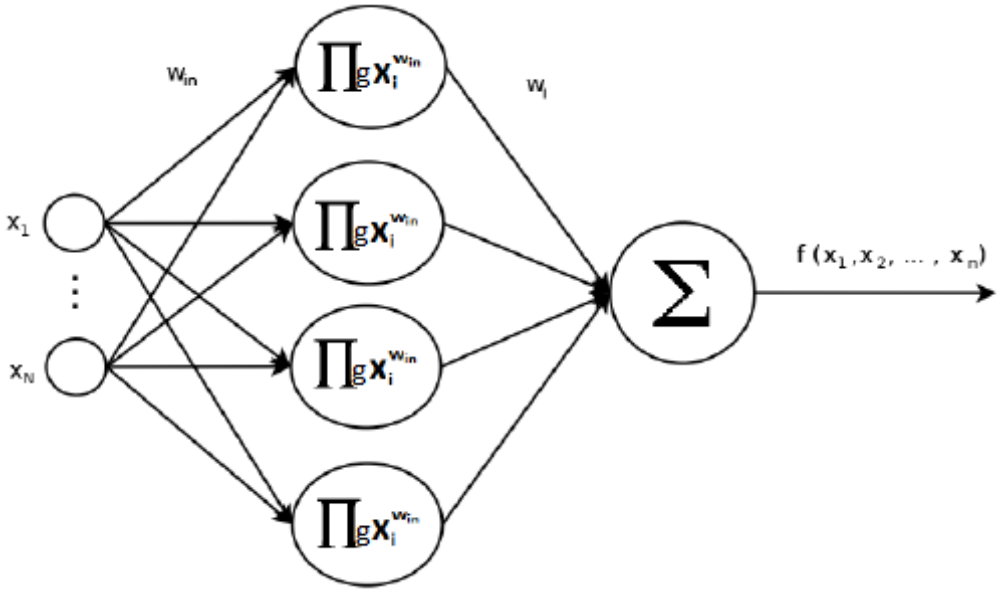
3.3.1.4. Redes de función exponenciada

Esta red se consigue preprocesando los *inputs* de una red Sigma-Pi modificada. Sea g la función de preprocesamiento, la primera capa de la red Sigma-Pi modificada computa funciones de la forma $g(x_1)^{n_1}, ..., g(x_N)^{n_N}$.

Por tanto, definimos la red de función exponenciada $\mathcal{F}$ como el conjunto de las funciones definidas en el dominio compacto $D = [0,1]^N$:

$$\mathcal{F} := \left\{ f(x_1, ..., x_N) = \sum_{i=1}^{I} w_i \prod_{n=1}^{N} g(x_n)^{w_{in}} : g \in C([0,1])\, invertible, w_i, w_{in} \in \mathbb{R} \right\}.$$

En consecuencia, este tipo de red tiene una arquitectura muy similar a la anterior:



A continuación, pasamos a ver si las funciones de $\mathcal{F}$ de esta red cumplen las condiciones del teorema de Stone-Weierstrass. Generalmente, procederemos de la misma forma que en la red anterior.

Lo primero es ver si la función constante $f(x_1, ..., x_N) = 1$ está entre las funciones que forman $\mathcal{F}$. Para ello, elegimos los valores $I = 1$, $w_1 = 1$ y $w_{1n} = 0$ para todo $n \in [1, N]$:

$$f(x_1, ..., x_N) = \sum_{i=1}^{1} \prod_{n=1}^{N} g(x_n)^0 = \prod_{n=1}^{N} 1 = 1.$$

Por tanto, esta red neuronal sí que puede computar la función constante.

En segundo lugar, vamos a comprobar que la red separa puntos de D:

Dados dos puntos $\overline{x} = (x_1, ..., x_N)$ e $\overline{y} = (y_1, ..., y_N) \in D$ tales que $\overline{x} \neq \overline{y}$, entonces existe algún k tal que $x_k \neq y_k$. Para encontrar $f \in \mathcal{F}$ tal que $f(\overline{x}) \neq f(\overline{y})$ basta elegir $I = 1$, $w_1 = 1$, la función g la identidad, $w_{1k} = 1$ y $w_{1j} = 0$ para $j \neq k$, de forma que

$$f(x_1, ..., x_N) = \sum_{i=1}^{1} 1 \prod_{n=1}^{N} g(x_n)^{w_{in}} = x_k,$$

$$f(y_1, ..., y_N) = \sum_{i=1}^{1} 1 \prod_{n=1}^{N} g(y_n)^{w_{in}} = y_k,$$

de modo que $f(x_1, ..., x_N) \neq f(y_1, ..., y_N)$.

Una vez comprobado esto, continuamos viendo que la red es cerrada para sumas y productos por números reales.

Dadas

$$f(x_1, ..., x_N) = \sum_{i=1}^{I} w_i \prod_{n=1}^{N} g(x_n)^{w_{in}},$$

$$h(x_1, ..., x_N) = \sum_{j=1}^{J} z_j \prod_{n=1}^{N} g(x_n)^{z_{jn}}.$$

Entonces

$$af + bh = \sum_{i=1}^{I} aw_i \prod_{n=1}^{N} g(x_n)^{w_{in}} + \sum_{j=1}^{J} bz_j \prod_{n=1}^{N} g(x_n)^{z_{jn}} = \sum_{k=1}^{I+J} w_k \prod_{n=1}^{N} g(x_n)^{w_{kn}} \in \mathcal{F},$$

donde

$$w_k = \begin{cases} aw_i & 1 \leq k \leq I \\ bz_j & I+1 \leq k \leq I+J \end{cases}$$

y

$$w_{kn} = \begin{cases} w_{in} & 1 \leq k \leq I \\ z_{jn} & I+1 \leq k \leq I+J. \end{cases}$$

Por último, veamos que, dadas dos funciones f y h computables por la red, el producto fh se puede escribir como una función de la red. Para este caso, las redes transforman el producto en suma mediante la propiedad de la exponencial vista previamente:

$$g(x)^n g(x)^m = exp(n\, lng(x))\, exp(m\, lng(x)) = exp((n+m)\, lng(x)) = g(x)^{n+m}.$$

A continuación, definimos las funciones f y h en $\mathcal{F}$:

$$f(x_1, ..., x_N) := \sum_{i=1}^{I} w_i \prod_{n=1}^{N} g(x_n)^{w_{in}},$$

$$h(x_1, ..., x_N) := \sum_{j=1}^{J} w_j \prod_{n=1}^{N} g(x_n)^{w_{jn}}.$$

Por tanto, el producto tendría la forma

$$\begin{aligned} fh &= \left(\sum_{i=1}^{I} w_i \prod_{n=1}^{N} g(x_n)^{w_{in}}\right)\left(\sum_{j=1}^{J} w_j \prod_{n=1}^{N} g(x_n)^{w_{jn}}\right) = \sum_{i=1}^{I}\sum_{j=1}^{J} w_i w_j \prod_{n=1}^{N} g(x_n)^{w_{in}} g(x_n)^{w_{jn}} \\ &= \sum_{i=1}^{I}\sum_{j=1}^{J} w_i w_j \prod_{n=1}^{N} exp(w_{in}\, ln\, g(x_n))\, exp(w_{jn}\, ln\, g(x_n)) \\ &= \sum_{i=1}^{I}\sum_{j=1}^{J} w_i w_j \prod_{n=1}^{N} exp((w_{in}+w_{jn})\, ln\, g(x_n)) = \sum_{i=1}^{I}\sum_{j=1}^{J} w_i w_j \prod_{n=1}^{N} g(x_n)^{w_{in}+w_{jn}} \\ &= \sum_{k=1}^{IJ} w_k \prod_{n=1}^{N} g(x_n)^{w_{kn}} \in \mathcal{F}, \end{aligned}$$

con $w_k = w_i w_j$, $w_{kn} = w_{in} + w_{jn}$.

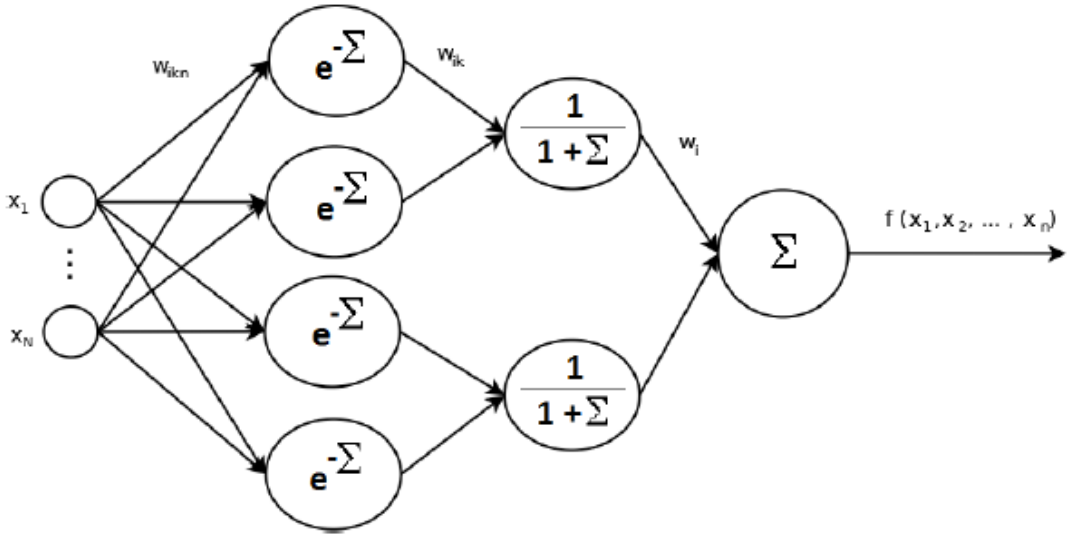
En conclusión, la red cumple las cuatro hipótesis del teorema de Stone-Weierstrass.

3.3.1.5. Redes logísticas modificadas

Se define $\mathcal{F}$ como el conjunto de las funciones definidas en el dominio compacto $D = [0,1]^N$ que tienen la siguiente forma:

$$\mathcal{F} := \left\{ f(x_1, ..., x_N) = \sum_{i=1}^{I} w_i \left[1 + \sum_{k=1}^{K} exp\left(- \sum_{n=1}^{N} w_{ikn} x_n \right) \right]^{-1} : w_i, w_{ikn} \in \mathbb{R} \right\}.$$

Esta red se conoce como red logística modificada y su estructura en forma de árbol puede verse en la figura siguiente:



Ahora veamos que estas funciones forman una subálgebra de $C(D)$ que separa puntos y contiene las constantes, es decir, satisface las hipótesis del teorema de Stone-Weierstrass.

Para empezar, hay que comprobar que entre estas funciones con las que la red es capaz de trabajar se incluye la función constante $f(x_1, ..., x_N) = 1$. Vamos a comprobar qué sucede con una función f cuando $I = 1$, $K = 1$, $w_1 = 2$ y $w_{1n} = 0$ para todo $n \in [1, N]$:

$$f(x_1, ..., x_N) = \sum_{i=1}^{1} 2 \left[1 + \sum_{k=1}^{1} exp\left(- \sum_{n=1}^{N} 0 x_n \right) \right]^{-1} = 2\ [1 + exp\ (0)]^{-1} = \frac{2}{1+1} = 1.$$

Por tanto, la función constante sí es computable por esta red neuronal.

En segundo lugar, vamos a comprobar que la red separa puntos de D:

Dados dos puntos $\overline{x} = (x_1, ..., x_N)$ e $\overline{y} = (y_1, ..., y_N) \in D$ tales que $\overline{x} \neq \overline{y}$, entonces existe algún l tal que $x_l \neq y_l$. Para encontrar $f \in \mathcal{F}$ tal que $f(\overline{x}) \neq f(\overline{y})$ basta elegir $I = 1$, $K = 1$, $w_1 = 1$, $w_{1k} = 1$ y $w_{1j} = 0$ para $j \neq k$, de manera que

$$f(x_1, ..., x_N) = \sum_{i=1}^{1} 1 \left[1 + \sum_{k=1}^{1} exp \left(- \sum_{n=1}^{N} w_{ikn} x_n \right) \right]^{-1} = \frac{1}{1 + exp\ (-x_l)},$$

$$f(y_1, ..., y_N) = \sum_{i=1}^{1} 1 \left[1 + \sum_{k=1}^{1} exp \left(- \sum_{n=1}^{N} w_{ikn} y_n \right) \right]^{-1} = \frac{1}{1 + exp\ (-y_l)},$$

de modo que, como $x_l \neq y_l$, entonces $f(x_1, ..., x_N) \neq f(y_1, ..., y_N)$.

Una vez comprobado esto, continuamos viendo que la red es cerrada para sumas y productos por números reales. Para comprobar esto, dadas dos funciones continuas f y g producidas por la red, hay que ver que su suma $f + g$ y su producto por un escalar tienen la forma de la red.

Dadas

$$f(x_1, ..., x_N) = \sum_{i=1}^{I} w_i \left[1 + \sum_{k=1}^{K} exp \left(- \sum_{n=1}^{N} w_{ikn} x_n \right) \right]^{-1},$$

$$g(x_1, ..., x_N) = \sum_{j=1}^{J} z_j \left[1 + \sum_{l=1}^{L} exp \left(- \sum_{n=1}^{N} z_{jln} x_n \right) \right]^{-1}.$$

Entonces

$$\begin{aligned} af + bg &= \sum_{i=1}^{I} a w_i \left[1 + \sum_{k=1}^{K} exp \left(- \sum_{n=1}^{N} w_{ikn} x_n \right) \right]^{-1} + \sum_{j=1}^{J} b z_j \left[1 + \sum_{l=1}^{L} exp \left(- \sum_{n=1}^{N} z_{jln} x_n \right) \right]^{-1} \\ &= \sum_{h=1}^{I+J} w_h \left[1 + \sum_{k=1}^{K} exp \left(- \sum_{n=1}^{N} w_{hkn} x_n \right) \right]^{-1} \in \mathcal{F}, \end{aligned}$$

donde

$$w_h = \begin{cases} a w_i & 1 \leq k \leq I \\ b z_j & I + 1 \leq k \leq I + J \end{cases}$$

y

$$w_{hkn} = \begin{cases} w_{ikn} & 1 \leq h \leq I, \\ & 1 \leq k \leq K \\ z_{jln} & I+1 \leq h \leq I+J, \\ & K+1 \leq k \leq K+L. \end{cases}$$

Finalmente, vamos a ver que, dadas dos funciones f y g computables por la red, el producto fg forma parte de la red. En este caso, la capacidad de transformar productos en sumas es evidente, ya que para la multiplicación de exponenciales sucede lo siguiente:

Dados $x, y \in \mathbb{R}$,

$$exp(x)\, exp(y) = exp\,(x+y).$$

De acuerdo a la definición dada en $\mathcal{F}$, las funciones se definirían de la siguiente manera.

$$f(x_1, ..., x_N) = \sum_{i=1}^{I} w_i \left[1 + \sum_{k=1}^{K} exp\, \left(-\sum_{n=1}^{N} w_{ikn} x_n\right)\right]^{-1},$$

$$g(x_1, ..., x_N) = \sum_{j=1}^{J} z_j \left[1 + \sum_{l=1}^{L} exp\, \left(-\sum_{n=1}^{N} z_{jln} x_n\right)\right]^{-1}.$$

Por tanto, el producto tendría la forma

$$\begin{aligned} fg &= \sum_{i=1}^{I} w_i \left[1 + \sum_{k=1}^{K} exp\, \left(-\sum_{n=1}^{N} w_{ikn} x_n\right)\right]^{-1} \sum_{j=1}^{J} z_j \left[1 + \sum_{l=1}^{L} exp\, \left(-\sum_{n=1}^{N} z_{jln} x_n\right)\right]^{-1} \\ &= \sum_{i=1}^{I} \sum_{j=1}^{J} w_i z_j \left[1 + \sum_{k=1}^{K} exp\, \left(-\sum_{n=1}^{N} w_{ikn} x_n\right)\right]^{-1} \left[1 + \sum_{l=1}^{L} exp\, \left(-\sum_{n=1}^{N} z_{jln} x_n\right)\right]^{-1} \\ &= \sum_{i=1}^{I} \sum_{j=1}^{J} w_i z_j \left[1 + \sum_{k=1}^{K} exp\, \left(-\sum_{n=1}^{N} w_{ikn} x_n\right) + \sum_{l=1}^{L} exp\, \left(-\sum_{n=1}^{N} z_{jln} x_n\right)\right. \\ &\quad \left. + \sum_{k=1}^{K} \sum_{l=1}^{L} exp\, \left(-\sum_{n=1}^{N} (w_{ikn} + z_{jkn}) x_n\right)\right]^{-1} \\ &= \sum_{h=1}^{IJ} w_h \left[1 + \sum_{p=1}^{P=K+L+KL} exp\, \left(-\sum_{n=1}^{N} w_{hpn} x_n\right)\right]^{-1} \in \mathcal{F}, \end{aligned}$$

donde

$$w_h = w_i z_j$$

y

$$w_{hpn} = \begin{cases} w_{ikn} & 1 \leq p \leq K \\ z_{jln} & K+1 \leq p \leq K+L \\ w_{ikn} + z_{jln} & K+L \leq p \leq P = K+L+KL. \end{cases}$$

En conclusión, la red cumple las cuatro hipótesis del teorema de Stone-Weierstrass.

Capítulo 4

Conclusiones

En cuanto a las prácticas, los avances más significativos realizados por mi durante la estancia se encuentran en el informe de *Blosc* de 2020:

`https://blosc.org/posts/mid-2020-progress-report/`.

En este informe aparecen tanto la nueva funcionalidad de *Maskout* en el apartado *C-Blosc2* como la funcionalidad de *blocking* en el apartado *Caterva/cat4py*.

Además, el código desarrollado puede encontrarse en el Anexo I.

Por otra parte, la mejora conseguida en la librería *Caterva* se puede observar en el *benchmark* "Caterva performance against Zarr and HDF5": `https://github.com/Blosc/cat4py/blob/master/notebooks/slicing-performance.ipynb`.

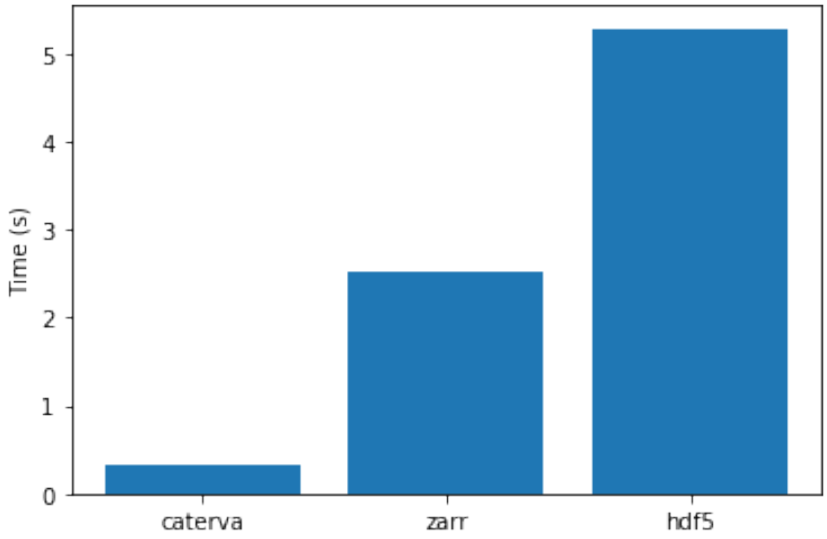
Este ejemplo está destinado a comparar velocidades de obtención de hiperplanos (*slices*) en una matriz multidimensional utilizando diferentes compresores: *Caterva*, *Zarr* y *HDF5*.

En él se trabaja con un *array* tridimensional, cuyas propiedades son:

- Tamaño del *array*: shape = [500, 250, 200]
- Tamaño del *chunk* (partición): chunkshape = [500, 10, 200]
- Tamaño del *block* (subpartición): blockshape = [40, 10, 50]

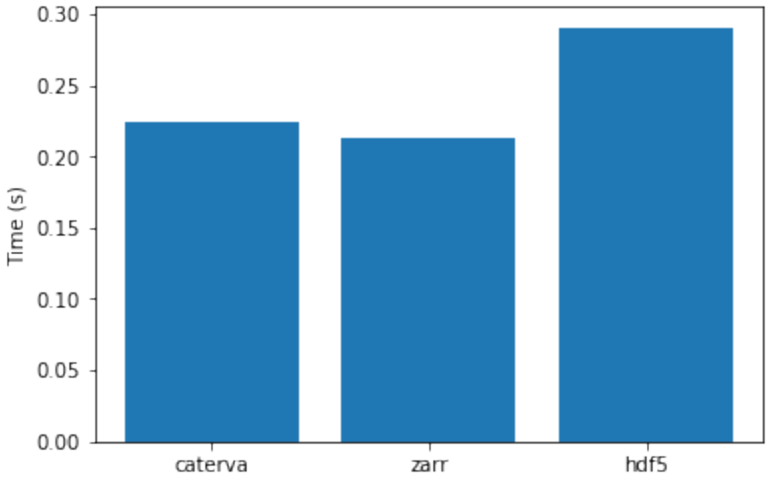
Además, se analiza la obtención de un hiperplano en cada dimensión con los diferentes compresores.

Figura 4.1: Tiempos de descompresión del hiperplano del eje 0



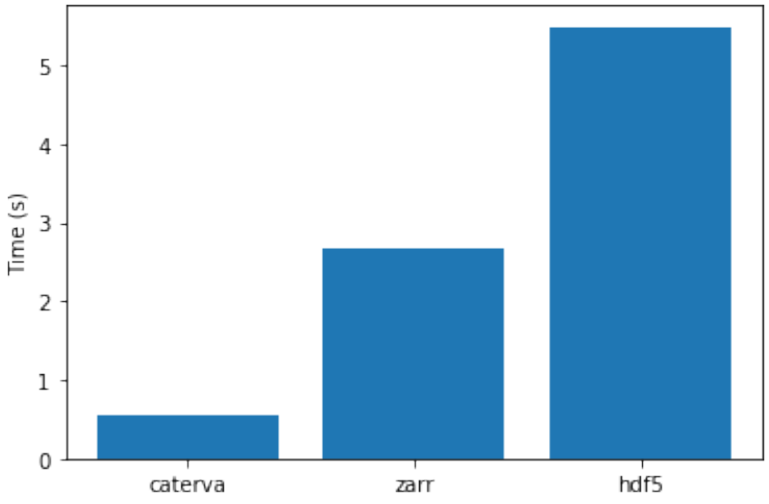
Dado que en el eje 0 el tamaño de *block* es mucho menor que el tamaño de *chunk*, entonces vemos que el tiempo de descompresión de *Caterva* es mucho menor que el de los otros compresores, debido a que no usan subparticiones.

Figura 4.2: Tiempos de descompresión del hiperplano del eje 1



En el eje 1, sin embargo, el tamaño de *block* es igual que el tamaño de *chunk*, entonces vemos que el tiempo de descompresión de *Caterva* es apenas mayor que el de Zarr y menor que el de HDF5, a pesar de que los otros compresores no tienen la carga extra del código destinado a lidiar con las subparticiones.

Figura 4.3: Tiempos de descompresión del hiperplano del eje 2



Por último, en el eje 2 el ejemplo es muy parecido al del eje 0, aunque la diferencia entre el tamaño de *block* y el tamaño de *chunk* no es tan grande, y de la misma manera la diferencia en el tiempo de descompresión no es tan grande.

En conclusión, para ejemplos en los que el tamaño de *block* es notablemente menor que el tamaño de *chunk* se obtienen resultados mejores para *Caterva* que para el resto de compresores que no utilizan subparticiones.

En cuanto al Proyecto de Final de Grado, me gustaría resaltar el hecho de que un teorema tan importante como el teorema de Stone-Weierstrass pueda demostrarse con técnicas elementales al alcance de cualquier estudiante del grado de Matemáticas. Por otra parte, ha resultado realmente interesante estudiar en profundidad la aplicación de un teorema del ámbito de las matemáticas puras a un campo tan en auge y aplicado como el de las redes neuronales. Queda pendiente el análisis de algunas otras redes como la red de fracción parcial, la red de perceptrón o la red de función de paso.

Bibliografía

[1] Julio Bernués, *El Teorema de Stone-Weierstrass*, La Gaceta de la RSME, **13**, 2010, 705–711.

[2] Bruno Brosowski y FRank Deutsch, *An elementary proof of the Stone-Weierstrass Theorem*, Proceeding of the American Mathematical Society, **81**, 1981, 89-92.

[3] Neil E. Cotter, *The Stone-Weierstrass theorem and its application to neural networks*, IEEE Transactions on neural networks, **1**, 1990, 290-295.

[4] Richard M. Reese, *Understanding and Using C Pointers*, O'Reilly Media Inc., 2013.

[5] Robben Riksen, 'Using Artificial Neural Networks in the Calculation of Mortgage Prepayment Risk′, PhD. Thesis, University of Amsterdam, 2017.

[6] Pinal Thakkar, 'Neural Networks Satisfying Stone-weiestrass Theorem And Approximating′, PhD. Thesis, University of Central Florida, 2004.

Anexo A

Anexo I

En esta sección se va a ver el código de los archivos de C caterva.h y caterva.c.

A.1. Caterva.h

Al ser el archivo cabecera de Caterva, este contiene su API pública. De este modo, en dicho documento se dispone de los diferentes métodos y estructuras que se utilizan en Caterva:

```
#ifndef CATERVA_HEADER_FILE
#define CATERVA_HEADER_FILE

#include <stdio.h>
#include <stdlib.h>
#include <blosc2.h>
```

```
/* Version numbers */
#define CATERVA_VERSION_MAJOR    0    /* for major interface/format changes  */
#define CATERVA_VERSION_MINOR    2    /* for minor interface/format changes  */
#define CATERVA_VERSION_RELEASE  3    /* for tweaks, bug-fixes, or development */

#define CATERVA_VERSION_STRING   "0.2.3-dev"  /* string version.  Sync with above! */
#define CATERVA_VERSION_DATE     "2019-10-28"    /* date version */

/* The version for metalayer format; starts from 0 and it must not exceed 127 */
#define CATERVA_METALAYER_VERSION 0

/* The maximum number of dimensions for Caterva arrays */
#define CATERVA_MAXDIM 8


/**
 * @brief Formats to store #caterva_array_t data.
 */
typedef enum {
    CATERVA_STORAGE_BLOSC,
    //!< Indicates that data is stored using a Blosc superchunk.
    CATERVA_STORAGE_PLAINBUFFER,
    //!< Indicates that data is stored using a plain buffer.
} caterva_storage_t;


/**
 * @brief A context for caterva containers.
 *
 * In parenthesis it is shown the default value used internally when a \c NULL value
 * is passed to the constructor.
 */
typedef struct {
```

```
    void *(*alloc)(size_t);
    //!< The allocation memory function used internally (malloc)
    void (*free)(void *);
    //!< The free memory function used internally (free)
    blosc2_cparams cparams;
    //!< The blosc compression params used
    blosc2_dparams dparams;
    //!< The blosc decompression params used
} caterva_ctx_t;


/**
 * @brief A dimensions vector that can represent shapes or points
 */
typedef struct {
    int64_t dims[CATERVA_MAXDIM];
    //!< The size of each dimension
    int8_t ndim;
    //!< The number of dimensions
} caterva_dims_t;


/**
 * @brief Default caterva dimensions vector
 */
static const caterva_dims_t CATERVA_DIMS_DEFAULTS = {
    .dims = {1, 1, 1, 1, 1, 1, 1, 1},
    .ndim = 1
};


/**
 * @brief An *optional* cache for a single block.
```

```
 *
 * When a block is needed, it is copied into this cache. In this way, if the same block
 * is needed again afterwards, it is not necessary to recover it because it is already
 * in the cache.
 */
struct part_cache_s {
    uint8_t *data;
    //!< Pointer to the block data.
    int32_t nchunk;
    //!< The block number in cache. If @p nchunk equals to -1, it means that the cache
    is empty.
};


/**
 * @brief A multidimensional container that allows compressed data.
 */
typedef struct {
    caterva_ctx_t *ctx;
    //!< Caterva context
    caterva_storage_t storage;
    //!< Storage type
    blosc2_schunk *sc;
    //!< Pointer to a Blosc superchunk
    //!< Only is used if \p storage equals to \p #CATERVA_STORAGE_BLOSC
    uint8_t *buf;
    //!< Pointer to a plain buffer where data is stored.
    //!< Only is used if \p storage equals to \p #CATERVA_STORAGE_PLAINBUFFER.
    int64_t shape[CATERVA_MAXDIM];
    //!< Shape of original data.
    int32_t pshape[CATERVA_MAXDIM];
    //!< Shape of each partition.
    int64_t eshape[CATERVA_MAXDIM];
```

```
    //!< Shape of padded data.
    int32_t spshape[CATERVA_MAXDIM];
    //!< Shape of each subpartition.
    int64_t epshape[CATERVA_MAXDIM];
    //!< Shape of padded partition.
    int32_t next_pshape[CATERVA_MAXDIM];
    //!< Shape of next partition to be appened.
    int64_t size;
    //!< Size of original data.
    int32_t psize;
    //!< Size of each partition.
    int64_t esize;
    //!< Size of padded data.
    int32_t spsize;
    //!< Size of each subpartition.
    int64_t epsize;
    //!< Size of padded partition.
    int64_t next_size;
    //!< Size of next partiton to be appened.
    int8_t ndim;
    //!< Data dimensions.
    bool empty;
    //!< Indicate if an array is empty or is filled with data.
    bool filled;
    //!< Indicate if an array is completely filled or not.
    int64_t nparts;
    //!< Number of partitions in the array.
    struct part_cache_s part_cache;
    //!< A partition cache.
} caterva_array_t;


/**
```

```
 * @brief Create a context for Caterva functions.
 *
 * @param all The allocation function to use internally. If it is \c NULL, malloc is
 * used.
 * @param free The free function to use internally. If it is \c NULL, free is used.
 * @param cparams The compression parameters used when a caterva container is created.
 * @param dparams The decompression parameters used when data of a caterva container
 * is decompressed.
 *
 * @return A pointer to the new caterva context. \p NULL is returned if this fails.
 */
caterva_ctx_t *caterva_new_ctx(void *(*all)(size_t), void (*free)(void *),
blosc2_cparams cparams, blosc2_dparams dparams);


/**
 * @brief Free a caterva context
 *
 * @param ctx Pointer to the context to be freed
 *
 * @return An error code
 */
int caterva_free_ctx(caterva_ctx_t *ctx);


/**
 * @brief Create a caterva dimensions vector
 *
 * A #caterva_dims_t can be used to represent shapes or points of the container.
 *
 * @param dims The size of each dimension
 * @param ndim The number of dimensions
 *
```

```
 * @return The caterva dimensions vector created
 */
caterva_dims_t caterva_new_dims(const int64_t *dims, int8_t ndim);


/**
 * @brief Create a caterva empty container
 *
 * When a container is created, only the partition shape, the storage type and
 * the context are defined.
 * It should be noted that the shape is defined when a container is filled and not
 * when it is created.
 *
 * If \p pshape is \c NULL, the data container will be stored using a plain buffer.
 *
 * However, if \p pshape is not \c NULL, the data container will be stored using a
 * superchunk.
 * In particular, if \p fr is \c NULL it will be stored in the memory and if it is
 * not \c NULL
 * it will be stored on disk.
 *
 * @param ctx Pointer to the caterva context to be used
 * @param fr Pointer to the blosc frame used to store data on disk
 * @param pshape The shape of each partition
 * @param spshape The shape of each subpartition
 *
 * @return A pointer to the empty caterva container created
 */
caterva_array_t *caterva_empty_array(caterva_ctx_t *ctx, blosc2_frame *fr,
caterva_dims_t *pshape);

caterva_array_t *caterva_empty_array_2(caterva_ctx_t *ctx, blosc2_frame *fr,
caterva_dims_t *pshape, caterva_dims_t *spshape);
```

```
/**
 * @brief Free a caterva container
 *
 * @param carr Pointer to the container to be freed
 *
 * @return An error code
 */
int caterva_free_array(caterva_array_t *carr);


/**
 * @brief Reorders a buffer in order to be able to append subpartitions
 *
 * @param chunk Pointer to the buffer where data will be stored
 * @param size_chunk Size of the destination buffer
 * @param src A pointer to the buffer where data is stored
 * @param size_src Size of the source buffer
 * @param carr Pointer to the container where useful parameters are stored
 * @param ctx Pointer to the caterva context to be used
 *
 * @return An error code
 */
int caterva_repart_chunk(int8_t *chunk, int size_chunk, void *src, int size_src,
caterva_array_t *carr, caterva_ctx_t *ctx);


/**
 * @brief It works inversely to caterva_repart_chunk, reordering a reparted chunk
 * in order to get the original one
 *
 * @param chunk Pointer to the buffer where data will be stored
```

```
 * @param size_chunk Size of the destination buffer
 * @param rchunk Pointer to the buffer where data is stored
 * @param size_rchunk Size of the reparted chunk buffer
 * @param carr Pointer to the container where useful parameters are stored
 * @param ctx Pointer to the caterva context to be used
 *
 * @return An error code
 */
int caterva_derepart_chunk(int8_t *chunk, int size_chunk, int8_t *rchunk,
int size_rchunk, caterva_array_t *carr, caterva_ctx_t *ctx);


/**
 * Append a partition to a caterva container (until it is completely filled)
 *
 * @param carr Pointer to the container where data will be appended
 * @param part A pointer to the buffer where data is stored
 * @param partsize Size (in bytes) of the buffer
 *
 * @return An error code
 */
int caterva_append(caterva_array_t *carr, void *part, int64_t partsize);

int caterva_append_2(caterva_array_t *carr, void *part, int64_t partsize);


/**
 * @brief Create a caterva container from a frame
 *
 * @param ctx Pointer to the caterva context to be used.
 *   The context should be the same as the one used to create the array.
 * @param frame The frame for the caterva container.
 * @param copy If true, a new, sparse in-memory super-chunk is created.
```

```
 *   Else, a frame-backed one is created (i.e. no copies are made).
 *
 * @return A pointer to the new caterva container
 */
caterva_array_t *caterva_from_frame(caterva_ctx_t *ctx, blosc2_frame *frame, bool copy);


/**
 * @brief Create a caterva container from a serialized frame
 *
 * @param ctx Pointer to the caterva context to be used.
 *   The context should be the same as the one used to create the array.
 * @param sframe The serialized frame for the caterva container.
 * @param copy If true, a new, sparse in-memory super-chunk is created.
 *   Else, a frame-backed one is created (i.e. no copies are made).
 *
 * @return A pointer to the new caterva container
 */
caterva_array_t *caterva_from_sframe(caterva_ctx_t *ctx, uint8_t *sframe,
int64_t len, bool copy);


/**
 * @brief Read a caterva container from disk
 *
 * @param ctx Pointer to the caterva context to be used.
 *   The context should be the same as the one used to create the array.
 * @param filename The filename of the caterva container on disk.
 * @param copy If true, a new, sparse in-memory super-chunk is created.
 *   Else, a frame-backed one is created (i.e. no copies are made).
 *
 * @return A pointer to the new caterva container
 */
```

```
caterva_array_t *caterva_from_file(caterva_ctx_t *ctx, const char *filename, bool copy);


/**
 * @brief Create a caterva container from the data obtained in a C buffer
 *
 * @param dest Pointer to the container that will be created with the buffer data
 * @param shape The shape of the buffer data
 * @param src A pointer to the C buffer where data is stored
 *
 * @return An error code
 */
int caterva_from_buffer(caterva_array_t *dest, caterva_dims_t *shape, const void *src);

int caterva_from_buffer_2(caterva_array_t *dest, caterva_dims_t *shape, const void *src);


/**
 * @brief Extract the data into a C buffer from a caterva container
 *
 * @param src Pointer to the container from which the data will be obtained
 * @param dest Pointer to the buffer where data will be stored
 *
 * @return An error code
 */
int caterva_to_buffer(caterva_array_t *src, void *dest);

int caterva_to_buffer_2(caterva_array_t *src, void *dest);


/**
 * @brief Get a slice into an empty caterva container from another caterva container
 *
```

```
 * @param dest Pointer to the empty container where the obtained slice will be stored
 * @param src Pointer to the container from which the slice will be obtained
 * @param start The coordinates where the slice will begin
 * @param stop The coordinates where the slice will end
 *
 * @return An error code
 */
int caterva_get_slice(caterva_array_t *dest, caterva_array_t *src, caterva_dims_t *start,
caterva_dims_t *stop);

int caterva_get_slice_2(caterva_array_t *dest, caterva_array_t *src,
caterva_dims_t *start, caterva_dims_t *stop);


/**
 * @brief Repartition a caterva container
 *
 * It can only be used if the container is based on a blosc superchunk since
 * it is the only one
 * that has the concept of partition.
 *
 * @param dest Pointer to the empty container with the new partition shape.
 * @param src Pointer to the container to be repartitioned.
 *
 * @return An error code
 */
int caterva_repart(caterva_array_t *dest, caterva_array_t *src);


/**
 * @brief Squeeze a caterva container
 *
 * This function remove single-dimensional entries from the shape of a caterva container.
```

```
 *
 * @param src Pointer to the container to be squeezed
 *
 * @return An error code
 */
int caterva_squeeze(caterva_array_t *src);

int caterva_squeeze_2(caterva_array_t *src);


/**
 * @brief Get a slice into a C buffer from a caterva container
 *
 * @param dest Pointer to the buffer where data will be stored
 * @param src Pointer to the container from which the slice will be extracted
 * @param start The coordinates where the slice will begin
 * @param stop The coordinates where the slice will end
 * @param d_pshape The partition shape of the buffer
 * @return An error code
 */
int caterva_get_slice_buffer(void *dest, caterva_array_t *src, caterva_dims_t *start,
                             caterva_dims_t *stop, caterva_dims_t *d_pshape);
/**
 * @brief Get a slice into a C buffer from a caterva container (with subpartitions)
 *
 * @param dest Pointer to the buffer where data will be stored
 * @param src Pointer to the container from which the slice will be extracted
 * @param start The coordinates where the slice will begin
 * @param stop The coordinates where the slice will end
 * @param d_pshape The partition shape of the buffer
 * @return An error code
 */
int caterva_get_slice_buffer_2(void *dest, caterva_array_t *src, caterva_dims_t *start,
```

```
                            caterva_dims_t *stop, caterva_dims_t *d_pshape);


/**
 * @brief Get a slice (without copy) into a C pointer from a caterva container
 *
 * With this function the data is not copied, thus allowing a higher speed.
 *
 * It can only be used if the container is based on a buffer. Also, the slice
 * obtained should
 * not be modified since it is a reference to the container data. If the slice
 * is modified, the container will also be modified.
 *
 * @param dest Pointer to the C pointer where pointer data will be referenced
 * @param src Pointer to the container from which the slice will be extracted
 * @param start The coordinates where the slice will begin
 * @param stop The coordinates where the slice will end
 * @param d_pshape The partition shape of the buffer
 *
 * @return An error code
 */
int caterva_get_slice_buffer_no_copy(void **dest, caterva_array_t *src,
         caterva_dims_t *start, caterva_dims_t *stop, caterva_dims_t *d_pshape);


/**
 * @brief Set a slice into a caterva container from a C buffer
 *
 * It can only be used if the container is based on a buffer.
 *
 * @param dest Pointer to the caterva container where the partition will be set
 * @param src Pointer to the buffer where the slice data is
 * @param start The coordinates where the slice will begin
```

```
 * @param stop The coordinates where the slice will end
 *
 * @return An error code
 */
int caterva_set_slice_buffer(caterva_array_t *dest, void *src, caterva_dims_t *start,
                             caterva_dims_t *stop);


/**
 * @brief Update the shape of a caterva container.
 *
 * This is used when data is added to the container to update the shape
 *
 * @param src Pointer to the container from which the shape will be updated
 * @param shape The new shape of the container
 *
 * @return An error code
 */
int caterva_update_shape(caterva_array_t *src, caterva_dims_t *shape);


/**
 * @brief Get the shape of a caterva array
 *
 * @param src pointer to the container from which the partition shape will be obtained
 *
 * @return The partition shape of the caterva array
 */
caterva_dims_t caterva_get_shape(caterva_array_t *src);


/**
 * @brief Get the partition shape of a caterva array
```

```
 *
 * @param src pointer to the container from which the partition shape will be obtained
 *
 * @return The partition shape of the caterva array
 */
caterva_dims_t caterva_get_pshape(caterva_array_t *src);


/**
 * @brief Get the subpartition shape of a caterva array
 *
 * @param src pointer to the container from which the partition shape will be obtained
 *
 * @return The subpartition shape of the caterva array
 */
caterva_dims_t caterva_get_spshape(caterva_array_t *src);


/**
 * @brief Make a copy of the container data.
 *
 * The copy is done into \p dest container
 *
 * @param dest Pointer to the container where data is copied
 * @param src Pointer to the container from which data is copied
 *
 * @return An error code
 */
int caterva_copy(caterva_array_t *dest, caterva_array_t *src);


#endif
```

A.2. Caterva.c

En este archivo se encuentra el código de las funciones y estructuras listadas en caterva.h:

```
caterva_ctx_t *caterva_new_ctx(void *(*c_alloc)(size_t), void (*c_free)(void *),
blosc2_cparams cparams, blosc2_dparams dparams) {
    caterva_ctx_t *ctx;
    ctx = (caterva_ctx_t *) malloc(sizeof(caterva_ctx_t));
    if (c_alloc == NULL) {
        ctx->alloc = malloc;
    } else {
        ctx->alloc = c_alloc;
    }
    if (c_free == NULL) {
        ctx->free = free;
    } else {
        ctx->free = c_free;
    }
    ctx->cparams = cparams;
    ctx->dparams = dparams;
    return ctx;
}


caterva_dims_t caterva_new_dims(const int64_t *dims, int8_t ndim) {
    caterva_dims_t dims_s = CATERVA_DIMS_DEFAULTS;
    for (int i = 0; i < ndim; ++i) {
        dims_s.dims[i] = dims[i];
    }
```

```
    dims_s.ndim = ndim;
    return dims_s;
}


static int32_t serialize_meta(int8_t ndim, int64_t *shape, const int32_t *pshape,
uint8_t **smeta) {
    // Allocate space for Caterva metalayer
    int32_t max_smeta_len = 1 + 1 + 1 + (1 + ndim * (1 + sizeof(int64_t))) + \
        (1 + ndim * (1 + sizeof(int32_t))) + (1 + ndim * (1 + sizeof(int32_t)));
    *smeta = malloc((size_t)max_smeta_len);
    uint8_t *pmeta = *smeta;

    // Build an array with 5 entries (version, ndim, shape, pshape, bshape)
    *pmeta++ = 0x90 + 5;

    // version entry
    *pmeta++ = CATERVA_METALAYER_VERSION;  // positive fixnum (7-bit positive integer)
    assert(pmeta - *smeta < max_smeta_len);

    // ndim entry
    *pmeta++ = (uint8_t)ndim;  // positive fixnum (7-bit positive integer)
    assert(pmeta - *smeta < max_smeta_len);

    // shape entry
    *pmeta++ = (uint8_t)(0x90) + ndim;  // fix array with ndim elements
    for (int8_t i = 0; i < ndim; i++) {
        *pmeta++ = 0xd3;  // int64
        swap_store(pmeta, shape + i, sizeof(int64_t));
        pmeta += sizeof(int64_t);
    }
    assert(pmeta - *smeta < max_smeta_len);
```

```
    // pshape entry
    *pmeta++ = (uint8_t)(0x90) + ndim;  // fix array with ndim elements
    for (int8_t i = 0; i < ndim; i++) {
        *pmeta++ = 0xd2;  // int32
        swap_store(pmeta, pshape + i, sizeof(int32_t));
        pmeta += sizeof(int32_t);
    }
    assert(pmeta - *smeta <= max_smeta_len);

    // bshape entry
    *pmeta++ = (uint8_t)(0x90) + ndim;  // fix array with ndim elements
    int32_t *bshape = malloc(CATERVA_MAXDIM * sizeof(int32_t));
    for (int8_t i = 0; i < ndim; i++) {
        *pmeta++ = 0xd2;  // int32
        bshape[i] = 0;  // FIXME: update when support for multidimensional bshapes
        would be ready
        // NOTE: we need to initialize the header so as to avoid false negatives
        in valgrind
        swap_store(pmeta, bshape + i, sizeof(int32_t));
        pmeta += sizeof(int32_t);
    }
    free(bshape);
    assert(pmeta - *smeta <= max_smeta_len);
    int32_t slen = (int32_t)(pmeta - *smeta);

    return slen;
}


static int32_t deserialize_meta(uint8_t *smeta, uint32_t smeta_len,
caterva_dims_t *shape, caterva_dims_t *pshape) {
    uint8_t *pmeta = smeta;
```

```
// Check that we have an array with 5 entries (version, ndim, shape, pshape, bshape)
assert(*pmeta == 0x90 + 5);
pmeta += 1;
assert(pmeta - smeta < smeta_len);

// version entry
int8_t version = pmeta[0];  // positive fixnum (7-bit positive integer)
assert (version <= CATERVA_METALAYER_VERSION);
pmeta += 1;
assert(pmeta - smeta < smeta_len);

// ndim entry
int8_t ndim = pmeta[0];  // positive fixnum (7-bit positive integer)
assert (ndim <= CATERVA_MAXDIM);
pmeta += 1;
assert(pmeta - smeta < smeta_len);
shape->ndim = ndim;
pshape->ndim = ndim;

// shape entry
// Initialize to ones, as required by Caterva
for (int i = 0; i < CATERVA_MAXDIM; i++) shape->dims[i] = 1;
assert(*pmeta == (uint8_t)(0x90) + ndim);  // fix array with ndim elements
pmeta += 1;
for (int8_t i = 0; i < ndim; i++) {
    assert(*pmeta == 0xd3);   // int64
    pmeta += 1;
    swap_store(shape->dims + i, pmeta, sizeof(int64_t));
    pmeta += sizeof(int64_t);
}
assert(pmeta - smeta < smeta_len);

// pshape entry
```

```
    // Initialize to ones, as required by Caterva
    for (int i = 0; i < CATERVA_MAXDIM; i++) pshape->dims[i] = 1;
    assert(*pmeta == (uint8_t)(0x90) + ndim);  // fix array with ndim elements
    pmeta += 1;
    for (int8_t i = 0; i < ndim; i++) {
        assert(*pmeta == 0xd2);  // int32
        pmeta += 1;
        swap_store(pshape->dims + i, pmeta, sizeof(int32_t));
        pmeta += sizeof(int32_t);
    }
    assert(pmeta - smeta <= smeta_len);

    // bshape entry
    // Initialize to ones, as required by Caterva
    // for (int i = 0; i < CATERVA_MAXDIM; i++) bshape->dims[i] = 1;
    assert(*pmeta == (uint8_t)(0x90) + ndim);  // fix array with ndim elements
    pmeta += 1;
    for (int8_t i = 0; i < ndim; i++) {
        assert(*pmeta == 0xd2);  // int32
        pmeta += 1;
        // swap_store(bshape->dims + i, pmeta, sizeof(int32_t));
        pmeta += sizeof(int32_t);
    }
    assert(pmeta - smeta <= smeta_len);
    uint32_t slen = (uint32_t)(pmeta - smeta);
    assert(slen == smeta_len);
    return 0;
}


caterva_array_t *caterva_empty_array(caterva_ctx_t *ctx, blosc2_frame *frame,
caterva_dims_t *pshape) {
    /* Create a caterva_array_t buffer */
```

```
caterva_array_t *carr = (caterva_array_t *) ctx->alloc(sizeof(caterva_array_t));
carr->size = 1;
carr->psize = 1;
carr->esize = 1;
// The partition cache (empty initially)
carr->part_cache.data = NULL;
carr->part_cache.nchunk = -1;  // means no valid cache yet
carr->sc = NULL;
carr->buf = NULL;

if (pshape != NULL) {
    carr->storage = CATERVA_STORAGE_BLOSC;
    carr->ndim = pshape->ndim;
    for (unsigned int i = 0; i < CATERVA_MAXDIM; i++) {
        carr->pshape[i] = (int32_t)(pshape->dims[i]);
        carr->shape[i] = 1;
        carr->eshape[i] = 1;
        carr->psize *= carr->pshape[i];
    }

    blosc2_schunk *sc = blosc2_new_schunk(ctx->cparams, ctx->dparams, frame);
    if (frame != NULL) {
        // Serialize the dimension info in the associated frame
        if (sc->nmetalayers >= BLOSC2_MAX_METALAYERS) {
            fprintf(stderr, "the number of metalayers for this frame has been exceeded\n
            return NULL;
        }
        uint8_t *smeta = NULL;
        int32_t smeta_len = serialize_meta(carr->ndim, carr->shape, carr->pshape, &smeta
        if (smeta_len < 0) {
            fprintf(stderr, "error during serializing dims info for Caterva");
            return NULL;
        }
```

```
            // And store it in caterva metalayer
            int retcode = blosc2_add_metalayer(sc, "caterva", smeta, (uint32_t)smeta_len);
            if (retcode < 0) {
                return NULL;
            }
            free(smeta);
        }
        /* Create a schunk (for a frame-disk-backed one, this implies serializing the heade
        carr->sc = sc;
    } else {
        carr->storage = CATERVA_STORAGE_PLAINBUFFER;
    }

    /* Copy context to caterva_array_t */
    carr->ctx = (caterva_ctx_t *) ctx->alloc(sizeof(caterva_ctx_t));
    memcpy(&carr->ctx[0], &ctx[0], sizeof(caterva_ctx_t));

    carr->empty = true;
    carr->filled = false;
    carr->nparts = 0;
    return carr;
}


caterva_array_t *caterva_empty_array_2(caterva_ctx_t *ctx, blosc2_frame *frame,
caterva_dims_t *pshape, caterva_dims_t *spshape) {
    /* Create a caterva_array_t buffer */
    caterva_array_t *carr = (caterva_array_t *) ctx->alloc(sizeof(caterva_array_t));
    carr->size = 1;
    carr->psize = 1;
    carr->esize = 1;
    carr->spsize = 1;
    carr->epsize = 1;
```

```
// The partition cache (empty initially)
carr->part_cache.data = NULL;
carr->part_cache.nchunk = -1;  // means no valid cache yet
carr->sc = NULL;
carr->buf = NULL;

if (pshape != NULL) {
    carr->storage = CATERVA_STORAGE_BLOSC;
    carr->ndim = pshape->ndim;
    for (unsigned int i = 0; i < CATERVA_MAXDIM; i++) {
        carr->pshape[i] = (int32_t)(pshape->dims[i]);
        carr->next_pshape[i] = carr->pshape[i];
        carr->spshape[i] = (int32_t)(spshape->dims[i]);
        carr->shape[i] = 1;
        carr->eshape[i] = 1;
        carr->psize *= carr->pshape[i];
        carr->spsize *= carr->spshape[i];

        if (i < carr->ndim) {
            if (carr->pshape[i] % carr->spshape[i] == 0) {
                carr->epshape[i] = carr->pshape[i];
            } else {
                carr->epshape[i] = carr->pshape[i] + carr->spshape[i]
                - carr->pshape[i] % carr->spshape[i];
            }
        } else {
            carr->epshape[i] = 1;
        }
        carr->epsize *= carr->epshape[i];
    }

    blosc2_schunk *sc = blosc2_new_schunk(ctx->cparams, ctx->dparams, frame);
```

```
    if (frame != NULL) {
        // Serialize the dimension info in the associated frame
        if (sc->nmetalayers >= BLOSC2_MAX_METALAYERS) {
            fprintf(stderr, "the number of metalayers for this frame has been
            exceeded\n");
            return NULL;
        }
        uint8_t *smeta = NULL;
        int32_t smeta_len = serialize_meta(carr->ndim, carr->shape, carr->pshape,
        &smeta);
        if (smeta_len < 0) {
            fprintf(stderr, "error during serializing dims info for Caterva");
            return NULL;
        }
        // And store it in caterva metalayer
        int retcode = blosc2_add_metalayer(sc, "caterva", smeta,
        (uint32_t)smeta_len);
        if (retcode < 0) {
            return NULL;
        }
        free(smeta);
    }
    /* Create a schunk (for a frame-disk-backed one, this implies serializing
    the header on-disk */
    carr->sc = sc;
} else {
    carr->storage = CATERVA_STORAGE_PLAINBUFFER;
}

/* Copy context to caterva_array_t */
carr->ctx = (caterva_ctx_t *) ctx->alloc(sizeof(caterva_ctx_t));
memcpy(&carr->ctx[0], &ctx[0], sizeof(caterva_ctx_t));
```

```
    carr->empty = true;
    carr->filled = false;
    carr->nparts = 0;
    carr->next_size = carr->psize;
    return carr;
}


caterva_array_t *caterva_from_frame(caterva_ctx_t *ctx, blosc2_frame *frame, bool copy) {
    /* Create a caterva_array_t buffer */
    caterva_array_t *carr = (caterva_array_t *) ctx->alloc(sizeof(caterva_array_t));

    /* Copy context to caterva_array_t */
    carr->ctx = (caterva_ctx_t *) ctx->alloc(sizeof(caterva_ctx_t));
    memcpy(&carr->ctx[0], &ctx[0], sizeof(caterva_ctx_t));

    /* Create a schunk out of the frame */
    blosc2_schunk *sc = blosc2_schunk_from_frame(frame, copy);
    carr->sc = sc;
    carr->storage = CATERVA_STORAGE_BLOSC;

    blosc2_dparams *dparams;
    blosc2_schunk_get_dparams(carr->sc, &dparams);
    blosc2_cparams *cparams;
    blosc2_schunk_get_cparams(carr->sc, &cparams);
    memcpy(&carr->ctx->dparams, dparams, sizeof(blosc2_dparams));
    memcpy(&carr->ctx->cparams, cparams, sizeof(blosc2_cparams));

    // Deserialize the caterva metalayer
    caterva_dims_t shape;
    caterva_dims_t pshape;
    uint8_t *smeta;
    uint32_t smeta_len;
```

```
blosc2_get_metalayer(sc, "caterva", &smeta, &smeta_len);
deserialize_meta(smeta, smeta_len, &shape, &pshape);
carr->size = 1;
carr->psize = 1;
carr->esize = 1;
carr->ndim = pshape.ndim;

for (int i = 0; i < CATERVA_MAXDIM; i++) {
    carr->shape[i] = shape.dims[i];
    carr->size *= shape.dims[i];
    carr->pshape[i] = (int32_t)(pshape.dims[i]);
    carr->psize *= carr->pshape[i];
    if (shape.dims[i] % pshape.dims[i] == 0) {
        // The case for shape.dims[i] == 1 and pshape.dims[i] == 1 is handled here
        carr->eshape[i] = shape.dims[i];
    } else {
        carr->eshape[i] = shape.dims[i] + pshape.dims[i] - shape.dims[i]
        % pshape.dims[i];
    }
    carr->esize *= carr->eshape[i];
}

// The partition cache (empty initially)
carr->part_cache.data = NULL;
carr->part_cache.nchunk = -1;  // means no valid cache yet
carr->empty = false;
if (carr->sc->nchunks == carr->esize / carr->psize) {
    carr->filled = true;
} else {
    carr->filled = false;
}

return carr;
```

```
}


caterva_array_t *caterva_from_sframe(caterva_ctx_t *ctx, uint8_t *sframe, int64_t
len, bool copy) {
    // Generate a real frame first
    blosc2_frame *frame = blosc2_frame_from_sframe(sframe, len, copy);
    // ...and create a caterva array out of it
    caterva_array_t *array = caterva_from_frame(ctx, frame, copy);
    if (copy) {
        // We don't need the frame anymore
        blosc2_free_frame(frame);
    }
    return array;
}


caterva_array_t *caterva_from_file(caterva_ctx_t *ctx, const char *filename, bool copy) {
    // Open the frame on-disk...
    blosc2_frame *frame = blosc2_frame_from_file(filename);
    // ...and create a caterva array out of it
    caterva_array_t *array = caterva_from_frame(ctx, frame, copy);
    if (copy) {
        // We don't need the frame anymore
        blosc2_free_frame(frame);
    }
    return array;
}


int caterva_free_ctx(caterva_ctx_t *ctx) {
    free(ctx);
    return 0;
```

```
}


int caterva_free_array(caterva_array_t *carr) {
    switch (carr->storage) {
        case CATERVA_STORAGE_BLOSC:
            if (carr->sc != NULL) {
                blosc2_free_schunk(carr->sc);
            }
            break;
        case CATERVA_STORAGE_PLAINBUFFER:
            if (carr->buf != NULL) {
                carr->ctx->free(carr->buf);
            }
    }
    void (*aux_free)(void *) = carr->ctx->free;
    caterva_free_ctx(carr->ctx);
    aux_free(carr);
    return 0;
}


int caterva_update_shape(caterva_array_t *carr, caterva_dims_t *shape) {
    carr->empty = false;
    if (carr->storage == CATERVA_STORAGE_BLOSC) {
        if (carr->ndim != shape->ndim) {
            printf("caterva array ndim and shape ndim are not equal\n");
            return -1;
        }
        carr->size = 1;
        carr->esize = 1;
        for (int i = 0; i < CATERVA_MAXDIM; ++i) {
            carr->shape[i] = shape->dims[i];
```

```
    if (i < shape->ndim) {
        if (shape->dims[i] % carr->pshape[i] == 0) {
            carr->eshape[i] = shape->dims[i];
        } else {
            carr->eshape[i] = shape->dims[i] + carr->pshape[i]
            - shape->dims[i] % carr->pshape[i];
        }
    } else {
        carr->eshape[i] = 1;
    }
    carr->size *= carr->shape[i];
    carr->esize *= carr->eshape[i];
}

uint8_t *smeta = NULL;
// Serialize the dimension info ...
int32_t smeta_len = serialize_meta(carr->ndim, carr->shape, carr->pshape, &smeta);
if (smeta_len < 0) {
    fprintf(stderr, "error during serializing dims info for Caterva");
    return -1;
}
// ... and update it in its metalayer
if (blosc2_has_metalayer(carr->sc, "caterva") < 0) {
    int retcode = blosc2_add_metalayer(carr->sc, "caterva", smeta,
    (uint32_t) smeta_len);
    if (retcode < 0) {
        return -1;
    }
}
else {
    int retcode = blosc2_update_metalayer(carr->sc, "caterva", smeta,
    (uint32_t) smeta_len);
    if (retcode < 0) {
```

```
                return -1;
            }
        }
    } else {
        carr->ndim = shape->ndim;
        carr->size = 1;
        carr->esize = 1;
        for (int i = 0; i < CATERVA_MAXDIM; ++i) {
            carr->shape[i] = shape->dims[i];
            carr->eshape[i] = shape->dims[i];
            carr->pshape[i] = (int32_t)(shape->dims[i]);
            carr->epshape[i] = shape->dims[i];
            carr->spshape[i] = (int32_t)(shape->dims[i]);
            carr->size *= carr->shape[i];
            carr->esize *= carr->eshape[i];
            carr->psize *= carr->pshape[i];
            carr->epsize *= carr->eshape[i];
            carr->spsize *= carr->pshape[i];
        }
    }

    return 0;
}


int caterva_repart_chunk(int8_t *chunk, int size_chunk, void *src, int size_src,
caterva_array_t *carr, caterva_ctx_t *ctx){
    if(ctx != carr->ctx) {
        return -1;
    }
    if (size_chunk != carr->epsize * carr->ctx->cparams.typesize) {
        return -2;
    }
```

```
if (size_src != carr->psize * carr->ctx->cparams.typesize) {
    return -3;
}

const int8_t *src_b = (int8_t *) src;
memset(chunk, 0, size_chunk);
int32_t d_pshape[CATERVA_MAXDIM];
int64_t d_epshape[CATERVA_MAXDIM];
int32_t d_spshape[CATERVA_MAXDIM];
int8_t d_ndim = carr->ndim;

for (int i = 0; i < CATERVA_MAXDIM; ++i) {
    d_pshape[(CATERVA_MAXDIM - d_ndim + i) % CATERVA_MAXDIM] = carr->pshape[i];
    d_epshape[(CATERVA_MAXDIM - d_ndim + i) % CATERVA_MAXDIM] = carr->epshape[i];
    d_spshape[(CATERVA_MAXDIM - d_ndim + i) % CATERVA_MAXDIM] = carr->spshape[i];
}

int64_t aux2[CATERVA_MAXDIM];
aux2[7] = d_epshape[7] / d_spshape[7];
for (int i = CATERVA_MAXDIM - 2; i >= 0; i--) {
    aux2[i] = d_epshape[i] / d_spshape[i] * aux2[i + 1];
}

/* Fill each subpartition buffer */
int64_t orig[CATERVA_MAXDIM];
int32_t actual_spsize[CATERVA_MAXDIM];
for (int64_t sci = 0; sci < carr->epsize / carr->spsize; sci++) {
    /*Calculate the coord. of the subpartition first element */
    orig[7] = sci % (d_epshape[7] / d_spshape[7]) * d_spshape[7];
    for (int i = CATERVA_MAXDIM - 2; i >= 0; i--) {
        orig[i] = sci % (aux2[i]) / (aux2[i + 1]) * d_spshape[i];
    }
    /* Calculate if padding with 0s is needed for this subpartition */
```

```
for (int i = CATERVA_MAXDIM - 1; i >= 0; i--) {
    if (orig[i] + d_spshape[i] > d_spshape[i]) {
        actual_spsize[i] = d_pshape[i] - orig[i];
    } else {
        actual_spsize[i] = d_spshape[i];
    }
}
int32_t seq_copylen = actual_spsize[7] * carr->ctx->cparams.typesize;
/* Copy each line of data from src_b to chunk */
int64_t ii[CATERVA_MAXDIM];
for (ii[6] = 0; ii[6] < actual_spsize[6]; ii[6]++) {
    for (ii[5] = 0; ii[5] < actual_spsize[5]; ii[5]++) {
        for (ii[4] = 0; ii[4] < actual_spsize[4]; ii[4]++) {
            for (ii[3] = 0; ii[3] < actual_spsize[3]; ii[3]++) {
                for (ii[2] = 0; ii[2] < actual_spsize[2]; ii[2]++) {
                    for (ii[1] = 0; ii[1] < actual_spsize[1]; ii[1]++) {
                        for (ii[0] = 0; ii[0] < actual_spsize[0]; ii[0]++) {
                            int64_t d_a = d_spshape[7];
                            int64_t d_coord_f = sci * carr->spsize;
                            for (int i = CATERVA_MAXDIM - 2; i >= 0; i--) {
                                d_coord_f += ii[i] * d_a;
                                d_a *= d_spshape[i];
                            }

                            int64_t s_coord_f = orig[7];
                            int64_t s_a = d_pshape[7];
                            for (int i = CATERVA_MAXDIM - 2; i >= 0; i--) {
                                s_coord_f += (orig[i] + ii[i]) * s_a;
                                s_a *= d_pshape[i];
                            }
                            memcpy(chunk + d_coord_f * carr->ctx->cparams.typesize,
                                   src_b + s_coord_f * carr->ctx->cparams.typesize,
                                   seq_copylen);
```

```
                                }
                            }
                        }
                    }
                }
            }
        }
    }
    return 0;
}

int caterva_derepart_chunk(int8_t *chunk, int size_chunk, int8_t *rchunk, int size_rchunk,
caterva_array_t *carr, caterva_ctx_t *ctx){
    if(ctx != carr->ctx) {
        return -1;
    }
    if (size_chunk != carr->psize * carr->ctx->cparams.typesize) {
        return -2;
    }
    if (size_rchunk != carr->epsize * carr->ctx->cparams.typesize) {
        return -3;
    }

    memset(chunk, 0, size_chunk);
    int32_t d_pshape[CATERVA_MAXDIM];
    int64_t d_epshape[CATERVA_MAXDIM];
    int32_t d_spshape[CATERVA_MAXDIM];
    int8_t d_ndim = carr->ndim;

    for (int i = 0; i < CATERVA_MAXDIM; ++i) {
        d_pshape[(CATERVA_MAXDIM - d_ndim + i) % CATERVA_MAXDIM] = carr->pshape[i];
        d_epshape[(CATERVA_MAXDIM - d_ndim + i) % CATERVA_MAXDIM] = carr->epshape[i];
        d_spshape[(CATERVA_MAXDIM - d_ndim + i) % CATERVA_MAXDIM] = carr->spshape[i];
```

```
}

int64_t aux2[CATERVA_MAXDIM];
aux2[7] = d_epshape[7] / d_spshape[7];
for (int i = CATERVA_MAXDIM - 2; i >= 0; i--) {
    aux2[i] = d_epshape[i] / d_spshape[i] * aux2[i + 1];
}

/* Fill each subpartition buffer */
int64_t orig[CATERVA_MAXDIM];
int32_t actual_spsize[CATERVA_MAXDIM];
for (int64_t sci = 0; sci < carr->epsize / carr->spsize; sci++) {
    /*Calculate the coord. of the subpartition first element */
    orig[7] = sci % (d_epshape[7] / d_spshape[7]) * d_spshape[7];
    for (int i = CATERVA_MAXDIM - 2; i >= 0; i--) {
        orig[i] = sci % (aux2[i]) / (aux2[i + 1]) * d_spshape[i];
    }
    /* Calculate if padding with 0s is needed for this subchunk */
    for (int i = CATERVA_MAXDIM - 1; i >= 0; i--) {
        if (orig[i] + d_spshape[i] > d_spshape[i]) {
            actual_spsize[i] = d_pshape[i] - orig[i];
        } else {
            actual_spsize[i] = d_spshape[i];
        }
    }
    int32_t seq_copylen = actual_spsize[7] * carr->ctx->cparams.typesize;
    /* Reorder each line of data from rchunk to chunk */
    int64_t ii[CATERVA_MAXDIM];
    for (ii[6] = 0; ii[6] < actual_spsize[6]; ii[6]++) {
        for (ii[5] = 0; ii[5] < actual_spsize[5]; ii[5]++) {
            for (ii[4] = 0; ii[4] < actual_spsize[4]; ii[4]++) {
                for (ii[3] = 0; ii[3] < actual_spsize[3]; ii[3]++) {
                    for (ii[2] = 0; ii[2] < actual_spsize[2]; ii[2]++) {
```

```
                            for (ii[1] = 0; ii[1] < actual_spsize[1]; ii[1]++) {
                                for (ii[0] = 0; ii[0] < actual_spsize[0]; ii[0]++) {
                                    int64_t d_a = d_spshape[7];
                                    int64_t d_coord_f = sci * carr->spsize;
                                    for (int i = CATERVA_MAXDIM - 2; i >= 0; i--) {
                                        d_coord_f += ii[i] * d_a;
                                        d_a *= d_spshape[i];
                                    }

                                    int64_t s_coord_f = orig[7];
                                    int64_t s_a = d_pshape[7];
                                    for (int i = CATERVA_MAXDIM - 2; i >= 0; i--) {
                                        s_coord_f += (orig[i] + ii[i]) * s_a;
                                        s_a *= d_pshape[i];
                                    }
                                    memcpy(chunk + s_coord_f * carr->ctx->cparams.typesize,
                                           rchunk + d_coord_f * carr->ctx->cparams.typesize,
                                           seq_copylen);
                                }
                            }
                        }
                    }
                }
            }
        }
    }
    return 0;
}



int caterva_append(caterva_array_t *carr, void *part, int64_t partsize) {
    if (partsize != (int64_t) carr->psize * carr->ctx->cparams.typesize) {
```

```
        return -1;
    }
    if (carr->filled) {
        return -2;
    }

    if (carr->storage == CATERVA_STORAGE_BLOSC) {
        blosc2_schunk_append_buffer(carr->sc, part, partsize);
    } else {
        if (carr->nparts == 0) {
            carr->buf = malloc(carr->size * (size_t) carr->ctx->cparams.typesize);
        }
        int64_t start_[CATERVA_MAXDIM], stop_[CATERVA_MAXDIM];
        for (int i = 0; i < carr->ndim; ++i) {
            start_[i] = 0;
            stop_[i] = start_[i] + carr->pshape[i];
        }
        caterva_dims_t start = caterva_new_dims(start_, carr->ndim);
        caterva_dims_t stop = caterva_new_dims(stop_, carr->ndim);
        caterva_set_slice_buffer(carr, part, &start, &stop);
    }
    carr->nparts++;
    if (carr->nparts == carr->esize / carr->psize) {
        carr->filled = true;
    }

    return 0;
}


int caterva_append_2(caterva_array_t *carr, void *part, int64_t partsize) {
    if (carr->filled) {
        printf("Already filled");
```

```
    return -1;
}
if (partsize != carr->next_size * carr->ctx->cparams.typesize) {
    printf("Wrong partition size: it must be %ld", carr->next_size);
    return -2;
}

bool padding = false;
int32_t size_chunk = carr->psize * carr->ctx->cparams.typesize;
if (partsize != (int64_t) size_chunk) {
    padding = true;
}
caterva_ctx_t *ctx = carr->ctx;
int64_t typesize = carr->ctx->cparams.typesize;
int64_t size_rep = carr->epsize * typesize;
int8_t *rep = ctx->alloc(size_rep);
int32_t c_pshape[CATERVA_MAXDIM];
int8_t c_ndim = carr->ndim;

if (padding) {
    uint8_t *chunk = malloc(size_chunk);
    memset(chunk, 0, size_chunk);
    int32_t next_pshape[CATERVA_MAXDIM];
    for (int i = 0; i < CATERVA_MAXDIM; ++i) {
        next_pshape[(CATERVA_MAXDIM - c_ndim + i) % CATERVA_MAXDIM] =
        carr->next_pshape[i];
        c_pshape[(CATERVA_MAXDIM - c_ndim + i) % CATERVA_MAXDIM] = carr->pshape[i];
    }
    int32_t seq_copylen = next_pshape[7] * carr->ctx->cparams.typesize;
    bool blank;
    int32_t ii[CATERVA_MAXDIM];
    int ind_src = 0; int ind_dest = 0;
    for (ii[0] = 0; ii[0] < c_pshape[0]; ii[0]++) {
```

```
        for (ii[1] = 0; ii[1] < c_pshape[1]; ii[1]++) {
            for (ii[2] = 0; ii[2] < c_pshape[2]; ii[2]++) {
                for (ii[3] = 0; ii[3] < c_pshape[3]; ii[3]++) {
                    for (ii[4] = 0; ii[4] < c_pshape[4]; ii[4]++) {
                        for (ii[5] = 0; ii[5] < c_pshape[5]; ii[5]++) {
                            for (ii[6] = 0; ii[6] < c_pshape[6]; ii[6]++) {
                                // Calculate if line is full of 0s
                                blank = false;
                                for(int i = 0; i < CATERVA_MAXDIM - 1; i++) {
                                    if (ii[i] >= next_pshape[i]) {
                                        blank = true;
                                        break;
                                    }
                                }
                                if (! blank) {
                                    memcpy(chunk + ind_dest *
                                    carr->ctx->cparams.typesize, part + ind_src *
                                    carr->ctx->cparams.typesize,
                                           seq_copylen);
                                    ind_src += next_pshape[7];
                                }
                                ind_dest += c_pshape[7];
                            }
                        }
                    }
                }
            }
        }
    }
    caterva_repart_chunk(rep, size_rep, chunk, size_chunk, carr, ctx);
    ctx->free(chunk);
} else {
    caterva_repart_chunk(rep, size_rep, part, partsize, carr, ctx);
```

```
}
if (carr->storage == CATERVA_STORAGE_BLOSC) {
    blosc2_schunk_append_buffer(carr->sc, rep, size_rep);
} else {

    if (carr->nparts == 0) {
        carr->buf = malloc(carr->size * (size_t) carr->ctx->cparams.typesize);
    }
    int64_t start_[CATERVA_MAXDIM], stop_[CATERVA_MAXDIM];
    for (int i = 0; i < carr->ndim; ++i) {
        start_[i] = 0;
        stop_[i] = start_[i] + carr->pshape[i];
    }
    caterva_dims_t start = caterva_new_dims(start_, carr->ndim);
    caterva_dims_t stop = caterva_new_dims(stop_, carr->ndim);
    caterva_set_slice_buffer(carr, part, &start, &stop);

}
carr->nparts++;
if (carr->nparts == carr->esize / carr->psize) {
    carr->filled = true;
}
ctx->free(rep);
// Calculate chunk position in each dimension
int64_t c_shape[CATERVA_MAXDIM];
int64_t c_eshape[CATERVA_MAXDIM];
for (int i = 0; i < CATERVA_MAXDIM; ++i) {
    c_shape[(CATERVA_MAXDIM - c_ndim + i) % CATERVA_MAXDIM] = carr->shape[i];
    c_eshape[(CATERVA_MAXDIM - c_ndim + i) % CATERVA_MAXDIM] = carr->eshape[i];
    c_pshape[(CATERVA_MAXDIM - c_ndim + i) % CATERVA_MAXDIM] = carr->pshape[i];
}

int64_t aux[CATERVA_MAXDIM];
```

```
    int64_t poschunk[CATERVA_MAXDIM];
    aux[7] = c_eshape[7] / c_pshape[7];
    for (int i = CATERVA_MAXDIM - 2; i >= 0; i--) {
        aux[i] = c_eshape[i] / c_pshape[i] * aux[i + 1];
    }
    poschunk[7] = carr->nparts % aux[7];
    for (int i = CATERVA_MAXDIM - 2; i >= 0; i--) {
        poschunk[i] = (carr->nparts % aux[i]) / aux[i + 1];
    }

    // Update next_pshape, next_size
    carr->next_size = 1;
    int64_t n_pshape[CATERVA_MAXDIM];
    for (int i = 0; i < CATERVA_MAXDIM; ++i) {
        n_pshape[i] = c_pshape[i];
        if ((poschunk[i] >= (c_eshape[i]/c_pshape[i])-1) && (c_eshape[i] > c_shape[i])) {
            n_pshape[i] -= c_eshape[i] - c_shape[i];
        }
        carr->next_size *= n_pshape[i];
    }
    for (int i = 0; i < CATERVA_MAXDIM; ++i) {
        carr->next_pshape[i] = n_pshape[(CATERVA_MAXDIM - c_ndim + i) % CATERVA_MAXDIM];
    }

    return 0;
}


int caterva_from_buffer(caterva_array_t *dest, caterva_dims_t *shape, const void *src) {
    const int8_t *src_b = (int8_t *) src;
    caterva_update_shape(dest, shape);

    if (dest->storage == CATERVA_STORAGE_BLOSC) {
```

```
if (dest->sc->nbytes > 0) {
    printf("Caterva container must be empty!");
    return -1;
}
int64_t d_shape[CATERVA_MAXDIM];
int64_t d_eshape[CATERVA_MAXDIM];
int32_t d_pshape[CATERVA_MAXDIM];
int8_t d_ndim = dest->ndim;

for (int i = 0; i < CATERVA_MAXDIM; ++i) {
    d_shape[(CATERVA_MAXDIM - d_ndim + i) % CATERVA_MAXDIM] = dest->shape[i];
    d_eshape[(CATERVA_MAXDIM - d_ndim + i) % CATERVA_MAXDIM] = dest->eshape[i];
    d_pshape[(CATERVA_MAXDIM - d_ndim + i) % CATERVA_MAXDIM] = dest->pshape[i];
}

caterva_ctx_t *ctx = dest->ctx;
int32_t typesize = dest->sc->typesize;
int8_t *chunk = ctx->alloc((size_t) dest->psize * typesize);

/* Calculate the constants out of the for  */
int64_t aux[CATERVA_MAXDIM];
aux[7] = d_eshape[7] / d_pshape[7];
for (int i = CATERVA_MAXDIM - 2; i >= 0; i--) {
    aux[i] = d_eshape[i] / d_pshape[i] * aux[i + 1];
}

/* Fill each chunk buffer */
int64_t desp[CATERVA_MAXDIM];
int32_t actual_psize[CATERVA_MAXDIM];
for (int64_t ci = 0; ci < dest->esize / dest->psize; ci++) {
    memset(chunk, 0, dest->psize * typesize);
    /* Calculate the coord. of the chunk first element */
    desp[7] = ci % (d_eshape[7] / d_pshape[7]) * d_pshape[7];
```

```
for (int i = CATERVA_MAXDIM - 2; i >= 0; i--) {
    desp[i] = ci % (aux[i]) / (aux[i + 1]) * d_pshape[i];
}
/* Calculate if padding with 0s is needed for this chunk */
for (int i = CATERVA_MAXDIM - 1; i >= 0; i--) {
    if (desp[i] + d_pshape[i] > d_shape[i]) {
        actual_psize[i] = d_shape[i] - desp[i];
    } else {
        actual_psize[i] = d_pshape[i];
    }
}
int32_t seq_copylen = actual_psize[7] * typesize;
/* Copy each line of data from chunk to arr */
int64_t ii[CATERVA_MAXDIM];
for (ii[6] = 0; ii[6] < actual_psize[6]; ii[6]++) {
    for (ii[5] = 0; ii[5] < actual_psize[5]; ii[5]++) {
        for (ii[4] = 0; ii[4] < actual_psize[4]; ii[4]++) {
            for (ii[3] = 0; ii[3] < actual_psize[3]; ii[3]++) {
                for (ii[2] = 0; ii[2] < actual_psize[2]; ii[2]++) {
                    for (ii[1] = 0; ii[1] < actual_psize[1]; ii[1]++) {
                        for (ii[0] = 0; ii[0] < actual_psize[0]; ii[0]++) {
                            int64_t d_a = d_pshape[7];
                            int64_t d_coord_f = 0;
                            for (int i = CATERVA_MAXDIM - 2; i >= 0; i--) {
                                d_coord_f += ii[i] * d_a;
                                d_a *= d_pshape[i];
                            }
                            int64_t s_coord_f = desp[7];
                            int64_t s_a = d_shape[7];
                            for (int i = CATERVA_MAXDIM - 2; i >= 0; i--) {
                                s_coord_f += (desp[i] + ii[i]) * s_a;
                                s_a *= d_shape[i];
                            }
```

```
                                        memcpy(chunk + d_coord_f * typesize, src_b +
                                        s_coord_f * typesize, seq_copylen);
                                    }
                                }
                            }
                        }
                    }
                }
                blosc2_schunk_append_buffer(dest->sc, chunk, (size_t) dest->psize * typesize);
            }
            ctx->free(chunk);
        } else {
            // Plain buffer
            if (dest->buf != NULL) {
                printf("Caterva container must be empty!");
                return -1;
            }
            dest->buf = malloc(dest->size * (size_t) dest->ctx->cparams.typesize);
            memcpy(dest->buf, src, dest->size * (size_t) dest->ctx->cparams.typesize);
        }
        dest->filled = true;
        return 0;
}

int caterva_from_buffer_2(caterva_array_t *dest, caterva_dims_t *shape, const void *src) {

    const int8_t *src_b = (int8_t *) src;
    caterva_update_shape(dest, shape);

    if (dest->storage == CATERVA_STORAGE_BLOSC) {
        if (dest->sc->nbytes > 0) {
            printf("Caterva container must be empty!");
```

```
    return -1;
}
int64_t d_shape[CATERVA_MAXDIM];
int64_t d_eshape[CATERVA_MAXDIM];
int32_t d_pshape[CATERVA_MAXDIM];
int8_t d_ndim = dest->ndim;

for (int i = 0; i < CATERVA_MAXDIM; ++i) {
    d_shape[(CATERVA_MAXDIM - d_ndim + i) % CATERVA_MAXDIM] = dest->shape[i];
    d_eshape[(CATERVA_MAXDIM - d_ndim + i) % CATERVA_MAXDIM] = dest->eshape[i];
    d_pshape[(CATERVA_MAXDIM - d_ndim + i) % CATERVA_MAXDIM] = dest->pshape[i];
}

caterva_ctx_t *ctx = dest->ctx;
int32_t typesize = dest->sc->typesize;
int8_t *chunk = ctx->alloc((size_t) dest->psize * typesize);
int8_t *rchunk = ctx->alloc((size_t) dest->epsize * typesize);

/* Calculate the constants out of the for  */
int64_t aux[CATERVA_MAXDIM];
aux[7] = d_eshape[7] / d_pshape[7];
for (int i = CATERVA_MAXDIM - 2; i >= 0; i--) {
    aux[i] = d_eshape[i] / d_pshape[i] * aux[i + 1];
}

/* Fill each chunk buffer */
int64_t desp[CATERVA_MAXDIM];
int32_t actual_psize[CATERVA_MAXDIM];
for (int64_t ci = 0; ci < dest->esize / dest->psize; ci++) {
    memset(chunk, 0, dest->psize * typesize);
    memset(rchunk, 0, dest->epsize * typesize);

    /* Calculate the coord. of the chunk first element */
```

```
desp[7] = ci % (d_eshape[7] / d_pshape[7]) * d_pshape[7];
for (int i = CATERVA_MAXDIM - 2; i >= 0; i--) {
    desp[i] = ci % (aux[i]) / (aux[i + 1]) * d_pshape[i];
}
/* Calculate if padding with 0s is needed for this chunk */
for (int i = CATERVA_MAXDIM - 1; i >= 0; i--) {
    if (desp[i] + d_pshape[i] > d_shape[i]) {
        actual_psize[i] = d_shape[i] - desp[i];
    } else {
        actual_psize[i] = d_pshape[i];
    }
}
int32_t seq_copylen = actual_psize[7] * typesize;
/* Copy each line of data from src_b to chunk */
int64_t ii[CATERVA_MAXDIM];
for (ii[6] = 0; ii[6] < actual_psize[6]; ii[6]++) {
    for (ii[5] = 0; ii[5] < actual_psize[5]; ii[5]++) {
        for (ii[4] = 0; ii[4] < actual_psize[4]; ii[4]++) {
            for (ii[3] = 0; ii[3] < actual_psize[3]; ii[3]++) {
                for (ii[2] = 0; ii[2] < actual_psize[2]; ii[2]++) {
                    for (ii[1] = 0; ii[1] < actual_psize[1]; ii[1]++) {
                        for (ii[0] = 0; ii[0] < actual_psize[0]; ii[0]++) {
                            int64_t d_a = d_pshape[7];
                            int64_t d_coord_f = 0;
                            for (int i = CATERVA_MAXDIM - 2; i >= 0; i--) {
                                d_coord_f += ii[i] * d_a;
                                d_a *= d_pshape[i];
                            }
                            int64_t s_coord_f = desp[7];
                            int64_t s_a = d_shape[7];
                            for (int i = CATERVA_MAXDIM - 2; i >= 0; i--) {
                                s_coord_f += (desp[i] + ii[i]) * s_a;
                                s_a *= d_shape[i];
```

```
                                        }
                                        memcpy(chunk + d_coord_f * typesize, src_b +
                                        s_coord_f * typesize, seq_copylen);
                                    }
                                }
                            }
                        }
                    }
                }
                // Copy each chunk from rchunk to dest
                caterva_repart_chunk(rchunk, (int) dest->epsize * typesize, chunk,
                (int) dest->psize * typesize, dest, ctx);
                blosc2_schunk_append_buffer(dest->sc, rchunk, (size_t) dest->epsize *
                typesize);
            }
            ctx->free(chunk);
            ctx->free(rchunk);
        } else {
            // Plain buffer
            if (dest->buf != NULL) {
                printf("Caterva container must be empty!");
                return -1;
            }
            dest->buf = malloc(dest->size * (size_t) dest->ctx->cparams.typesize);
            memcpy(dest->buf, src, dest->size * (size_t) dest->ctx->cparams.typesize);
        }
        dest->filled = true;
        return 0;
    }

    int caterva_to_buffer(caterva_array_t *src, void *dest) {
        if (src->storage == CATERVA_STORAGE_BLOSC) {
```

```
int8_t *d_b = (int8_t *) dest;
int64_t s_shape[CATERVA_MAXDIM];
int64_t s_pshape[CATERVA_MAXDIM];
int64_t s_eshape[CATERVA_MAXDIM];
int8_t s_ndim = src->ndim;

for (int i = 0; i < CATERVA_MAXDIM; ++i) {
    s_shape[(CATERVA_MAXDIM - s_ndim + i) % CATERVA_MAXDIM] = src->shape[i];
    s_eshape[(CATERVA_MAXDIM - s_ndim + i) % CATERVA_MAXDIM] = src->eshape[i];
    s_pshape[(CATERVA_MAXDIM - s_ndim + i) % CATERVA_MAXDIM] = src->pshape[i];
}

/* Initialise a chunk buffer */
caterva_ctx_t *ctx = src->ctx;
int typesize = src->ctx->cparams.typesize;
int8_t *chunk = (int8_t *) ctx->alloc((size_t) src->psize * typesize);

/* Calculate the constants out of the for  */
int64_t aux[CATERVA_MAXDIM];
aux[7] = s_eshape[7] / s_pshape[7];
for (int i = CATERVA_MAXDIM - 2; i >= 0; i--) {
    aux[i] = s_eshape[i] / s_pshape[i] * aux[i + 1];
}

/* Fill array from schunk (chunk by chunk) */
int64_t desp[CATERVA_MAXDIM], r[CATERVA_MAXDIM];
for (int32_t ci = 0; ci < (int) src->esize / src->psize; ci++) {
    blosc2_schunk_decompress_chunk(src->sc, (int) ci, chunk,
    (size_t) src->psize * typesize);
    /* Calculate the coord. of the chunk first element in arr buffer */
    desp[7] = ci % aux[7] * s_pshape[7];
    for (int i = CATERVA_MAXDIM - 2; i >= 0; i--) {
        desp[i] = ci % (aux[i]) / (aux[i + 1]) * s_pshape[i];
```

```
}
/* Calculate if pad with 0 are needed in this chunk */
for (int i = CATERVA_MAXDIM - 1; i >= 0; i--) {
    if (desp[i] + s_pshape[i] > s_shape[i]) {
        r[i] = s_shape[i] - desp[i];
    } else {
        r[i] = s_pshape[i];
    }
}

/* Copy each line of data from chunk to arr */
int64_t s_coord_f, d_coord_f, s_a, d_a;
int64_t ii[CATERVA_MAXDIM];
for (ii[6] = 0; ii[6] < r[6]; ii[6]++) {
    for (ii[5] = 0; ii[5] < r[5]; ii[5]++) {
        for (ii[4] = 0; ii[4] < r[4]; ii[4]++) {
            for (ii[3] = 0; ii[3] < r[3]; ii[3]++) {
                for (ii[2] = 0; ii[2] < r[2]; ii[2]++) {
                    for (ii[1] = 0; ii[1] < r[1]; ii[1]++) {
                        for (ii[0] = 0; ii[0] < r[0]; ii[0]++) {
                            s_coord_f = 0;
                            s_a = s_pshape[7];
                            for (int i = CATERVA_MAXDIM - 2; i >= 0; i--) {
                                s_coord_f += ii[i] * s_a;
                                s_a *= s_pshape[i];
                            }
                            d_coord_f = desp[7];
                            d_a = s_shape[7];
                            for (int i = CATERVA_MAXDIM - 2; i >= 0; i--) {
                                d_coord_f += (desp[i] + ii[i]) * d_a;
                                d_a *= s_shape[i];
                            }
                            memcpy(&d_b[d_coord_f * typesize],
```

```
                                        &chunk[s_coord_f * typesize], r[7] * typesize);
                                }
                            }
                        }
                    }
                }
            }
        }
        ctx->free(chunk);
    } else {
        memcpy(dest, src->buf, src->size * (size_t) src->ctx->cparams.typesize);
    }
    return 0;
}

int caterva_to_buffer_2(caterva_array_t *src, void *dest) {
    if (src->storage == CATERVA_STORAGE_BLOSC) {
        int8_t *d_b = (int8_t *) dest;
        int64_t s_shape[CATERVA_MAXDIM];
        int64_t s_pshape[CATERVA_MAXDIM];
        int64_t s_eshape[CATERVA_MAXDIM];
        int8_t s_ndim = src->ndim;

        for (int i = 0; i < CATERVA_MAXDIM; ++i) {
            s_shape[(CATERVA_MAXDIM - s_ndim + i) % CATERVA_MAXDIM] = src->shape[i];
            s_eshape[(CATERVA_MAXDIM - s_ndim + i) % CATERVA_MAXDIM] = src->eshape[i];
            s_pshape[(CATERVA_MAXDIM - s_ndim + i) % CATERVA_MAXDIM] = src->pshape[i];
        }

        /* Initialise a chunk buffer */
        caterva_ctx_t *ctx = src->ctx;
        int typesize = src->sc->typesize;
```

```
size_t size_rchunk = (size_t) src->epsize * typesize;
size_t size_chunk = (size_t) src->psize * typesize;
int8_t *rchunk = (int8_t *) ctx->alloc(size_rchunk);
int8_t *chunk = (int8_t *) ctx->alloc(size_chunk);

/* Calculate the constants out of the for  */
int64_t aux[CATERVA_MAXDIM];
aux[7] = s_eshape[7] / s_pshape[7];
for (int i = CATERVA_MAXDIM - 2; i >= 0; i--) {
    aux[i] = s_eshape[i] / s_pshape[i] * aux[i + 1];
}

/* Fill array from schunk (chunk by chunk) */
int64_t desp[CATERVA_MAXDIM], r[CATERVA_MAXDIM];
for (int64_t ci = 0; ci < src->esize / src->psize; ci++) {
    blosc2_schunk_decompress_chunk(src->sc, (int) ci, rchunk,
    (size_t) src->epsize * typesize);
    caterva_derepart_chunk(chunk, size_chunk, rchunk, size_rchunk, src, src->ctx);

    /* Calculate the coord. of the chunk first element in arr buffer */
    desp[7] = ci % aux[7] * s_pshape[7];
    for (int i = CATERVA_MAXDIM - 2; i >= 0; i--) {
        desp[i] = ci % (aux[i]) / (aux[i + 1]) * s_pshape[i];
    }
    /* Calculate if pad with 0 are needed in this chunk */
    for (int i = CATERVA_MAXDIM - 1; i >= 0; i--) {
        if (desp[i] + s_pshape[i] > s_shape[i]) {
            r[i] = s_shape[i] - desp[i];
        } else {
            r[i] = s_pshape[i];
        }
    }
```

```
    /* Copy each line of data from chunk to arr */
    int64_t s_coord_f, d_coord_f, s_a, d_a;
    int64_t ii[CATERVA_MAXDIM];
    for (ii[6] = 0; ii[6] < r[6]; ii[6]++) {
        for (ii[5] = 0; ii[5] < r[5]; ii[5]++) {
            for (ii[4] = 0; ii[4] < r[4]; ii[4]++) {
                for (ii[3] = 0; ii[3] < r[3]; ii[3]++) {
                    for (ii[2] = 0; ii[2] < r[2]; ii[2]++) {
                        for (ii[1] = 0; ii[1] < r[1]; ii[1]++) {
                            for (ii[0] = 0; ii[0] < r[0]; ii[0]++) {
                                s_coord_f = 0;
                                s_a = s_pshape[7];
                                for (int i = CATERVA_MAXDIM - 2; i >= 0; i--) {
                                    s_coord_f += ii[i] * s_a;
                                    s_a *= s_pshape[i];
                                }
                                d_coord_f = desp[7];
                                d_a = s_shape[7];
                                for (int i = CATERVA_MAXDIM - 2; i >= 0; i--) {
                                    d_coord_f += (desp[i] + ii[i]) * d_a;
                                    d_a *= s_shape[i];
                                }
                                memcpy(&d_b[d_coord_f * typesize],
                                &chunk[s_coord_f * typesize], r[7] * typesize);
                            }
                        }
                    }
                }
            }
        }
    }
}
ctx->free(chunk);
```

```
        ctx->free(rchunk);
        return 0;
    } else {
        memcpy(dest, src->buf, src->size * (size_t) src->ctx->cparams.typesize);
    }

}

int caterva_get_slice_buffer(void *dest, caterva_array_t *src, caterva_dims_t *start,
                             caterva_dims_t *stop, caterva_dims_t *d_pshape) {
    uint8_t *bdest = dest;   // for allowing pointer arithmetic
    int64_t start_[CATERVA_MAXDIM];
    int64_t stop_[CATERVA_MAXDIM];
    int64_t d_pshape_[CATERVA_MAXDIM];
    int64_t s_pshape[CATERVA_MAXDIM];
    int64_t s_eshape[CATERVA_MAXDIM];
    int8_t s_ndim = src->ndim;

    if (src->storage == CATERVA_STORAGE_BLOSC) {
        for (int i = 0; i < CATERVA_MAXDIM; ++i) {
            start_[(CATERVA_MAXDIM - s_ndim + i) % CATERVA_MAXDIM] = start->dims[i];
            stop_[(CATERVA_MAXDIM - s_ndim + i) % CATERVA_MAXDIM] = stop->dims[i];
            d_pshape_[(CATERVA_MAXDIM - s_ndim + i) % CATERVA_MAXDIM] = d_pshape->dims[i];
            s_eshape[(CATERVA_MAXDIM - s_ndim + i) % CATERVA_MAXDIM] = src->eshape[i];
            s_pshape[(CATERVA_MAXDIM - s_ndim + i) % CATERVA_MAXDIM] = src->pshape[i];
        }

        // Acceleration path for the case where we are doing (1-dim) aligned chunk reads
        if ((s_ndim == 1) && (src->pshape[0] == d_pshape->dims[0]) &&
            (start->dims[0] % src->pshape[0] == 0) &&
            (stop->dims[0] % src->pshape[0] == 0)) {
            int nchunk = (int)(start->dims[0] / src->pshape[0]);
            // In case of an aligned read, decompress directly in destination
```

```
    blosc2_schunk_decompress_chunk(src->sc, nchunk, bdest,
    (size_t)src->psize * src->sc->typesize);
    return 0;
}

for (int j = 0; j < CATERVA_MAXDIM - s_ndim; ++j) {
    start_[j] = 0;
}
/* Create chunk buffers */
caterva_ctx_t *ctx = src->ctx;
int typesize = src->sc->typesize;

uint8_t *chunk;
bool local_cache;
if (src->part_cache.data == NULL) {
    chunk = (uint8_t *) ctx->alloc((size_t) src->psize * typesize);
    local_cache = true;
} else {
    chunk = src->part_cache.data;
    local_cache = false;
}
int64_t i_start[8], i_stop[8];
for (int i = 0; i < CATERVA_MAXDIM; ++i) {
    i_start[i] = start_[i] / s_pshape[i];
    i_stop[i] = (stop_[i] - 1) / s_pshape[i];
}

/* Calculate the used chunks */
int64_t ii[CATERVA_MAXDIM], jj[CATERVA_MAXDIM];
int64_t c_start[CATERVA_MAXDIM], c_stop[CATERVA_MAXDIM];
for (ii[0] = i_start[0]; ii[0] <= i_stop[0]; ++ii[0]) {
    ...
}
```

```
    if (local_cache) {
        ctx->free(chunk);
    }
} else {
    caterva_dims_t shape = caterva_get_shape(src);
    int64_t s_shape[CATERVA_MAXDIM];
    for (int i = 0; i < CATERVA_MAXDIM; ++i) {
        start_[(CATERVA_MAXDIM - s_ndim + i) % CATERVA_MAXDIM] = start->dims[i];
        stop_[(CATERVA_MAXDIM - s_ndim + i) % CATERVA_MAXDIM] = stop->dims[i];
        s_shape[(CATERVA_MAXDIM - s_ndim + i) % CATERVA_MAXDIM] = shape.dims[i];
        d_pshape_[(CATERVA_MAXDIM - s_ndim + i) % CATERVA_MAXDIM] = d_pshape->dims[i];
    }
    for (int j = 0; j < CATERVA_MAXDIM - s_ndim; ++j) {
        start_[j] = 0;
    }
    int64_t jj[CATERVA_MAXDIM];
    jj[7] = start_[7];
    for (jj[0] = start_[0]; jj[0] < stop_[0]; ++jj[0]) {
        for (jj[1] = start_[1]; jj[1] < stop_[1]; ++jj[1]) {
            for (jj[2] = start_[2]; jj[2] < stop_[2]; ++jj[2]) {
                for (jj[3] = start_[3]; jj[3] < stop_[3]; ++jj[3]) {
                    for (jj[4] = start_[4]; jj[4] < stop_[4]; ++jj[4]) {
                        for (jj[5] = start_[5]; jj[5] < stop_[5]; ++jj[5]) {
                            for (jj[6] = start_[6]; jj[6] < stop_[6]; ++jj[6]) {
                                int64_t chunk_pointer = 0;
                                int64_t chunk_pointer_inc = 1;
                                for (int i = CATERVA_MAXDIM - 1; i >= 0; --i) {
                                    chunk_pointer += jj[i] * chunk_pointer_inc;
                                    chunk_pointer_inc *= s_shape[i];
                                }
                                int64_t buf_pointer = 0;
                                int64_t buf_pointer_inc = 1;
                                for (int i = CATERVA_MAXDIM - 1; i >= 0; --i) {
```

```
                                    buf_pointer += (jj[i] - start_[i])
                                    * buf_pointer_inc;
                                    buf_pointer_inc *= d_pshape_[i];
                                }
                                memcpy(&bdest[buf_pointer
                                * src->ctx->cparams.typesize],
                                       &src->buf[chunk_pointer
                                       * src->ctx->cparams.typesize],
                                       (stop_[7] - start_[7])
                                       * src->ctx->cparams.typesize);
                            }
                        }
                    }
                }
            }
        }
    }
    return 0;
}


int caterva_get_slice_buffer_2(void *dest, caterva_array_t *src, caterva_dims_t *start,
                               caterva_dims_t *stop, caterva_dims_t *d_pshape) {
    uint8_t *bdest = dest;   // for allowing pointer arithmetic
    int64_t start_[CATERVA_MAXDIM];
    int64_t stop_[CATERVA_MAXDIM];
    int64_t d_pshape_[CATERVA_MAXDIM];
    int64_t s_pshape[CATERVA_MAXDIM];
    int64_t s_eshape[CATERVA_MAXDIM];
    int64_t s_spshape[CATERVA_MAXDIM];
    int64_t s_epshape[CATERVA_MAXDIM];
    int8_t s_ndim = src->ndim;
```

```
if (src->storage == CATERVA_STORAGE_BLOSC) {
    for (int i = 0; i < CATERVA_MAXDIM; ++i) {
        start_[(CATERVA_MAXDIM - s_ndim + i) % CATERVA_MAXDIM] = start->dims[i];
        stop_[(CATERVA_MAXDIM - s_ndim + i) % CATERVA_MAXDIM] = stop->dims[i];
        d_pshape_[(CATERVA_MAXDIM - s_ndim + i) % CATERVA_MAXDIM] = d_pshape->dims[i];
        s_eshape[(CATERVA_MAXDIM - s_ndim + i) % CATERVA_MAXDIM] = src->eshape[i];
        s_pshape[(CATERVA_MAXDIM - s_ndim + i) % CATERVA_MAXDIM] = src->pshape[i];
        s_epshape[(CATERVA_MAXDIM - s_ndim + i) % CATERVA_MAXDIM] = src->epshape[i];
        s_spshape[(CATERVA_MAXDIM - s_ndim + i) % CATERVA_MAXDIM] = src->spshape[i];
    }

    // Acceleration path for the case where we are doing (1-dim) aligned chunk reads
    if ((s_ndim == 1) && (src->pshape[0] == d_pshape->dims[0]) &&
        (start->dims[0] % src->pshape[0] == 0) && (stop->dims[0] % src->pshape[0] == 0)
        int nchunk = (int)(start->dims[0] / src->pshape[0]);
        // In case of an aligned read, decompress directly in destination
        blosc2_schunk_decompress_chunk(src->sc, nchunk, bdest, (size_t)src->psize
        * src->sc->typesize);
        return 0;
    }

    for (int j = 0; j < CATERVA_MAXDIM - s_ndim; ++j) {
        start_[j] = 0;
    }
    /* Create chunk buffers */
    caterva_ctx_t *ctx = src->ctx;
    int typesize = src->sc->typesize;

    uint8_t *chunk;
    uint8_t *spart;
    spart = ctx->alloc((size_t) src->spsize * typesize);
    int64_t i_start[CATERVA_MAXDIM], i_stop[CATERVA_MAXDIM];
```

```
for (int i = 0; i < CATERVA_MAXDIM; ++i) {
    i_start[i] = start_[i] / s_pshape[i];
    i_stop[i] = (stop_[i] - 1) / s_pshape[i];
}

/* Calculate the used chunks */
bool needs_free = false;
int64_t ii[CATERVA_MAXDIM], jj[CATERVA_MAXDIM], kk[CATERVA_MAXDIM];
int64_t j_start[CATERVA_MAXDIM], j_stop[CATERVA_MAXDIM];
int64_t sp_start[CATERVA_MAXDIM], sp_stop[CATERVA_MAXDIM];
for (ii[0] = i_start[0]; ii[0] <= i_stop[0]; ++ii[0]) {
    for (ii[1] = i_start[1]; ii[1] <= i_stop[1]; ++ii[1]) {
        for (ii[2] = i_start[2]; ii[2] <= i_stop[2]; ++ii[2]) {
            for (ii[3] = i_start[3]; ii[3] <= i_stop[3]; ++ii[3]) {
                for (ii[4] = i_start[4]; ii[4] <= i_stop[4]; ++ii[4]) {
                    for (ii[5] = i_start[5]; ii[5] <= i_stop[5]; ++ii[5]) {
                        for (ii[6] = i_start[6]; ii[6] <= i_stop[6]; ++ii[6]) {
                            for (ii[7] = i_start[7]; ii[7] <= i_stop[7]; ++ii[7]) {
                                /* Get the chunk ii */
                                int nchunk = 0;
                                int inc = 1;
                                for (int i = CATERVA_MAXDIM - 1; i >= 0; --i) {
                                    nchunk += (int) (ii[i] * inc);
                                    inc *= (int) (s_eshape[i] / s_pshape[i]);
                                }

                                if ((src->part_cache.data == NULL)
                                || (src->part_cache.nchunk != nchunk)) {
                                    blosc2_schunk_get_chunk(src->sc, nchunk,
                                    &chunk, &needs_free);
                                }

                                if (src->part_cache.data != NULL) {
```

```
                                        src->part_cache.nchunk = nchunk;
                                    }

                                    /* Calculate the used subpartitions */
                                    for (int i = 0; i < CATERVA_MAXDIM; ++i) {
                                        ...
                                    }
                                    for (jj[0] = j_start[0]; jj[0] <= j_stop[0];
                                    ++jj[0]) {
                                        ...
                                    }
                                }
                            }
                        }
                    }
                }
            }
        }
        int buf_size = 1;
        for(int i=0; i< CATERVA_MAXDIM; i++){
            buf_size *= d_pshape_[i];
        }

        ctx->free(spart);
        if(needs_free){
            ctx->free(chunk);
        }
    } else {
        caterva_dims_t shape = caterva_get_shape(src);
        int64_t s_shape[CATERVA_MAXDIM];
        for (int i = 0; i < CATERVA_MAXDIM; ++i) {
            start_[(CATERVA_MAXDIM - s_ndim + i) % CATERVA_MAXDIM] = start->dims[i];
```

```
    stop_[(CATERVA_MAXDIM - s_ndim + i) % CATERVA_MAXDIM] = stop->dims[i];
    s_shape[(CATERVA_MAXDIM - s_ndim + i) % CATERVA_MAXDIM] = shape.dims[i];
    d_pshape_[(CATERVA_MAXDIM - s_ndim + i) % CATERVA_MAXDIM] = d_pshape->dims[i];
}
for (int j = 0; j < CATERVA_MAXDIM - s_ndim; ++j) {
    start_[j] = 0;
}
int64_t jj[CATERVA_MAXDIM];
jj[7] = start_[7];
for (jj[0] = start_[0]; jj[0] < stop_[0]; ++jj[0]) {
    for (jj[1] = start_[1]; jj[1] < stop_[1]; ++jj[1]) {
        for (jj[2] = start_[2]; jj[2] < stop_[2]; ++jj[2]) {
            for (jj[3] = start_[3]; jj[3] < stop_[3]; ++jj[3]) {
                for (jj[4] = start_[4]; jj[4] < stop_[4]; ++jj[4]) {
                    for (jj[5] = start_[5]; jj[5] < stop_[5]; ++jj[5]) {
                        for (jj[6] = start_[6]; jj[6] < stop_[6]; ++jj[6]) {
                            int64_t chunk_pointer = 0;
                            int64_t chunk_pointer_inc = 1;
                            for (int i = CATERVA_MAXDIM - 1; i >= 0; --i) {
                                chunk_pointer += jj[i] * chunk_pointer_inc;
                                chunk_pointer_inc *= s_shape[i];
                            }
                            int64_t buf_pointer = 0;
                            int64_t buf_pointer_inc = 1;
                            for (int i = CATERVA_MAXDIM - 1; i >= 0; --i) {
                                buf_pointer += (jj[i] - start_[i])
                                * buf_pointer_inc;
                                buf_pointer_inc *= d_pshape_[i];
                            }
                            memcpy(&bdest[buf_pointer * src->ctx->cparams.typesize],
                                   &src->buf[chunk_pointer
                                   * src->ctx->cparams.typesize],
                                   (stop_[7] - start_[7])
```

```
                                        * src->ctx->cparams.typesize);
                                }
                            }
                        }
                    }
                }
            }
        }
    }
    return 0;
}


int caterva_get_slice_buffer_no_copy(void **dest, caterva_array_t *src,
caterva_dims_t *start, caterva_dims_t *stop, caterva_dims_t *d_pshape) {
    CATERVA_UNUSED_PARAM(d_pshape);
    int64_t start_[CATERVA_MAXDIM];
    int64_t stop_[CATERVA_MAXDIM];
    int8_t s_ndim = src->ndim;

    caterva_dims_t shape = caterva_get_shape(src);
    int64_t s_shape[CATERVA_MAXDIM];
    for (int i = 0; i < CATERVA_MAXDIM; ++i) {
        start_[(CATERVA_MAXDIM - s_ndim + i) % CATERVA_MAXDIM] = start->dims[i];
        stop_[(CATERVA_MAXDIM - s_ndim + i) % CATERVA_MAXDIM] = stop->dims[i];
        s_shape[(CATERVA_MAXDIM - s_ndim + i) % CATERVA_MAXDIM] = shape.dims[i];
    }
    for (int j = 0; j < CATERVA_MAXDIM - s_ndim; ++j) {
        start_[j] = 0;
    }

    int64_t chunk_pointer = 0;
    int64_t chunk_pointer_inc = 1;
```

```
    for (int i = CATERVA_MAXDIM - 1; i >= 0; --i) {
        chunk_pointer += start_[i] * chunk_pointer_inc;
        chunk_pointer_inc *= s_shape[i];
    }
    *dest = &src->buf[chunk_pointer * src->ctx->cparams.typesize];

    return 0;
}

int caterva_set_slice_buffer(caterva_array_t *dest, void *src, caterva_dims_t *start,
                             caterva_dims_t *stop) {
    if (dest->storage == CATERVA_STORAGE_BLOSC) {
        return -1;
    }

    uint8_t *bsrc = src;   // for allowing pointer arithmetic
    int64_t start_[CATERVA_MAXDIM];
    int64_t stop_[CATERVA_MAXDIM];
    int8_t s_ndim = dest->ndim;

    if (dest->storage == CATERVA_STORAGE_PLAINBUFFER) {
        caterva_dims_t shape = caterva_get_shape(dest);
        int64_t d_shape[CATERVA_MAXDIM];
        int64_t s_shape[CATERVA_MAXDIM];
        for (int i = 0; i < CATERVA_MAXDIM; ++i) {
            start_[(CATERVA_MAXDIM - s_ndim + i) % CATERVA_MAXDIM] = start->dims[i];
            stop_[(CATERVA_MAXDIM - s_ndim + i) % CATERVA_MAXDIM] = stop->dims[i];
            d_shape[(CATERVA_MAXDIM - s_ndim + i) % CATERVA_MAXDIM] = (stop->dims[i]
            - start->dims[i]);
            s_shape[(CATERVA_MAXDIM - s_ndim + i) % CATERVA_MAXDIM] = shape.dims[i];
        }
        for (int j = 0; j < CATERVA_MAXDIM - s_ndim; ++j) {
            start_[j] = 0;
```

```
    d_shape[j] = 1;

}
int64_t jj[CATERVA_MAXDIM];
jj[7] = start_[7];
for (jj[0] = start_[0]; jj[0] < stop_[0]; ++jj[0]) {
    for (jj[1] = start_[1]; jj[1] < stop_[1]; ++jj[1]) {
        for (jj[2] = start_[2]; jj[2] < stop_[2]; ++jj[2]) {
            for (jj[3] = start_[3]; jj[3] < stop_[3]; ++jj[3]) {
                for (jj[4] = start_[4]; jj[4] < stop_[4]; ++jj[4]) {
                    for (jj[5] = start_[5]; jj[5] < stop_[5]; ++jj[5]) {
                        for (jj[6] = start_[6]; jj[6] < stop_[6]; ++jj[6]) {
                            int64_t chunk_pointer = 0;
                            int64_t chunk_pointer_inc = 1;
                            for (int i = CATERVA_MAXDIM - 1; i >= 0; --i) {
                                chunk_pointer += jj[i] * chunk_pointer_inc;
                                chunk_pointer_inc *= s_shape[i];
                            }
                            int64_t buf_pointer = 0;
                            int64_t buf_pointer_inc = 1;
                            for (int i = CATERVA_MAXDIM - 1; i >= 0; --i) {
                                buf_pointer += (jj[i] - start_[i]) *
                                buf_pointer_inc;
                                buf_pointer_inc *= d_shape[i];
                            }
                            memcpy(&dest->buf[chunk_pointer
                            * dest->ctx->cparams.typesize],
                                   &bsrc[buf_pointer
                                   * dest->ctx->cparams.typesize],
                                   (stop_[7] - start_[7])
                                   * dest->ctx->cparams.typesize);
                        }
                    }
```

```
                        }
                    }
                }
            }
        }
    }

    return 0;
}


int caterva_get_slice(caterva_array_t *dest, caterva_array_t *src, caterva_dims_t
*start, caterva_dims_t *stop){
    if (start->ndim != stop->ndim) {
        return -1;
    }
    if (start->ndim != src->ndim) {
        return -1;
    }

    caterva_ctx_t *ctx = src->ctx;
    int typesize = ctx->cparams.typesize;
    int64_t shape_[CATERVA_MAXDIM];
    for (int i = 0; i < start->ndim; ++i) {
        shape_[i] = stop->dims[i] - start->dims[i];
    }
    for (int i = (int) start->ndim; i < CATERVA_MAXDIM; ++i) {
        shape_[i] = 1;
        start->dims[i] = 0;
    }
    caterva_dims_t shape = caterva_new_dims(shape_, start->ndim);
    caterva_update_shape(dest, &shape);
```

```
    if (dest->storage == CATERVA_STORAGE_BLOSC) {
        uint8_t *chunk = ctx->alloc((size_t) dest->psize * typesize);
        int64_t d_pshape[CATERVA_MAXDIM];
        int64_t d_start[CATERVA_MAXDIM];
        int64_t d_stop[CATERVA_MAXDIM];
        int8_t d_ndim = dest->ndim;
        for (int i = 0; i < CATERVA_MAXDIM; ++i) {
            d_pshape[(CATERVA_MAXDIM - d_ndim + i) % CATERVA_MAXDIM] = dest->pshape[i];
            d_start[(CATERVA_MAXDIM - d_ndim + i) % CATERVA_MAXDIM] = start->dims[i];
            d_stop[(CATERVA_MAXDIM - d_ndim + i) % CATERVA_MAXDIM] = stop->dims[i];
        }
        int64_t ii[CATERVA_MAXDIM];
        for (ii[0] = d_start[0]; ii[0] < d_stop[0]; ii[0] += d_pshape[0]) {
            ...
        }
        ctx->free(chunk);
    } else {
        uint64_t size = 1;
        for (int i = 0; i < stop->ndim; ++i) {
            size *= stop->dims[i] - start->dims[i];
        }
        dest->buf = malloc(size * typesize);
        caterva_get_slice_buffer(dest->buf, src, start, stop, &shape);
    }
    dest->filled = true;
    return 0;
}

int caterva_get_slice_2(caterva_array_t *dest, caterva_array_t *src,
caterva_dims_t *start, caterva_dims_t *stop){
    if (start->ndim != stop->ndim) {
        return -1;
    }
```

```
if (start->ndim != src->ndim) {
    return -1;
}
for (int i = 0; i < CATERVA_MAXDIM; i++) {
    if ((start->dims[i] < 0) || (stop->dims[i] > src->shape[i])) {
        return -2;
    }
    if (start->dims[i] > stop->dims[i]) {
        return -3;
    }
}

caterva_ctx_t *ctx = src->ctx;
int typesize = ctx->cparams.typesize;
int64_t shape_[CATERVA_MAXDIM];
for (int i = 0; i < start->ndim; ++i) {
    shape_[i] = stop->dims[i] - start->dims[i];
}
for (int i = (int) start->ndim; i < CATERVA_MAXDIM; ++i) {
    shape_[i] = 1;
    start->dims[i] = 0;
}
caterva_dims_t shape = caterva_new_dims(shape_, start->ndim);
caterva_update_shape(dest, &shape);

if (dest->storage == CATERVA_STORAGE_BLOSC) {
    uint8_t *chunk = ctx->alloc((size_t) dest->psize * typesize);
    int64_t d_pshape[CATERVA_MAXDIM];
    int64_t d_start[CATERVA_MAXDIM];
    int64_t d_stop[CATERVA_MAXDIM];
    int8_t d_ndim = dest->ndim;
    for (int i = 0; i < CATERVA_MAXDIM; ++i) {
        d_pshape[(CATERVA_MAXDIM - d_ndim + i) % CATERVA_MAXDIM]
```

```
            = dest->next_pshape[i];
            d_start[(CATERVA_MAXDIM - d_ndim + i) % CATERVA_MAXDIM] = start->dims[i];
            d_stop[(CATERVA_MAXDIM - d_ndim + i) % CATERVA_MAXDIM] = stop->dims[i];
        }
        // Append each src slice chunk to dest
        int64_t ii[CATERVA_MAXDIM];
        int64_t appended_shape[CATERVA_MAXDIM];
        for (ii[0] = d_start[0]; ii[0] < d_stop[0]; ii[0] += appended_shape[0]) {
            ...
        }
        ctx->free(chunk);
    } else {
        uint64_t size = 1;
        for (int i = 0; i < stop->ndim; ++i) {
            size *= stop->dims[i] - start->dims[i];
        }
        dest->buf = malloc(size * typesize);
        caterva_get_slice_buffer(dest->buf, src, start, stop, &shape);
    }
    dest->filled = true;
    return 0;
}

int caterva_repart(caterva_array_t *dest, caterva_array_t *src) {
    if (src->storage != CATERVA_STORAGE_BLOSC) {
        return -1;
    }
    int64_t start_[CATERVA_MAXDIM] = {0, 0, 0, 0, 0, 0, 0, 0};
    caterva_dims_t start = caterva_new_dims(start_, dest->ndim);
    int64_t stop_[CATERVA_MAXDIM];
    for (int i = 0; i < dest->ndim; ++i) {
        stop_[i] = src->shape[i];
    }
```

```
    caterva_dims_t stop = caterva_new_dims(stop_, dest->ndim);
    caterva_get_slice(dest, src, &start, &stop);
    return 0;
}

int caterva_squeeze(caterva_array_t *src) {
    uint8_t nones = 0;
    int64_t newshape_[CATERVA_MAXDIM];
    int32_t newpshape_[CATERVA_MAXDIM];

    if (src->storage == CATERVA_STORAGE_BLOSC) {
        for (int i = 0; i < src->ndim; ++i) {
            if (src->shape[i] != 1) {
                newshape_[nones] = src->shape[i];
                newpshape_[nones] = src->pshape[i];
                nones += 1;
            }
        }
        for (int i = 0; i < CATERVA_MAXDIM; ++i) {
            if (i < nones) {
                src->pshape[i] = newpshape_[i];
            } else {
                src->pshape[i] = 1;
            }
        }
    } else {
        for (int i = 0; i < src->ndim; ++i) {
            if (src->shape[i] != 1) {
                newshape_[nones] = src->shape[i];
                nones += 1;
            }
        }
    }
```

```
    src->ndim = nones;
    caterva_dims_t newshape = caterva_new_dims(newshape_, nones);
    caterva_update_shape(src, &newshape);

    return 0;
}


int caterva_squeeze_2(caterva_array_t *src) {
    uint8_t nones = 0;
    int64_t newshape_[CATERVA_MAXDIM];
    int32_t newpshape_[CATERVA_MAXDIM];
    int32_t newspshape_[CATERVA_MAXDIM];
    int32_t newepshape_[CATERVA_MAXDIM];

    if (src->storage == CATERVA_STORAGE_BLOSC) {
        for (int i = 0; i < src->ndim; ++i) {
            if (src->shape[i] != 1) {
                newshape_[nones] = src->shape[i];
                newpshape_[nones] = src->pshape[i];
                newspshape_[nones] = src->spshape[i];
                newepshape_[nones] = src->epshape[i];
                nones += 1;
            }
        }
        for (int i = 0; i < CATERVA_MAXDIM; ++i) {
            if (i < nones) {
                src->pshape[i] = newpshape_[i];
                src->spshape[i] = newspshape_[i];
                src->epshape[i] = newepshape_[i];
            } else {
                src->pshape[i] = 1;
                src->spshape[i] = 1;
```

```
            }
        }
    } else {
        for (int i = 0; i < src->ndim; ++i) {
            if (src->shape[i] != 1) {
                newshape_[nones] = src->shape[i];
                nones += 1;
            }
        }
    }
    src->ndim = nones;
    caterva_dims_t newshape = caterva_new_dims(newshape_, nones);
    caterva_update_shape(src, &newshape);

    return 0;
}


caterva_dims_t caterva_get_shape(caterva_array_t *src){
    caterva_dims_t shape = caterva_new_dims(src->shape, src->ndim);
    return shape;
}


caterva_dims_t caterva_get_pshape(caterva_array_t *src) {
    caterva_dims_t pshape;
    for (int i = 0; i < src->ndim; ++i) {
        pshape.dims[i] = src->pshape[i];
    }
    pshape.ndim = src->ndim;
    return pshape;
}
```

```
caterva_dims_t caterva_get_spshape(caterva_array_t *src) {
    caterva_dims_t spshape;
    for (int i = 0; i < src->ndim; ++i) {
        spshape.dims[i] = src->spshape[i];
    }
    spshape.ndim = src->ndim;
    return spshape;
}

int caterva_copy(caterva_array_t *dest, caterva_array_t *src) {
    caterva_dims_t shape = caterva_new_dims(src->shape, src->ndim);
    if (src->storage == CATERVA_STORAGE_PLAINBUFFER) {
        if (dest->storage == CATERVA_STORAGE_PLAINBUFFER) {
            caterva_update_shape(dest, &shape);
            dest->buf = malloc((size_t) dest->size * dest->ctx->cparams.typesize);
            memcpy(dest->buf, src->buf, dest->size * dest->ctx->cparams.typesize);
            dest->filled = true;
        } else {
            caterva_from_buffer(dest, &shape, src->buf);
        }
    } else {
        if (dest->storage == CATERVA_STORAGE_PLAINBUFFER) {
            caterva_update_shape(dest, &shape);
            dest->buf = malloc((size_t) dest->size * dest->ctx->cparams.typesize);
            caterva_to_buffer(src, dest->buf);
            dest->filled = true;
        } else {
            caterva_repart(dest, src);
        }
    }
    return 0;
}
```